Dependency-Driven Distribution of Synchronous Programs

Daniel Baudisch, Jens Brandt, Klaus Schneider

Dependency-Driven Distribution of Synchronous Programs

Daniel Baudisch, Jens Brandt, and Klaus Schneider

Embedded Systems Group
Department of Computer Science
University of Kaiserslautern
http://es.cs.uni-kl.de

Abstract In this paper, we describe an automatic synthesis procedure that distributes synchronous programs on a set of desynchronized processing elements. Our distribution procedure consists of three steps: First, we translate the given synchronous program to synchronous guarded actions. Second, we analyze their data dependencies and represent them in a so-called action dependency graph (ADG). Third, the ADG is subsequently partitioned into of sub-graphs where cuts can be made horizontal (for a pipelined execution) or vertical (for a concurrent execution). Finally, we generate for each sub-graph a corresponding component and automatically synthesize a communication infrastructure between these components.

1 Introduction

Synchronous programming languages like Esterel [5], Lustre [22] or Quartz [30] are all based on the synchronous model of computation [3]. Its core is the synchronous hypothesis, which divides the program execution into *micro* and *macro steps*. Thereby, micro steps, which represent computation and communication, are all executed in zero time. Consumption of time is explicitly modeled by grouping a finite number of micro steps to macro steps, which all consume the same amount of logical time. As a consequence, all threads of the program run in lockstep, i. e. they automatically synchronize at the end of each macro step. Since all micro steps of a macro step are executed at the same point of time (at least from the semantical point of view), their ordering within the macro step is irrelevant. Therefore, values of variables are determined with respect to macro steps instead of micro steps, i. e. variables do not change within a macro-step.

This abstraction guarantees many properties which are desirable for the development of safety-critical embedded systems. It enforces *deterministic concurrency*, which has many advantages in system design, e.g. to avoid Heisenbugs (i.e. bugs that disappear when one tries to simulate/test them), and it is the key to a straightforward translation of synchronous programs to hardware circuits [4,27,30]. Furthermore, the concise formal semantics of synchronous languages makes them particularly attractive for reasoning about program properties [33,28], correctness and worst-case execution time [26,7].

However, the other side of the coin is that the synchronous model of computation makes both the compilation and synthesis quite difficult. While causality [6,36] and

schizophrenia [33] problems already challenge compilers, the synthesis procedures often have to map a synchronous program to a target architecture that does not provide perfect synchrony. This mismatch between the synchronous model used for the development of a system and most real-world implementation environments poses serious problems, in particular for distributed and parallel embedded applications (as in the automotive or avionic industries), where the target architecture is a heterogeneous set of interconnected processing elements.

Using these architectures, it is often not reasonable to maintain a global clock, which synchronizes all components. In addition to communication latencies, which would slow down the execution, the varying speeds of the individual components would lead to unnecessary idle times. As the slowest component in each step defines the global speed, the resulting performance would often become unacceptable.

In general, there are many ways to partition and distribute a synchronous program into single components. The simplest approach requires that the structure of the system description corresponds to the one of the final target architecture. However, this very simple approach has several drawbacks: first, it is not in the spirit of model-based design, where the system description should be independent of the target system as long as possible. Second, it allows one only to partition the set of modules used in the system description into components, and therefore, it does not allow one to split a single module into different components. Finally, the communication among the sub-systems that correspond to the identified components has to be adapted since there is no longer a global clock.

The contribution of this paper is therefore twofold: First, it presents a partitioning of synchronous programs into concurrent, desynchronized parts. Second, it provides an automatic synthesis of a generic communication infrastructure between these components, which ensures that the implementation still complies with the synchronous semantics of the original source program.

Thereby, it integrates and extends our previous approaches: [1] extracts independent parts of a synchronous program to extract concurrent threads, whereas [2] slices chains of dependencies to create a pipelined system. In this paper, we integrate both partitioning approaches so that an arbitrary combination of concurrent and pipelined execution becomes possible. Furthermore, we do not rely on a specific synthesis target: the partitioning and the communication infrastructure are constructed in a target-independent intermediate format so that each component can be later mapped to hardware or software, as well as the communication between them can be mapped to appropriate protocols.

There is some previous work which has already considered the automatic distribution of synchronous programs to an asynchronous network of processing elements: In [20,14], a clock-driven distribution of Lustre programs is presented which partitions and distributes the system according to the clock that triggers each part. While this approach has shown to produce quite efficient implementations, it may suffer from a significant drawback: Mutual data dependencies between components may require that some component must be further decomposed into smaller components, which may require in turn additional communication and synchronization effort. In our approach, this is avoided by construction.

