

Desynchronizing Synchronous Programs by Modes

Jens Brandt, Mike Gemünde and Klaus Schneider

Embedded Systems Group

Department of Computer Science, University of Kaiserslautern

P.O. Box 3049, 67659 Kaiserslautern, Germany

<http://es.cs.uni-kl.de>

Abstract

The synchronous programming paradigm simplifies the specification and verification of reactive systems. However, synchronous programs must be often implemented on architectures that do not follow this model of computation (like distributed systems or systems-on-a-chip). This gives rise to desynchronization techniques, which map the synchronous program to a platform without global time while preserving the original synchronous semantics.

In this paper, we present a new approach to desynchronize synchronous programs. Our approach is based on partitioning the system and deriving suited computation modes for that partitioning. These computation modes dynamically decouple the components of a system by eliminating temporarily unnecessary and undesired computations and communications. We present several variations of the general concept and discuss their pros and cons. Finally, we illustrate our approach with the help of several small examples.

1. Introduction

Synchronous languages [2] like Esterel [4] and its variants [11, 18] offer many important features to fulfill the requirements imposed by embedded systems: first, it is possible to generate efficient software *and* hardware from the same synchronous program [3], which allows fast simulation of the application-specific hardware as well as late changes of the hardware-software partitioning of a system. Second, it is possible to determine tight bounds on the reaction time by a simplified worst-case execution time analysis (since loops do not appear in reaction steps). Third, the formal semantics of these languages allows the application of formal methods to formally prove (1) the correctness of the compile algorithm and (2) the correctness of a particular program with respect to given formal specifications [15–17, 21].

All these advantages are due to the paradigm of perfect synchrony, which assumes that the overall system behavior is divided into a sequence of reactions, so called macro-steps. Computations within such a reaction are completed in zero-time, and the results are instantaneously broadcasted to all components in the system. Macro steps provide a

good abstraction level so that perfect synchrony simplifies reactive programming in that developers do not have to bother about many low-level details like timing, synchronization and scheduling. On the other hand, this paradigm has some drawbacks that make the compilation and synthesis of synchronous programs quite difficult: While causality [19, 20, 22, 23] and schizophrenia [21, 24] problems challenge compilers, the synthesis procedures have to map a synchronous program to a target architecture that does, in general, not provide perfect synchrony.

The mismatch between the synchronous model used for the development of a system and most real-world implementation environments is apparent: embedded applications in the automotive or avionic industry are based on a heterogeneous set of distributed processing elements. Similarly, state-of-the-art microprocessors are no longer implemented on monolithic synchronous chips, and are instead implemented as multi-core processors as a system-on-a-chip. Using these architectures, it is in general not reasonable to maintain a global clock, since the speed of the individual components significantly varies. As the slowest component defines the global speed, the resulting performance would be unacceptable.

Multi-clocked systems can be seen as one solution. These systems use different clocks depending on the considered abstraction level: the higher the abstraction level, the slower is the corresponding clock. A somehow extreme case of multi-clock systems are GALS (globally asynchronous, locally synchronous) systems: in a GALS, a number of synchronous components are connected asynchronously, exchanging data by sending messages over asynchronous channels. This model is very general and perfectly matches real-world systems. Hence, existing methods and tools for synchronous systems can be used throughout the whole development process, while the implementation architecture does not necessarily correspond to the synchronous paradigm. Therefore, synthesis tools for synchronous languages should consider the GALS model as a target architecture. However, they must add some synchronization logic in order to preserve the semantics, i.e. to guarantee that the behavior of the synthesized system corresponds to the original one.

In the simplest case, a synchronous component can be integrated in an asynchronous environment by using a wrap-

per that reads the asynchronous inputs of the component and triggers its execution when all its inputs are available. The synchronous system then performs a macro-step that computes its next state and also generates values for all of its outputs. As this naive implementation corresponds to the emulation of a global clock, it does not scale very well. Hence, more sophisticated protocols are needed that consider the data dependencies of the synchronous system. In general, this process is called desynchronization.

Desynchronization of synchronous programs involves two aspects:

- First, the temporal desynchronization: The synchronous paradigm postulates that all micro-steps of a system are executed simultaneously. In many cases, however, this restriction is too strong: For example, if there is no dependency between two threads, then there is no reason why both should be forced to run in synchronous locksteps. Instead, one could allow each component to react as soon as possible to increase the overall performance of the system. As different parts of the system will then execute parts of different macro-steps, their output values must be resynchronized at the system boundary. This aspect is related to latency-insensitive design (coping with different latencies).
- Second, desynchronization by lazy evaluation: In synchronous systems, all variables have a unique value for each macro-step. This value is immediately available for all other parts of the system. Obviously, not all values are always interesting for all other components. Instead, depending on the current state, only *some* inputs are required to compute only *some* outputs (that will be read by the other components). Hence, developers are interested in eliminating unnecessary computation and communication between desynchronized components.

Several researchers have already considered the desynchronization of synchronous programs, as in particular, the pioneering work of Potop et al. [1, 12–14]. Their definitions of endochrony and isochrony, which are introduced in [1, 13], are intended to answer the question which systems can be desynchronized safely without additional effort while preserving the original behavior with respect to trace-equivalence. As the original definitions lead to complex analyses (in particular the analysis is not modular), a weak version has been defined [12], which is simpler to check, but still sufficient in practice. Finally, [14] presents another variant, which is based on the finer-grained model. While previous publications have used synchronous transition systems at the macro-step level, the more intuitive level of micro-steps is used in [14]. This also transfers the whole theory in a causal context, and sending and receiving of messages is explicitly visible.