Related work appeared also in the implementation of digital circuits where the number of cycles required to transmit a signal from one component to another can only be done when the final layout has been derived. To this end, latency-insensitive [10,12,11] and synchronous elastic systems [18,17,24] have been proposed to make the communication between the synchronous modules independent of a global clock. We also make use of these ideas for distributing a given synchronous system description into desynchronized components.

The rest of the paper is organized as follows: Section 2 briefly introduces synchronous guarded actions, which serve as a starting point for the synthesis procedure of this paper. Section 3 explains how we analyze the data-dependencies of the guarded actions by means of an action-dependency graph (ADG), which gives rise to a partition of the guarded actions. Section 4 explains the construction of the communication infrastructure. Finally, Section 5 concludes with a short summary.

2 Synchronous Guarded Actions

Synchronous systems [3,21] as implemented by synchronous languages like Esterel [5] and Quartz [30,33,29] divide their computation into single reactions. Within each reaction, new inputs are synchronously read from all input ports, and new outputs are synchronously generated on all output ports with respect to the current state of the system and the current inputs. Furthermore, the reaction determines the state for the next reaction. It is very important for synchronous languages that *variables do not change during the macro step*. For this reason, all micro steps are viewed to be executed at the same point of time (as they are executed in the same variable environment). The instantaneous feedback due to immediate assignments to outputs can therefore lead to so-called causality problems [6,34,35]. Compilers check the causality of a program at compile time with a fixpoint analysis that essentially corresponds to those used for checking the speed-independence of asynchronous circuits via ternary simulation [9]. Besides the causality analysis, compilers for synchronous languages often perform further checks to avoid runtime exceptions like out-of-bound overflows or division by zero. Moreover, most compilers for synchronous languages also allow the use of formal verification, usually by means of model checking.

The compiler of our Averest system[1] is split into several compile phases: The front-end translates a synchronous program into an equivalent set of *(synchronous) guarded actions* [16,19,23,25] of the form $\langle \gamma \Rightarrow A \rangle$ (see [33,8,30]). 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 this paper, these are the assignments of the source language, i. e. the guarded actions have either the form $\langle \gamma \Rightarrow x = \tau \rangle$ (for an immediate assignment) or $\langle \gamma \Rightarrow \mathit{next}(x) = \tau \rangle$ (for a delayed assignment). In each macro step, the guards γ of all actions (of all variables) are checked simultaneously. If a guard γ is true, the right-hand side τ of the action is immediately evaluated. Immediate actions $x = \tau$ assign the computed value immediately to the variable x, while the updates of delayed actions $\mathit{next}(x) = \tau$ are deferred to the following macro step. If no action sets

[1] http://www.averest.org

the value of a variable in the current step, it is determined by the so-called *reaction to absence*, which usually keeps the value of the previous step. In general, a different behavior (like resetting to a default value) is possible, but for the sake of simplicity, we do not elaborate these cases in the following.

Hence, if an immediate assignment $x = \tau$ is enabled in the current macro step, the current value of x must be equal to the value of τ. Implementations must therefore make sure that x is not read before the value of τ is evaluated so that one implements the programmer's view that the assignment was performed in zero time.

Synchronous systems are always deterministic, because there is no choice among activated guarded actions, since *all of the enabled* actions must be fired. Hence, any system is guaranteed to produce the same outputs for the same inputs. However, forcing conflicting actions to fire simultaneously may lead to causality problems. This is a well-studied problem for synchronous systems and many analysis procedures have been developed to spot and eliminate these problems [32,35,31,36]. In the following section, we assume that a program is causally correct and that for each variable at most one action is active in a macro step.

$$\left[\begin{array}{ll} a = x + y & b = x - y \\ c = z \cdot z & r \Rightarrow x = p \\ s \Rightarrow \mathit{next}(x) = a & s \Rightarrow y = q \\ \neg s \Rightarrow y = o & \mathit{next}(r) = s \\ \mathit{next}(o) = a \cdot b & m = b + c \end{array}\right]$$

Figure 1. Synchronous Guarded Actions

Figure 1 shows a set of synchronous guarded actions, which will serve as a running example in the following. Note that the translation of synchronous programs into guarded actions is already the first step towards our distribution, since it allows us to split the system into subsets of guarded actions that will form the distributed components.

3 Partitioning System Descriptions into Components

As we already mentioned in the previous section, synchronous guarded actions must be executed according to their causal data dependencies. As we want to map the actions onto a network of asynchronous processing elements, the partition must also reflect the causal order. Before we explain our approach, we first need to give some basic definitions about the dependencies between actions and the variables accessed by them.

Definition 1 (Read and Write Dependencies). *Let* $\mathsf{FV}(\tau)$ *denote the free variables occurring in the expression* τ*. Then, the dependencies from actions to variables are defined as follows:*

$$\begin{aligned} &\mathsf{rdVars}\,(\gamma \Rightarrow x = \tau) := \mathsf{FV}(\tau) \cup \mathsf{FV}(\gamma) \\ &\mathsf{rdVars}\,(\gamma \Rightarrow \mathit{next}(x) = \tau) := \mathsf{FV}(\tau) \cup \mathsf{FV}(\gamma) \\ &\mathsf{wrVars}\,(\gamma \Rightarrow x = \tau) := \{x\} \\ &\mathsf{wrVars}\,(\gamma \Rightarrow \mathit{next}(x) = \tau) := \{\mathit{next}(x)\} \end{aligned}$$

The dependencies from variables to actions are determined as follows:

$$\mathsf{rdActs}\,(x) := \{\gamma \Rightarrow A \mid x \in \mathsf{rdVars}\,(\gamma \Rightarrow A)\}$$
$$\mathsf{wrActs}\,(x) := \{\gamma \Rightarrow A \mid x \in \mathsf{wrVars}\,(\gamma \Rightarrow A)\}$$

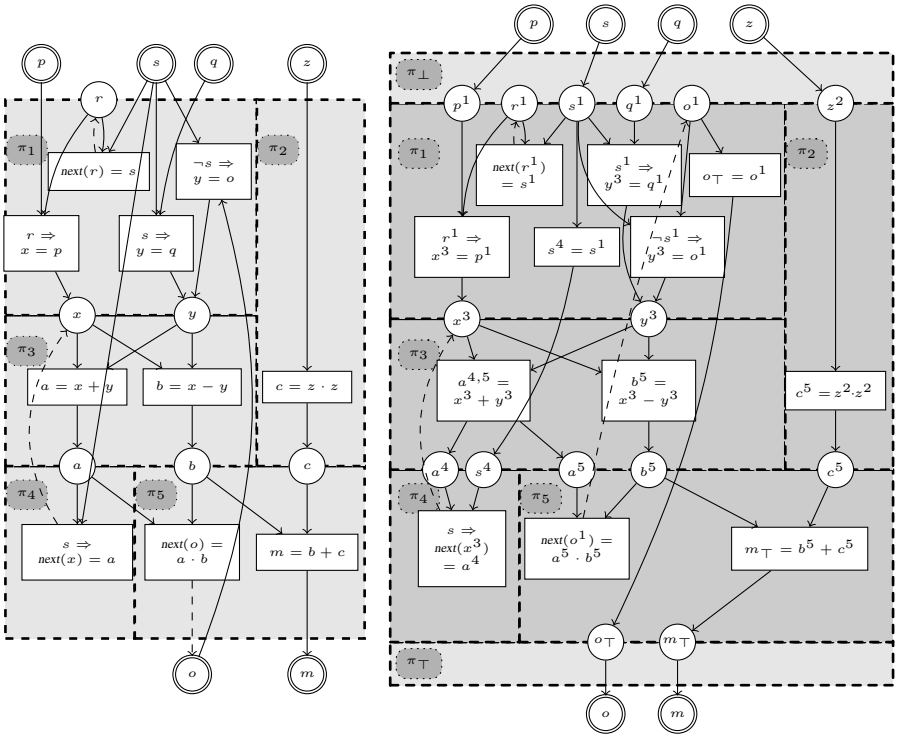

Figure 2. Left: Partitioned ADG, Right: Partioned ADG with Intermediate Variables

For a given set of guarded actions, the dependencies between all individual elements can be illustrated by an *Action Dependency Graph (ADG)*, which is a bipartite graph consisting of vertices $\mathcal{V}$ representing variables, vertices $\mathcal{A}$ representing the guarded actions and labeled edges representing the dependencies. Thereby, a solid (or dashed) edge from $\langle \gamma \Rightarrow A \rangle$ to x denotes that action A writes x in the current step (or next step). Similarly, a solid edge from x to $\langle \gamma \Rightarrow A \rangle$ expresses that x is read in A, i. e. it appears in the guard γ or in the right-hand side of action A. Thus, this graph exactly encodes the restrictions for the execution of the guarded actions of a synchronous system. An action can be only executed if all read variables are known. Similarly, a variable is only known if all actions writing it in the current step have been evaluated before.