In this paper, we present a new method for the construction of desynchronized systems from a given synchronous

program. In contrast to the frameworks based on endochrony and isochrony, we have a slightly different starting point and demand a stronger property preservation: we aim at implementing a desynchronized system, which has exactly the same observable behavior at its interface as the original synchronous system (in terms of bisimulation equivalence). However, while the behavior observed at the interface is still the same, the internal implementation does no longer follow the synchronous paradigm by computing for *all* variables values at *every* point of time. Instead, values of internal variables are only computed if necessary, i.e. if required to determine the outputs. Our main contribution in this context is the introduction of *computation modes* which are responsible for the actual decoupling of the individual components. They are the key to the elimination of unnecessary synchronization and computation, which are due to the synchronous paradigm.

The rest of the paper is structured as follows: Section 2 describes the starting point of our desynchronization approach: we consider guarded actions, which are obtained as intermediate representation by the compilation of a synchronous program. Then, Section 3 outlines our view of the desynchronization task and relates it to previous work. Section 4 focuses on the partitioning of the system. Section 5 introduces modes and sketches several variations of the basic concept with their pros and cons. Finally, we draw some preliminary conclusions in Section 6.

2. Synchronous Guarded Actions

The compilation of synchronous languages is more difficult than the compilation of traditional sequential languages. Special problems like causality cycles and schizophrenic statements have to be solved by the compilers [5, 20, 21]. It is therefore reasonable and natural to split the overall compilation process into several phases, such that the first steps are independent of the realization that is finally used. A common intermediate format of the compilation are *guarded actions* [6, 8–10]. Due to their simplicity, we use them as the starting point for our desynchronization approach. At the level of guarded actions, all sophisticated control-flow constructs synchronous languages usually provide have already been eliminated by the compiler [21]. Thus, we can focus on the actual goal of desynchronization.

Hence, the synchronous systems we consider in the following are defined by sets of guarded actions of the form $\gamma \Rightarrow \mathcal{A}$. The Boolean condition γ is called the guard, and \mathcal{A} is called the action of the guarded action, which corresponds to an action of the source language. In our case, these are the assignments of the synchronous program, i.e. the guarded actions have either the form $\gamma \Rightarrow x = \tau$ (for an immediate assignment) or $\gamma \Rightarrow \text{next}(x) = \tau$ (for a delayed assignment). Guarded actions are generated for both the data flow and the control flow of the program. The data flow

consists of all assignments of the program and determines the values of all declared variables. The control flow consists of guards $\gamma \Rightarrow \text{next}(\ell) = \text{true}$ where γ is a condition that is responsible for moving the control flow at the next point of time to the program location ℓ .

The semantics of these guarded actions is simple: In each macro-step, all guards are *simultaneously* checked. If a guard is true, its action is executed *immediately* (and simultaneously to the other activated ones): Immediate assignments assign the computed value instantaneously to the variable on the left-hand side of the assignment, while the delayed assignments are deferred to the next macro-step. We assume that a preceding causality analysis [19, 20, 22, 23] has already checked that the set of guarded actions is free of causality cycles. In general, the synchronous model of computation may lead to cyclic dependencies between the actions of the system since immediate assignments can instantaneously modify guards of actions. We will consider this issue in Section 4.

3. Desynchronizing Synchronous Systems

3.1. The Desynchronization Task

Synchronous systems are based on the paradigm of perfect synchrony. This paradigm relies on the fact that the overall computation is divided into a sequence of reactions, so-called macro-steps. If a system is decomposed into a number of parallel components, all of them run in lockstep and perform the mentioned operations. In contrast to this, systems that are asynchronously composed usually communicate over FIFO channels. Since there is no global clock in an asynchronous system, each component runs independently of the other components. Their composition is done by only unifying their input and output channels. Thus, their behavior can no longer be split into a sequence of steps or reactions.

Before we consider the desynchronization problem, we want to highlight a detail that is important for the rest of the paper: the distinction between *strictly synchronous* systems and *synchronous* systems.

- In a *strictly synchronous system*, every variable has a unique value at every point of time. The set of possible values is determined by the type of the variable, and the actual value is determined by the current or previous macro-steps that currently modify or modified the value of the variable. In a macro-step i , the system reads *all of the current inputs* x_1^i, \dots, x_m^i and computes *all current outputs* y_1^i, \dots, y_n^i . As the concurrently running components of the system also follow this paradigm and communicate over shared variables, all values of internal variables are also always computed. In case there is currently no action that modifies the value of a variable y_k , a default value defined by the declaration

of the variable is taken: memorized variables keep their previous value, while event variables are reset to a default value. This view is taken e. g. by the synchronous language Quartz [18] as well as by hardware circuits.

- Other synchronous languages, e. g. Esterel and Lustre are not strict in the above sense: instead of the values determined by the type of a variable, a variable may also have no value which is described by a special value \perp . This value is motivated by the view that components communicate with each other by emitting values. In case no value is emitted, the output variable has the special value \perp meaning an absence of a value. Languages like Esterel and Lustre are able to react to the absence of an event in that actions are executed if a particular variable has the absent value.

In principle, both views are equivalent, since \perp can be regarded as an additional value of each data type. Then, checking presence or absence of values is just a comparison with \perp . However, problems - in particular many problems related to desynchronization - must be reformulated when seen from either view.

For instance, one usually tries to minimize the communication between asynchronously running components. For ordinary synchronous systems, an apparent approach is to exchange information only if a true event (not equal to \perp) occurs, i.e. the absence of events corresponds to the absence of communication. However, this has problematic consequences: Since a delayed reception of an event cannot be distinguished from its absence, the absence of an event can no longer be detected. Previously, each component saw for each of its inputs x_i a sequence of values for each point of time $\langle x_i^0, x_i^1, \dots \rangle$, hence, the component was able to count the number of steps by just counting the number of previously received values of variable x_i . Skipping one or more of these values x_i^j like the absence values \perp prohibits the resynchronization of the data streams. Thus, absence events of synchronous systems can, in general, not be neglected in a desynchronized implementation.