The dependencies encoded in the ADG give rise to possible distributions of the original synchronous system. In the following, we do not focus on the question how to find an optimal solution for a given realization (e. g. software threads or hardware circuits) or target platform. Naturally, the concrete partition generally has a significant impact on the performance of the implementation. However, the appropriate data can be only provided by an external analysis tool, which knows many internals about the

target platform. We do not focus on that but provide a general method how the results of such an analysis can be exploited for desynchronization. Our approach is generally applicable and it only requires a legal partition, which is defined in the following.

Definition 2 (Legal Partition of an ADG). *A* partition Π *of an ADG is a mapping from actions to classes* $\pi \in \Pi$*. Let* $\mathsf{class}(A)$ *denote the class of an action* $A \in \mathcal{A}$*, and let* $\mathsf{gacts}(\pi)$ *denote all the actions occurring in class* π*. Let* $\sqsubseteq$ *be the reflexive and transitive closure of the following relation* $R \subseteq \mathcal{A} \times \mathcal{A}$*:* $(A_1, A_2) \in R \Leftrightarrow \mathsf{wrVars}(A_1) \cap \mathsf{rdVars}(A_2) \neq \{\}$*. A partition is* legal *iff* $\sqsubseteq$ *is a partial order.*

Note that, according to Definition 1, the intersection of $\mathsf{wrVars}(A_1)$ and $\mathsf{rdVars}(A_2)$ is empty if A_1 is a delayed action for a variable read in A_2. For example, the left hand side of Figure 2 shows the ADG of the actions of Figure 1. It is partitioned into five classes, which can be easily verified to be legal, since they form a partial order. This ensures that the partitioned implementation will be free of deadlocks.

Since all classes of the partition should run in a desynchronized way, they must be able to process data of different macro steps. This data is stored in communication channels between the classes in the later realization, which are modeled by additional variables in our model. Therefore, for each variable x of the original system, we declare several intermediate variables, one for each class that reads x, or formally:

Definition 3 (Read Access and Activity). *For all classes* $\pi \in \Pi$ *and for all variables* $x \in \mathcal{V}$*, the predicate* $\mathsf{read}(x, \pi)$ *denotes whether* x is read in class π*, i. e.* $\mathsf{read}(x, \pi) = \exists G.\ G \in \mathsf{gacts}(\pi) \wedge x \in \mathsf{rdVars}(G)$*. Additionally, we consider two virtual classes* $\pi_\bot \sqsubset \pi \sqsubset \pi_\top$ *for all* $\pi \in \Pi$*, and we assume* $\mathsf{read}(i, \pi_\bot)$ *and* $\mathsf{read}(o, \pi_\top)$ *for each input variable* i *and output variable* o*, respectively.*

If $\mathsf{read}(x, \pi)$ holds, an intermediate variable for x is inserted in class π, which provides the current input value of x. To distinguish all the different intermediate variables of x, we add a superscript π, where x^π represents the intermediate variable for x in class π. Among all the different copies of x, we select a set of *stable* incarnations. The stable incarnations mark the points in the partitioned system where a variable x must have become known, i. e. $\mathsf{stable}(x) = \{\pi \mid \mathsf{read}(x, \pi) \wedge \nexists \rho \sqsubset \pi.\mathsf{read}(x, \pi)\}$. Due to the concurrency of classes, there can be more than one incarnation. All write accesses to a variable will be forwarded to these stable incarnations leading to a high overhead in communication. Fortunately, each stable incarnation will get the same value for a defined macro step. Hence, communication overhead can be reduced by adding a canonical incarnation for x, written $\mathsf{canon}(x)$, that distributes the values for variable x to its stable incarnations.

Since all original variables have now been replaced by intermediate variables, the guarded actions must be rewritten to refer to them. Apparently, all actions (immediate and delayed) of the original system that are put in class π read variables with superscript π. Furthermore, all write accesses of a class π are forwarded to the canonical incarnation of this variable. We rewrite the actions as given in the function Transform shown in Figure 3. Thereby, let $[\gamma]^\pi$ be the operations that relabels all variables $x \in \mathsf{FV}(\gamma)$ with their superscripted counterparts x^π.