This problem is considered in the framework based on endochrony and isochrony: Both concepts formalize the set of systems that are able to neglect absence values \perp in communication while still producing equivalent results. Systems are thereby ‘equivalent’ if their output streams are the same after the removal of absence events. Formally, let S^a denote the compression of a sequence S of values, i.e. the removal of all absence values \perp :

$$\langle y^0, y^1, y^2, \dots \rangle^a = \langle y^{k_0}, y^{k_1}, y^{k_2}, \dots \rangle$$

where k_0, k_1, k_2 is a subsequence of $k = \langle 0, 1, 2, \dots \rangle$ such that $y^k \neq \perp$. For all outputs of the original system y and the corresponding ones of the desynchronized system \hat{y} , the following property is guaranteed:

$$\langle y^0, y^1, y^2, \dots \rangle^a = \langle \hat{y}^0, \hat{y}^1, \hat{y}^2, \dots \rangle$$

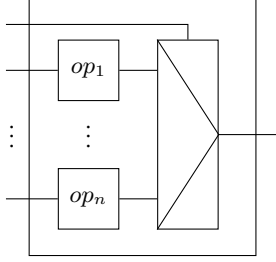


Figure 1. ALU Example

The main result of the endochrony and isochrony work is that in systems composed of endochronous components, absent values \perp do not need to be transmitted. Thus, these systems can be safely desynchronized without additional effort. The components will be able to reconstruct the synchronous reactions without explicit absence values. Thereby, a component is said to be endochronous if it can infer the information whether an event is present from the values of the other events that are present in the same step. Two components are isochronous if they agree on the presence and absence of values depending on the values which they both know. For details and a formal presentation of the approach, see [1, 12–14].

In strictly synchronous systems, this problem does not exist, and therefore, the question whether a module in these systems is endochronous or isochronous is ill-posed. Clearly, this question is only reasonable for modules that may produce values of a strict subset of its outputs. Even for Esterel, this only makes sense with the interpretation that *absence means not sending values*.

Nevertheless, the problem can be reformulated in strictly synchronous systems as follows: Assume that the values of variables are only transmitted when they change. Again, this saves communication but it introduces synchronization problems. However, this paper presents a different approach to the minimization of communication, which also aims at minimizing computation.

3.2. Our Approach

In contrast to the previous approach, we do not try to eliminate any designated value, but we aim at eliminating communication and computation that does not contribute to the result. For example, consider an ALU, which consists of several functional units, one for each arithmetic operation, and a multiplexer which selects the right result depending on the instruction (see Figures 1). Whatever the required operation is, obviously not all units need to compute their results and send them to the multiplexer. Apart from the unnecessary slow reaction time - a fully synchronous implementation always requires as long as the slowest operation (consider a simple adder in parallel to a division unit) -

the unconditional computation of all intermediate results wastes energy in the hardware circuit, and the unconditional communication slows down the communication (if all the channels from the individual units to the multiplexer are mapped to the same physical medium). Instead of this wasteful scheme, we aim at implementing the system so that only the unit of the chosen operation is activated, and only its result is sent to the multiplexer. This approach is quite different to the omission of absence values: we do not want the sender to decide which information is important, but the omission of values is determined by the receiver.

Another important difference to previous work is our property preservation criterion. Potop et al. focus, as already explained above, on the trace-equivalence of the synchronous and the asynchronous implementation. They create a system that has an asynchronous interface to the environment as illustrated by the dashed lines in Figure 2.

We want to keep the synchronous interface of the overall system, instead. This allows us to create a desynchronized system that preserves *all* properties of the original synchronous system, i. e. a system that cannot be distinguished from the original one (which is only possible if the interface is a synchronous one). If only trace equivalence is used, specifications of the synchronous system can be only transferred partially to the asynchronous world: e. g. if temporal logics have been used, only variables with direct causal relations can be used below the same temporal operator. Several concurrent streams cannot be related in general: e. g. assume that y_1 and y_2 are the outputs of component M_3'' . Then, specifications like $G(y_1 \vee y_2)$ do not make sense any more.

Hence, we keep the synchronous interface. Nevertheless, calculations inside the system do not have to follow this paradigm (see Figure 2). We aim at desynchronizing a system by changing the internal behavior while preserving the interface behavior. In particular, not all components of a system need to be in the same macro-step. This gives some freedom for a more efficient execution of the synchronous system. For the desynchronization, we can (re)partition the system, composing it of a completely different set of components and replacing their synchronous communication by an asynchronous one. The relationship between the original and the derived system is formalized by the following definition.

Definition 1 (Property Preservation): For a given synchronous system S consisting of synchronously running modules $S = C_1 \parallel \dots \parallel C_n$, a corresponding system S' is said to be a *correct desynchronization* if the following properties hold: (1) It consists of asynchronously running components $S' = C'_1 \parallel \dots \parallel C'_m$. (2) It fulfills exactly the same properties as the original system. Formally, for given system S or components C_i , a correct desynchronization determines components C'_i with the following property:

$$C_1 \parallel \dots \parallel C_n \models \phi \Leftrightarrow C'_1 \parallel \dots \parallel C'_m \models \phi$$