function Transform($\mathcal{G}$)
  $\mathcal{G}' := \{\}$
  forall $G \in \mathcal{G}$
    $\pi := \mathsf{class}(G)$
    case G :
    $\gamma \Rightarrow x = \tau$:
      $\mathcal{G}' := \mathcal{G}' \cup$
      $\langle [\gamma]^\pi \Rightarrow [x]^{\mathsf{canon}(x)} = [\tau]^\pi \rangle$
    $\gamma \Rightarrow \mathit{next}(x) = \tau$:
      $\mathcal{G}' := \mathcal{G}' \cup$
      $\langle [\gamma]^\pi \Rightarrow \mathit{next}([x]^{\mathsf{canon}(x)}) = [\tau]^\pi \rangle$
  return $\mathcal{G}'$

function CreateTransport($\mathcal{G}$)
  for $\pi = 1, \ldots, N$
    forall $x^\pi \in \mathcal{V}$
      $\mathcal{G} := \mathcal{G} \cup \{\mathsf{true} \Rightarrow x^\pi = \mathsf{pre}(x^\pi)\}$
  return $\mathcal{G}$

function CreateTransport′($\mathcal{G}$)
  for $\pi = 1, \ldots, N$
    $wr_\pi = \bigcup_{A \in \mathsf{gacts}(\pi)} \mathsf{wrVars}\,(A)$
    forall $x \in wr_\pi$
      $guard := (\mathsf{valid}_{\mathsf{in}}(\pi) \vee \mathsf{valid}(\pi))$
        $\wedge \mathsf{stop}_{\mathsf{in}}(\pi) \wedge \neg \mathsf{fire}(\pi)$
      $\mathcal{G} := \mathcal{G} \cup \{guard \Rightarrow \mathit{next}(x) = x\}$
  return $\mathcal{G}$

function DistributedSystem($\mathcal{G}$)
  $\mathcal{G} := \mathsf{Transform}(\mathcal{G})$
  $\mathcal{G} := \mathsf{CreateTransport}(\mathcal{G})$
  $\mathcal{G} := \mathsf{CreateTransport}'(\mathcal{G})$
  return $\mathcal{G}$

Figure 3. Functions to Distribute a Synchronous System

However, this causes a problem since all the classes generally process different macro steps, and each class can write to the same variable. Hence, values may not arrive in-order according to their logical time so that they have to be reordered explicitly. In our approach, this is accomplished by a *merge* component Merge_x, which provides an input for each class that may write to x. Such a component is attached to $\mathsf{canon}(x)$. In the current section, it does not play any role, since in a fully synchronous model, the *merge* component just implements the identity function. Its behavior is explained in detail in the next section.

Finally, we have to add the transport of the intermediate variables, which corresponds to the reaction to absence of a synchronous system: a class π that reads an intermediate variable x^π obtains its values from that class that precedes π and writes to x as soon as a set of variables is processed by this preceding class. The preceding class of class π is formally given by $\mathsf{classPre}(\mathsf{x}^\pi) = \max_\sqsubset\{j \mid \mathsf{read}(x^j, j) \wedge j \sqsubset \pi\}$ with $\max_\sqsubset(A) = \{\pi \,|\, \pi \in A \wedge \neg(\exists \nu \in A. : \pi \sqsubset \nu)\}$. Additionally, we define $\mathsf{pre}(x^\pi) = x^j$, $j \in \mathsf{classPre}(\mathsf{x}^\pi)$ as an arbitrary but determined predecessor of x^π. Due to concurrency of classes, a class may have more than one predecessor. Each predecessor generates for each set of inputs exactly one value for x. Furthermore, all incarnations of x are copies, i. e. they contain the same value for each input set with a defined logical time step. Hence, it is sufficient to forward the values for a variable x only from one preceding incarnation of x.

Note that incarnations following the stable ones do not require a Merge_x. Due to the Merge_x, the stable incarnation obtains the variable's values in the correct temporal ordering, and it will proceed sets of variables in-order. Hence, the stable incarnations will forward a variable's values in order and as a result of this, each succeeding incarnation obtains these values also in the correct temporal ordering. The right-hand side of Figure 2 shows the transformed set of guarded actions including the transport actions

for our running example. The ten original actions are rewritten so that they refer to the superscripted variables, and the remaining actions are due to the transfer of variables.

4 Communication Infrastructure

The previous section partitioned the system into a set of components, which are desynchronized in this section by introducing an appropriate communication infrastructure. Thereby, each class can be first synthesized separately and independently of the others. The individual classes are finally connected by channels that follow a generic desynchronizing protocol. We do not rely on a specific one but only require that it can model the validity of data values and the congestion of buffers (back-pressure). This is provided by latency insensitive protocols [12,15], synchronous elastic circuits [13] or almost any asynchronous communication infrastructure based on buffers.