where ϕ is an arbitrary property of the system. If the set

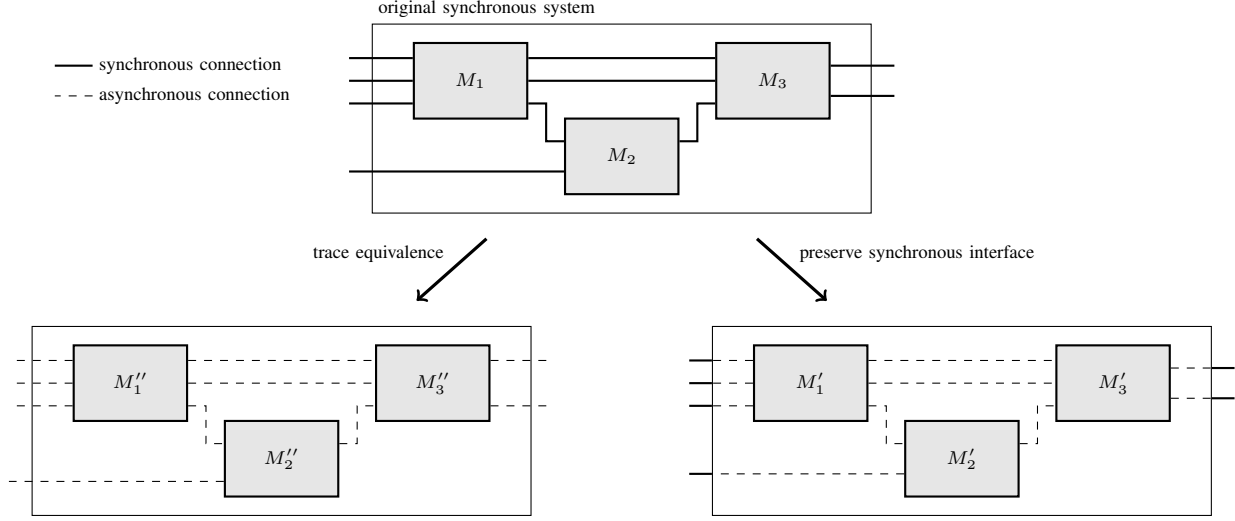


Figure 2. Desynchronizing a Synchronous System

of specifications is not given in advance, this implies that the desynchronized system has to produce for all outputs \hat{y} in each step the same values as the original system for the corresponding output y :

$$\langle y^0, y^1, y^2, \dots \rangle = \langle \hat{y}^0, \hat{y}^1, \hat{y}^2, \dots \rangle$$

The rest of the paper presents our desynchronization approach. First, we highlight how the system can be (re)partitioned, i.e. how the components C'_1, \dots, C'_m can be derived from a synchronous system description S . In Section 5, we then focus on the elimination of unnecessary communication and computation between the components.

4. Splitting the System into Components

A synchronous system is usually organized as a set of concurrently running modules. This partitioning is given by the modules in the source code - thus, it is the programmer's view to the system. Since modules of a well partitioned system encapsulate distinguished aspects, the structure is usually quite well suited for a desynchronization. However, better results are generally achieved by an automatic analysis of the data dependencies, which can be directly derived from the guarded actions. For the illustration of these dependencies, we use the Action Dependency Graphs.

Definition 2 (Action Dependency Graph): An Action Dependency Graph (ADG) is a bipartite graph which is given by a tuple $G = (V, A, E)$ consisting of

- a set $V = V_I \cup V_L \cup V_O$ of vertices representing input variables V_I , local variables V_L and output variables V_O ,
- the set of vertices A representing the guarded actions,
- and the set of edges E representing the dependencies between the actions and the variables. It is composed

of the two disjoint sets $E_I \subseteq (V_I \cup V_L) \times A$ and $E_O \subseteq A \times (V_O \cup V_L)$, which represent the read and write dependencies of the actions.

For a given synchronous system, the ADG contains a vertex for each variable v and for each action \mathcal{A} . Furthermore, it contains the following edges

- $((\gamma, \mathcal{A}), x) \in E_O$ iff x is written by action \mathcal{A} , i.e. iff $\mathcal{A}(\gamma, x = \tau)$ or $(\gamma, \text{next}(x) = \tau)$ (write dependency: **writes**(\mathcal{A}, x))
- $(x, (\gamma, \mathcal{A})) \in E_I$ iff x occurs in the guard γ or in the right-hand side of action \mathcal{A} (read dependency: **reads**(\mathcal{A}, x))

The graph encodes the restrictions for the execution of the guarded actions of a synchronous system. An action can be only executed if all variables occurring in its guard and the right-hand side of the assignment (i.e. its read variables) are known. Similarly, a variable is only known if all actions writing it have been evaluated before. Figure 3 shows an example. A compiler extracts four guarded actions for the data flow from the Quartz program Parallel (left-hand side of the figure): $\ell_1 \Rightarrow x_1 = \text{true}$, $\ell_1 \Rightarrow x_2 = \text{true}$, $\ell_1 \wedge x_1 \Rightarrow y_1 = \text{true}$, $\ell_1 \wedge x_2 \Rightarrow y_2 = \text{true}$. If the program is currently at location ℓ_1 , the local variables x_1, x_2 are made present, and depending on them the outputs y_1, y_2 are made present. The control flow is very simple in this example since the program terminates after this step.

The corresponding ADG is shown in the right-hand side of Figure 3. It reveals that the actions for x_1 and y_1 and the actions for x_2 and y_2 must be executed sequentially, while both groups can be executed in parallel. Hence, *parallelism in the program (given by the operator \parallel) is, in general, independent of the parallelism given by the dependencies*, and Action Dependency Graphs can be used to obtain a better partitioning of the system, since they are independent