In the following, we demonstrate how to apply the SELF protocol as described by Carmona et al. in [13] to the partitioned system to gain a synchronous elastic system. First, the classes require additional control logic for communication. The control logic guarantees the correct flow of information between the classes. The interface of each class $\pi \in \Pi$ is extended by two Boolean input signals. The input $\mathsf{valid}_{\mathsf{in}}(\pi)$ indicates that the current inputs of class π contain valid values, whereas the input $\mathsf{stop}_{\mathsf{in}}(\pi)$ tells the class whether its outputs can be processed by subsequent classes. Similarly, each class has two output signals, which drive the status signals of other classes: $\mathsf{valid}_{\mathsf{out}}(\pi)$ gives notice of the validity of the current outputs, while $\mathsf{stop}_{\mathsf{out}}(\pi)$ indicates whether the class is able to handle new inputs.

To control these flags, each class makes use of two additional variables $\mathsf{valid}(\pi)$ and $\mathsf{fire}(\pi)$, which memorize the validity of the current outputs and signalizes that class π can fire its actions, respectively:

1. If a class obtains valid inputs but currently has no valid outputs, it must read the inputs and fire its actions. Formally: $\mathsf{fire}(\pi) = \mathsf{valid}_{\mathsf{in}}(\pi) \wedge \neg\mathsf{valid}(\pi)$.
2. If a class obtains valid inputs or already has valid outputs, it has valid outputs in both cases. Formally: $\mathsf{valid}_{\mathsf{out}}(\pi) = (\mathsf{valid}_{\mathsf{in}}(\pi) \vee \mathsf{valid}(\pi))$.
3. If a class has valid outputs and a stop signal comes in, the internal output validity flag has to be set for the next step. Formally: $\mathit{next}(\mathsf{valid}(\pi)) = (\mathsf{valid}_{\mathsf{in}}(\pi) \vee \mathsf{valid}(\pi)) \wedge \mathsf{stop}_{\mathsf{in}}(\pi)$.
4. If a class obtains valid inputs but already has valid outputs or obtains a stop signal, then the class must set its own stop signal. Formally: $\mathsf{stop}_{\mathsf{out}}(\pi) = \mathsf{valid}_{\mathsf{in}}(\pi) \wedge (\mathsf{valid}(\pi) \vee \mathsf{stop}_{\mathsf{in}}(\pi))$.

All guarded actions are modified to take notice of the class's fire condition. The fire condition $\mathsf{fire}(\mathsf{class}(A))$ is added as an additional clause to the guards of all actions $A \in \mathcal{G}$ of the class's build as the conjunction of its old guard and the corresponding class's fire condition. Finally, if a class contains an action writing to a variable x and the class does not fire, but has to keep the value valid, it has to transport (copy) explicitly its value (see Function CreateTransport′ in Figure 3).

In a simple chain of classes $\pi_1, \pi_2, \ldots, \pi_N$ (as in a pipeline), the status signals can be simply connected between successive elements, i. e. $\mathsf{valid}_{\mathsf{in}}(\pi_{i+1}) = \mathsf{valid}_{\mathsf{out}}(\pi_i)$

and $\mathsf{stop}_{\mathsf{in}}(\pi) = \mathsf{stop}_{\mathsf{out}}(\pi_{i+1})$. For a general topology of the distribution, which is targeted in our approach, a more general solution is necessary. Each class that obtains its inputs from several others or sends its outputs to several others, needs to provide *join* or *fork* elements, respectively, as already explained in [13]. In the following, we explain the functioning of these elements in terms of our approach.

A join element is needed if a single class π obtains its inputs from several other classes $\pi_1, \ldots, \pi_n$. Obviously, it can only fire iff all inputs are valid. The *valid* flag of class π is set to $\mathsf{valid}_{\mathsf{in}}(\pi) = \bigwedge_{i=1,\ldots,n} \mathsf{valid}_{\mathsf{out}}(\pi_i)$. Values do not have to be stored internally in a join element since a valid input value is provided by the producing class until it is read, i. e. when all inputs are valid. The *stop* signals from class π to the preceding classes $\pi_1, \ldots, \pi_n$ are determined as follows: if class π stalls, the stall signal is simply broadcasted to all preceding classes. Furthermore, if some of these classes π_i already serves valid inputs but some other class π_j does not provide valid inputs yet, the class π_i must be also stalled, formally: $\mathsf{stop}_{\mathsf{out}}(\pi_i) = \mathsf{stop}_{\mathsf{in}}(\pi) \vee (\mathsf{valid}_{\mathsf{in}}(\pi_i) \wedge \bigvee_{j\in\{1,\ldots,n\}, i\neq j} \mathsf{valid}_{\mathsf{out}}(\pi_j))$.