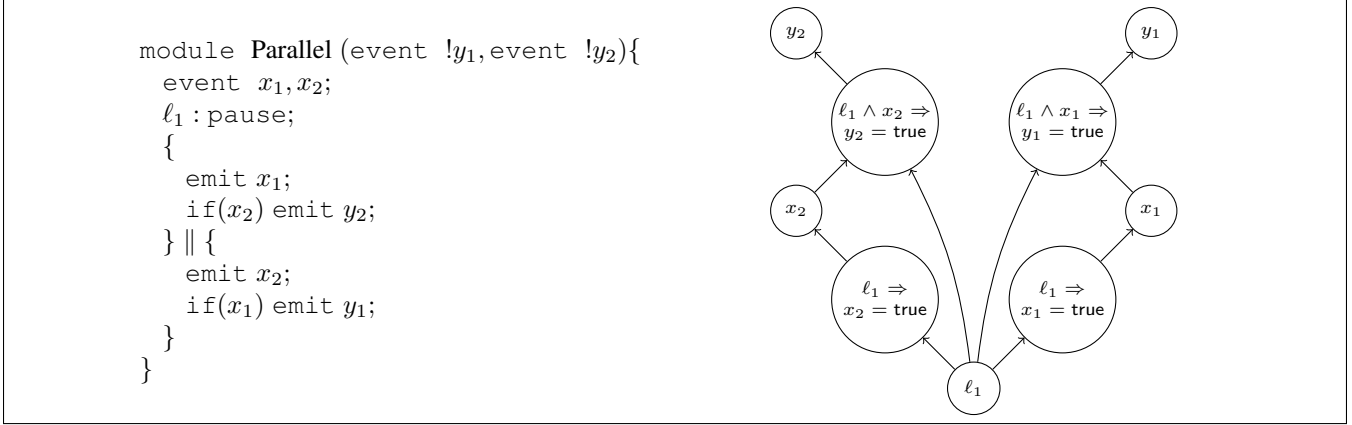


Figure 3. Quartz Program Parallel and its Action Dependency Graph

of the original program structure.

An apparently simple case is obtained if the graph consists of several unconnected components. These parts of the reaction can be executed in parallel without any restrictions. Since they do not have any dependencies, the corresponding parts of the graph do not need to be synchronized, and thus, each one can run independently of the other ones. Only the size of the buffers limits the divergence of the individual parts. Note again that the concurrent parts do not need to stem from different threads of the original synchronous program.

Unfortunately, many programs do not have this simple structure: Usually, there are data dependencies between the different parts of the system. Nevertheless, a static partitioning can always be obtained by computing each output value separately according to its dependencies: For each output y , a corresponding sub-ADG, the cone of influence of y , denoted by $\text{cone}_G(y) = (V_y, A_y, E_y)$ is retrieved from the ADG according to the following rules:

- $v \in V_y \Leftrightarrow (v = y) \vee (\exists \mathcal{A} \in A_y. \text{reads}(\mathcal{A}, v))$
- $\mathcal{A} \in A_y \Leftrightarrow (\exists x \in V_y. \text{writes}(\mathcal{A}, x))$
- $(x, \mathcal{A}) \in E_y \Leftrightarrow (x \in V_y) \wedge (\mathcal{A} \in A_y)$
- $(\mathcal{A}, x) \in E_y \Leftrightarrow (x \in V_y) \wedge (\mathcal{A} \in A_y)$

All these subgraphs are then extracted as independent ADGs by copying and renaming the nodes that are contained in more than sub-ADG.

Figure 4 shows a simple example. The synchronous source program CopyTwo_A takes two inputs and directly copies the inputs to the outputs. The right-hand side of the figure shows the ADG for this module. Both actions depend on the control-flow, namely the label ℓ_1 , which is repeatedly visited in the program. CopyTwo_B can be derived from CopyTwo_A by duplicating the control flow location ℓ_1 (and renaming the copy to ℓ_2), as the lower ADG shows. The program CopyTwo_B shows how this transformation would look like in the source code: The loop is split into two parallel ones, and only the inputs are still shared.

Alternatively, the shared part of the subgraph can be treated as a separate component, which is indicated by the dashed lines in the upper ADG of the figure. Since the construction ensures that there are no cyclic dependencies between the partitions of the system¹, the asynchronous execution can be realized by inserting a FIFO channel at the cuts of the graph. The shared part of the original graph now runs independently of the remaining two components, and its skew is again only limited by the buffer size.

Generally, partitioning the system into components creates a cut \mathcal{C} , which is given by a partial function that maps variables to pairs of components. If the graph is cut at variable v , the pairs of components (C_S, C_R) exchanging information over v is set as the function value of v , where S is the writing component and R the receiving one - thus, $\mathcal{C}(v) = (C_S, C_R)$. For each internal variable v , the cut function is undefined: $\mathcal{C}(v) = \uparrow$.

In the partitioned system, there is no longer a global clock. Instead, the components only synchronize by the presence of data values: As it is still guaranteed that each input port gets a value for each step, the component just has to wait for the arrival of all inputs, before it computes the outputs and transmits them to the other components. Although, the same communication and the same computation as in the fully synchronous execution takes place, this synchronization scheme generally yields better reaction times, i.e. the outputs are computed faster than in the fully synchronous reference implementation. The reason is simple: long reactions in one component do no longer delay the other reactions.

5. Controlling Components by Modes

The decomposition into several asynchronously running components, as presented in the previous section, is only

1. Cyclic (but causally correct) subparts of the system cannot be split, since the individual parts need an instantaneous communication link between each other to compute the reaction.

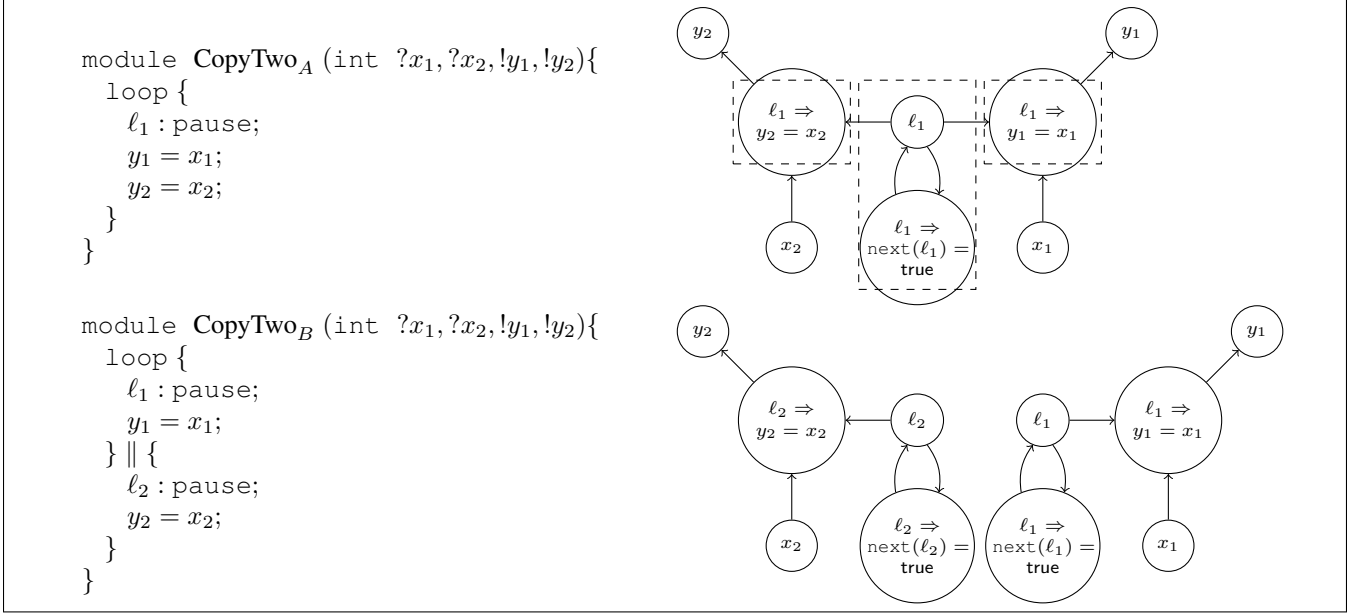


Figure 4. Quartz Program CopyTwo and its Action Dependency Graphs

the first step of desynchronization. It removes the need of a global clock and allows a skew between the components only bounded by the connecting buffers. However, another problem remains: although many synchronous systems do not use all variables in all steps, a fully synchronous execution (as well as the execution scheme presented so far) still enforces a complete computation of all the variables in the system. This causes a potentially immense overhead, if the system contains components that compute and transmit values at different rates or that are only sporadically active. In this case, the physical communication medium (especially if it is a shared by several logical channels) if flooded with irrelevant information. Even worse, the components that produce these unnecessary information need time and power to produce this useless information. In many embedded applications, this waste of resources cannot be afforded.

This section tackles this problem and proposes an approach to eliminate the unnecessary communication and computation. To this end, we introduce so-called *modes* for the components of the system. A mode basically defines which inputs are read and which outputs are computed and transmitted. Clearly, any other values must not be sent. Otherwise, the unexpected values destroy the timing of the components. With the help of this additional information, synchronization can be reconstructed by the components of the system - the only reason for the transmission of the irrelevant values. As modes are transmitted in addition to the actual data, the interface of each component is extended by an additional input that is used to set its mode.

5.1. Mode Variants

In principle, mode information could be added as status bits to all signals of the system.² From the theoretical side, this involves the same effort. However, most practical systems only have a small number of different configurations, i. e. how values are sent through the system. Hence, encoding these configurations in terms of modes is usually more efficient, while the variant with the additional bit appears as the worst case of modes.

Since the required inputs can be derived from a given set of requested outputs, and vice versa, there are two fundamentally different approaches to the introduction of modes: data modes and demand modes.

Data modes specify which variables are currently present and carry relevant information. All variables depending on irrelevant variables are marked as irrelevant, and the information is propagated through the network. This is the straightforward approach which corresponds to the solution for synchronous systems and explicitly sending absence values. Data modes are very easy to implement, and they can avoid unnecessary computation. However, they have certain drawbacks: They do neither reduce communication and nor decouple the components. Additionally, the designer has to determine on the sender side which values are needed on the receivers side. If the choice is correct, the resulting system is a combination of endochronous isochronously interacting components.

Demand modes tell a component (called the sender)

2. This shows the strong links to asynchronous circuit design, which has been already pointed out in [7].

which variables should be computed. Only variables which are needed for the computation of the current outputs are requested by the receiver. If there are several receiving components, then all of them determine the mode of the sender. All other values are not transmitted by the sender (and are thus not computed). The benefit of this approach is apparent: both communication and computation overheads are reduced. Components which do not contribute to the outputs of the current step can be deactivated or already produce results for subsequent macro-steps. In systems with demand modes, endochrony and isochrony can be guaranteed by construction. Though, a drawback exists: The information whether the value is needed is transmitted in the opposite direction to the value - the information flow has the opposite direction (hence the name *counterflow* appears in several related approaches). This again does not decouple the sender from the receiver, since the resulting cycle is instantaneous, i.e. it consists of value which are all related to the same macro-step. This counters the desynchronization efforts, since information should only flow in one direction to decouple the components.

An alternative implementation of the demand modes can solve this problem: Instead of sending modes from the component that request a value for a variable (backward, in opposite direction to the data flow), the mode flow can be also oriented in forward direction. This can be accomplished by moving or copying the parts of a component that request a value in front of the one that produces it. In the following, we consider this variant of *forward demand modes*. Note that also in this case, a mode is *not* a clock signal: if the component does not need to compute a value for the current step (i.e. its mode $m = \{\}$ requests the empty set of variables), the mode is not sent at all. Hence, components that do not contribute to the final result can be temporarily deactivated, or they can be used to already compute values for subsequent steps (due to the decoupling of Section 4).

5.2. Determining and Checking Modes

The computation of all mode information can be encapsulated in a separate component, a mode manager. It has access to all input and internal variables. With the help of this information, it determines which internal values must be computed in the current step and sends this information to all the components. In principle, determining the modes can be as complicated as computing the variables, since the mode manager is more or less a combination of all the guards in the system.