A fork element is used if a single class π writes a variable x which is read by several other classes $\pi_1, \ldots, \pi_n$. Since $\pi_1, \ldots, \pi_n$ are unrelated, they may read x in different macro steps. Without the fork element, this leads to a critical situation: On the one hand, the value of x would have to be invalidated to prevent the reading classes to read the same value again, and on the other hand, the value of x would have to be kept valid so the stalling classes can read this value as soon as they are able to fire. Therefore, the fork broadcasts the valid signal as soon as a new value arrives from class π but individually determines the acknowledge for each class $\pi_1, \ldots, \pi_n$. Hence, each one has its own signals $\mathsf{valid}_{\mathsf{out}}(\pi_i)$ and $\mathsf{stop}_{\mathsf{in}}(\pi_i)$.

Join and fork elements can be also used to provide a wrapper to the environment, which reconstructs a synchronous interface. If we insert a fork element in front of all system classes (with respect to the classical order $\subseteq$ as defined in the previous section), and a join element behind of all classes, which implement two virtual classes $\pi_\bot$ and $\pi_\top$, reading and distributing all inputs and collecting and writing all outputs so that the interaction with the environment is synchronized. The right-hand side of Figure 1 shows these virtual classes.

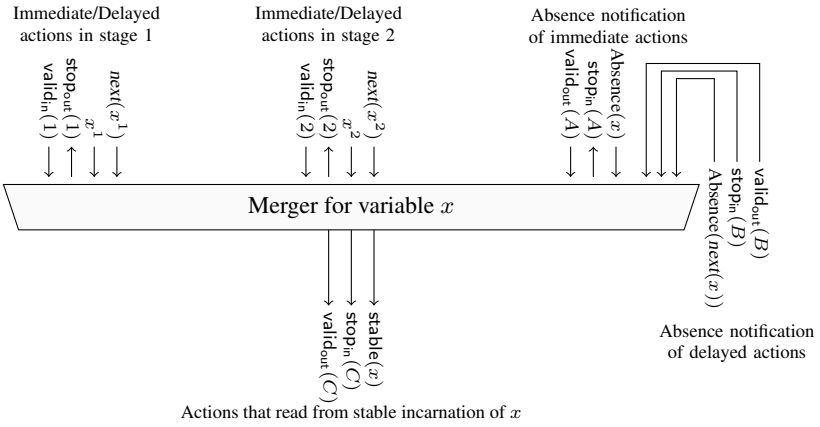

Figure 4. Merge Component for Variable x

As already mentioned, each class of the implementation may process another macro step but they all write to the same variables. Hence, for each variable x, a *merge component* Merge_x is needed to reorder values of x according to their macro steps. In principle, Merge_x waits until the value with the desired logical time arrives and forwards it. All other classes providing values which do not have the desired logical time stamp are stalled.

Figure 4 shows an exemplary structure of a Merge_x. In this example, two classes may write to x; therefore, each of these classes gets an interface to communicate with Merge_x. As the figure shows, Merge_x provides two input channels for each class - one for immediate and one channel for delayed writes to x. This distinction is necessary since immediate and delayed actions address different macro steps. However, both channels share the valid and stop signals because they stem from the same macro step.

Additionally, the interface also includes the valid and stop flags as already described above. With their help, the merge component Merge_x checks all incoming values for validity and the required logical time stamp. As long as the valid value for the required step is not available, $\mathsf{valid}_{\mathsf{out}}(\mathsf{Merge}_x)$ is unset. When the requested value becomes available, it is forwarded and the validity signal is set. Whenever a valid value arrives but does not have the requested time stamp, a stop signal is sent back, i. e. the sending class is stalled until the value is read. The logical time stamp for the currently required value is determined by an internal counter. As explained Section 2, each variable is written exactly once in each macro step, i. e. if a value arrives, the counter can be safely incremented by one.

Finally, the Merge_x is also responsible for the reaction to absence of variable x. It is implemented by two special input signals Absence(x) and Absence$(\mathit{next}(x))$, which are set iff no immediate action and delayed action can fire for a given macro step, respectively. These signals are driven by the class that succeeds the last class(es) containing immediate or delayed actions writing to x. In this case, Merge_x can assign a value to x, e. g. a default value or the value from the preceding macro step.

To the end, all required elements are available for creating the desynchronized system. The last step is to choose the correct channels, i. e. the connection of classes, in dependence of the target platform. In hardware synthesis, one would insert relay stations, and in software synthesis one can use queues with Lamport's clock synchronization to obtain a lightweight thread communication as already used in [2].