The overhead can be reduced by several structural changes: the mode manager does not need to communicate with all components directly. It can use other components to forward this mode, where any fixed route is possible. Since the mode is attached to the current front of values, it is always clear which step it refers to. Furthermore, the

mode manager can be split and be distributed across the system. Instead of simply forwarding the mode information, small mode managers are attached to individual components. These managers need the inputs and the state of all component receiving values from the administered one. Between the central and distributed realization, arbitrary intermediate forms are imaginable: the best choice depends on the actual structure of the asynchronous system - a mode manager per physical unit seems a reasonable choice.

Before we can explain how the modes are checked or determined, we need to define how desynchronized components read their inputs and write their outputs, since the modes of component must comply with its given behavior. Hence, the following communication protocol is only one of several alternatives, which can be adapted by the developer³, provided that the modes are changed accordingly. The overall protocol must be endochronous, i.e. the presence and variables can be inferred by the values of variables, which are known to be present.

We propose the following communication protocol of a desynchronized component. In each reaction, the component C performs the following steps:

- First, the component C reads its mode, which gives the set of requested variables $m = \{y_1, \dots, y_n\}$.
- The subgraph $G_m = \bigcup_{i=1 \dots n} \text{cone}_C(y_i)$ (see Section 4) is computed. It contains all actions that write the requested variables y_1, \dots, y_n or dependent ones.
- All the variables occurring in the guards $\bigcup_{(\gamma, \mathcal{A}) \in G_m} \text{vars}(\gamma)$ are read by the component.
- For all the actions whose guard is evaluated to true, the variables occurring in the right-hand side $\bigcup_{(\gamma, x=\tau) \in G_m} \text{vars}(\tau)$ are read.

It should be clear that setting consistent modes is a nontrivial task. If they are set by the designer, there must be some tool support that automatically verifies that the given *mode-ification* of a system is well-formed, i.e. whether exactly the requested set of outputs is computed by the system. Fortunately, checking this property is very simple. For all output variables of the component y , we introduce two additional status bits $\text{req}(y)$ and $\text{prov}(y)$. Then, the mode computation of the modes by the manager is encoded by setting the request bits $\text{req}(y_i)$ for $i = 1, \dots, n$ of $m = \{y_1, \dots, y_n\}$. If a component sets a value for y_i , i.e. if the guard of an action writing y_i is true, we set $\text{prov}(y)$. A well-formed *mode-ification* of the system is then given by the temporal logic formula $G \bigwedge_{i=1, \dots, n} \text{req}(y_i) = \text{prov}(y_i)$ and can be checked by any state-of-the-art model checker.

Alternatively, the modes for the components can be automatically determined by a tool. Thereby, a very simple synthesis approach is feasible in many cases: For the outputs

3. For example, the given behavior does not use the non-strictness of many operations (conjunction, disjunction, multiplication etc.). Hence, components may request inputs that do not contribute to the final result. Nevertheless, for the sake of clarity, we only consider a syntactical analysis.

of all components y we determine the condition $\text{req}(y)$ (i.e. when the output y contributes to the outputs of the system) by traversing the ADG of the system in opposite direction to the data flow. We start by setting $\text{req}(y) = \text{true}$ for each output y of the system. Then, we move to the components that compute these outputs. For each output y of the system, we determine the set of variables in its cone of dependence $\text{cone}(y)$ and update their request condition as follows: All actions are determined in which this variable x appears on the right hand-side. The disjunction of their guards is then added to the request condition $\text{req}(x)$. This is repeated until all components and all variables have been handled.

The previous algorithm does not scale well if there are many internal state variables in a component, since all the outputs generally depend on the whole state, which requires that it is computed in each step. In many real-world examples, this is not needed. A variant of liveness analysis as known from classical compiler construction is needed to check whether the internal state variables have an effect on the later outputs. At this point, knowledge about the application and the environment of the system needs to be given by the user in order to determine correct and efficient modes.

5.3. Example

Figure 5 gives an example. The Quartz program on the left-hand side has the following behavior: Initially, two subthreads are created. The first one consists of an `every` statement. Its body is started as soon as the Boolean variable c becomes true. The execution of the body is restarted (and aborted) when c is set again. The body first initializes a local variable s and then executes the sequences which follows the `during`. In each step the control flow is inside the sequence (including the initial one) the assignment to the output d is executed. The second subthread of the Quartz program is just an infinite loop, which continuously writes the square root of a to the output e .

The analysis of the dependencies between actions and variables may propose a decomposition of the program into three components. Component C_1 computes the square root of a , the second one C_2 the square of b , and the third one C_3 is responsible for the computation of the remaining part of the program (see Figure 5). Apparently, the first two components are combinational and only have a single output. Thus, both have two natural modes: either to compute the output or to do nothing. Therefore, the component determining the square of b exactly has these two modes. In contrast, the square root component only has one mode since its output is needed in all steps. Due to the local variable s and the control flow labels, the third component is sequential. All its modes include all outputs, since they correspond to the outputs of the overall system. The modes only differ in the

local variables that are computed. For example, s does not need to be computed after leaving the `during` statement, since its value is not required for any following value of d : before reaching the assignment $d = s$, s is reinitialized again.

Hence, a possible decomposition can be the one shown in Figure 5. In addition to the three components which have been already described above, there is a mode manager, which contains a fragment of the control flow (labels ℓ_1 to ℓ_4) of the original system that is responsible for the interaction of the modules.

6. Conclusions

This paper presents a new approach for the desynchronization of a synchronous system while maintaining its observable behavior at its interface. Our desynchronization approach is based on two aspects: First, dependency graphs reveal how the synchronous system can be distributed on an asynchronous architecture. Modes are used to decouple the obtained parts of the system by avoiding unnecessary communication and computation so that resources are efficiently used. We presented data and demand modes, and believe that forward demand modes are the most practicable ones. We sketched how manually obtained modes can be verified as well as automatically generated by the synthesis tool. While this paper introduced the general concept of modes, future work will consist of (1) evaluation of these concepts by real-world examples and (2) developing algorithms for mode computations given a system description and a concrete implementation platform.

References

- [1] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163:125–171, 2000.
- [2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
- [3] G. Berry. Synchronous languages for hardware and software reactive systems. In C. Delgado Kloos and E. Cerny, editors, *Conference on Computer Hardware Description Languages and Their Applications (CHDL)*, Toledo, Spain, 1997. Chapman & Hall.
- [4] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.
- [5] G. Berry. The constructive semantics of pure Esterel. <http://www-sop.inria.fr/esterel.org/>, July 1999.
- [6] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison Wesley, Austin, Texas, May 1989.
- [7] S. Dasgupta, D. Potop-Butucaru, B. Caillaud, and A. Yakovlev. Moving from weakly endochronous systems to delay-insensitive circuits. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 146(2):81–103, 2006.

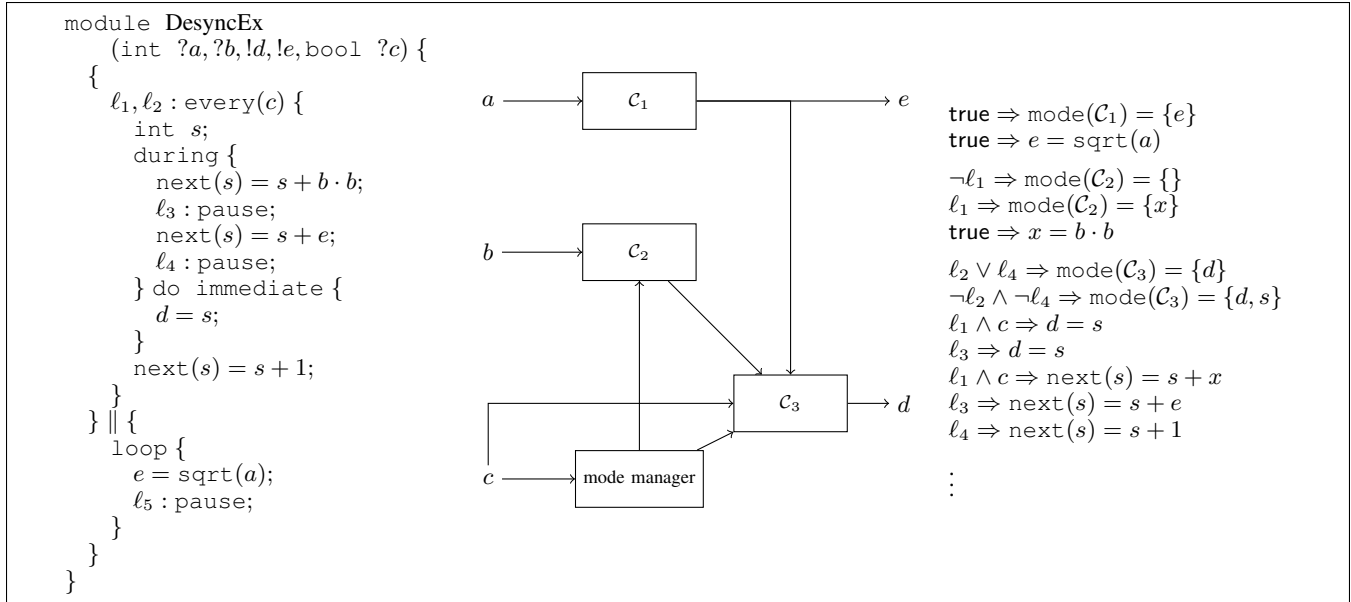


Figure 5. Quartz Program DesyncEx and its Desynchronization

- [8] D.L. Dill. The Murphi verification system. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 390–393, New Brunswick, NJ, USA, 1996. Springer.
- [9] H. Järvinen and R. Kurki-Suonio. The DisCo language and temporal logic of actions. Technical Report 11, Tampere University of Technology, Software Systems Laboratory, 1990.
- [10] L. Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Cooperation, 1991.
- [11] L. Lavagno and E. Sentovich. ECL: A specification environment for system-level design. In *Design Automation Conference (DAC)*, pages 511–516, New Orleans, Louisiana, USA, 1999. ACM.
- [12] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 48–57. IEEE Computer Society, 2005.
- [13] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 67–76. IEEE Computer Society, 2004.
- [14] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design (FMSD)*, 2006.
- [15] K. Schneider. A verified hardware synthesis for Esterel. In F.J. Rammig, editor, *Workshop on Distributed and Parallel Embedded Systems (DIPES)*, pages 205–214, Schloß Ehringerfeld, Germany, 2000. Kluwer.
- [16] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 143–156, Newcastle upon Tyne, UK, 2001. IEEE Computer Society.
- [17] K. Schneider. Proving the equivalence of microstep and macrostep semantics. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logic (TPHOL)*, volume 2410 of *LNCS*, pages 314–331, Hampton, VA, USA, 2002. Springer.
- [18] K. Schneider. The synchronous programming language Quartz. Internal Report (to appear), Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.
- [19] K. Schneider and J. Brandt. Performing causality analysis by bounded model checking. In *Conference on Application of Concurrency to System Design (ACSD)*, page 78, Xi’an, China, 2008. IEEE Computer Society.
- [20] K. Schneider, J. Brandt, and T. Schuele. Causality analysis of synchronous programs with delayed actions. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 179–189, Washington, DC, USA, 2004. ACM.
- [21] K. Schneider, J. Brandt, and T. Schuele. A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(4):71–97, 2006.
- [22] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In *Application of Concurrency to System Design (ACSD)*, pages 106–115, St. Malo, France, 2005. IEEE Computer Society.
- [23] T.R. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1996.
- [24] O. Tardieu and R. de Simone. Curing schizophrenia by program rewriting in Esterel. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 39–48, San Diego, California, USA, 2004. IEEE Computer Society.