5 Summary

In this paper, we presented an approach to partition a synchronous program into desynchronized classes. The key for desynchronizing a partitioned system is the adaption of the system's classes to a defined protocol, i. e. each class must be able to wait for inputs, to signalize the validity of its outputs, and to wait for succeeding classes. In particular, the interface of each class has to be extended by four flags ($\mathsf{valid}_{\mathsf{in}}(\pi)$, $\mathsf{valid}_{\mathsf{out}}(\pi)$, $\mathsf{stop}_{\mathsf{in}}(\pi)$, $\mathsf{stop}_{\mathsf{out}}(\pi)$). Finally, attaching the join and fork elements to the classes enables us to use relay stations or queues to run our classes desynchronized. The advantage of our approach is the ability to use it in both hardware and software synthesis by only extending it by inserting corresponding channels, i. e. a hardware synthesis

would insert relay stations between classes to obtain a latency insensitive system, and a software synthesis would insert queues to obtain decoupled threads.

References

1. D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. In *Design, Automation and Test in Europe (DATE)*, Dresden, Germany, 2010. EDA Consortium.
2. D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Generating software pipelines for OpenMP. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, Dresden, Germany, 2010.
3. 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.
4. G. Berry. A hardware implementation of pure Esterel. *Sadhana*, 17(1):95–130, March 1992.
5. 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.
6. G. Berry. The constructive semantics of pure Esterel. `http://www-sop.inria.fr/esterel.org/`, July 1999.
7. M. Boldt, C. Traulsen, and R. von Hanxleden. Compilation and worst-case reaction time analysis for multithreaded Esterel processing. *EURASIP Journal on Embedded Systems*, 2008. Article ID 594129.
8. J. Brandt and K. Schneider. Separate compilation for synchronous programs. In H. Falk, editor, *Software and Compilers for Embedded Systems (SCOPES)*, volume 320 of *ACM International Conference Proceeding Series*, pages 1–10, Nice, France, 2009. ACM.
9. J.A. Brzozowski and C.-J.H. Seger. *Asynchronous Circuits*. Springer, 1995.
10. L.P. Carloni. The role of back-pressure in implementing latency-insensitive systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 146(2):61–80, 2006.
11. L.P. Carloni, K.L. McMillan, and A. Sangiovanni-Vincentelli. Latency insensitive protocols. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 123–133, Trento, Italy, 1999. Springer.
12. L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, 2001.
13. J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic circuits. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 28(10):1437–1455, October 2009.
14. P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3):416–427, 1999.
15. M.R. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *Design Automation Conference (DAC)*, pages 576–581, San Diego, CA, USA, 2004. ACM.
16. K.M. Chandy and J. Misra. *Parallel Program Design*. Addison Wesley, Austin, Texas, May 1989.
17. J. Cortadella, M. Kishinevsky, and B. Grundmann. SELF: Specification and design of synchronous elastic circuits. In *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, 2006.
18. J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *Design Automation Conference (DAC)*, pages 657–662, San Francisco, CA, USA, 2006. ACM.

19. 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.
20. A. Girault and X. Nicollin. Clock-driven automatic distribution of Lustre programs. In R. Alur and I. Lee, editors, *International Conference on Embedded Software (EMSOFT)*, volume 2855 of *LNCS*, pages 206–222, Philadelphia, PA, USA, 2003. Springer.
21. N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.
22. N. Halbwachs. A synchronous language at work: the story of Lustre. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 3–11, Verona, Italy, 2005. IEEE Computer Society.
23. 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.
24. S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. In A. Gupta and P. Manolios, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 19–30, San Jose, California, USA, 2006. IEEE Computer Society.
25. L. Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Cooperation, 1991.
26. G. Logothetis and K. Schneider. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *Design, Automation and Test in Europe (DATE)*, pages 10196–10203, Munich, Germany, 2003. IEEE Computer Society.
27. F. Rocheteau and N. Halbwachs. Pollux, a Lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, Chateau de Bonas, 1991.
28. K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 143–154, Newcastle upon Tyne, UK, 2001. IEEE Computer Society.
29. K. Schneider. Proving the equivalence of microstep and macrostep semantics. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 2410 of *LNCS*, pages 314–331, Hampton, VA, USA, 2002. Springer.
30. K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.
31. K. Schneider and J. Brandt. Performing causality analysis by bounded model checking. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 78–87, Xi'an, China, 2008. IEEE Computer Society.
32. 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.
33. 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.
34. K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Improving constructiveness in code generators. In *Synchronous Languages, Applications, and Programming (SLAP)*, pages 1–19, Edinburgh, Scotland, UK, 2005.
35. K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In J. Desel and Y. Watanabe, editors, *Application of Concurrency to System Design (ACSD)*, pages 106–115, St. Malo, France, 2005. IEEE Computer Society.
36. T.R. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1996.