Jens Brandt

Synchronous Models for Embedded Software

Vom Fachbereich Informatik der Technischen Universität Kaiserslautern
zur Erlangung der Lehrbefugnis für das Fachgebiet Technische Informatik
angenommene Habilitationsschrift

Table of Contents

1 Introduction

1.1 Motivation

Compared to traditional software design, the design of embedded software is even more challenging: In addition to the correct implementation of the systems, one has to consider non-functional constraints such as real-time behavior, reliability, and energy consumption. Moreover, many embedded systems are used in safety-critical applications where errors can lead to enormous damages and even to the loss of human live. For this reason, formal verification is applied in many design flows using different kinds of formal verification methods.

The synchronous model of computation [74] has shown to be well-suited in this context. Its core is the paradigm of perfect synchrony which assumes that the overall system behavior is divided into a sequence of reactions, and all computations within a reaction are completed in zero time. This temporal abstraction simplifies reactive programming in that developers do not have to bother about many low-level details related to timing, synchronization and scheduling.

The introduction of a logical time scale is not only a very convenient programming model, it is also the key to generate *deterministic* single-threaded code from multi-threaded synchronous programs. Thus, synchronous programs can be directly executed on simple micro-controllers without using complex operating systems. Another advantage is the straightforward translation of synchronous programs to hardware circuits [76, 219, 229]. Furthermore, the concise formal semantics of synchronous languages makes them particularly attractive for reasoning about program properties and equivalences [37, 78, 225, 226, 228]. Finally, they allow developers to determine tight bounds on the reaction time by a simplified worst-case execution time analysis (since loops do not appear in reaction steps) [189, 190, 231, 232]

Although several success stories have already been reported [74] from safety-critical applications like avionic and automotive industries, there is still a need for further research on efficient compilation of synchronous languages. This is mainly due to the fact that embedded system architectures are getting highly parallel systems.

First, multicore processors have already replaced single-core processors in desktop computers, and they are more and more frequently used in embedded systems. Hence, the software code generators used in model-based design for the development of embedded systems have to be modified accordingly so that the increased performance offered by multicore processors can be effectively utilized.

Second, many embedded applications in the automotive or avionic industry are implemented on a heterogeneous set of distributed processing elements. Using these architectures, it is in general not reasonable to maintain a single abstraction of time, since the resulting performance would be unacceptably low. Instead, the use of several time scales (or clocks) is often considered: the higher the abstraction level, the slower is the corresponding clock. A somehow extreme case are GALS

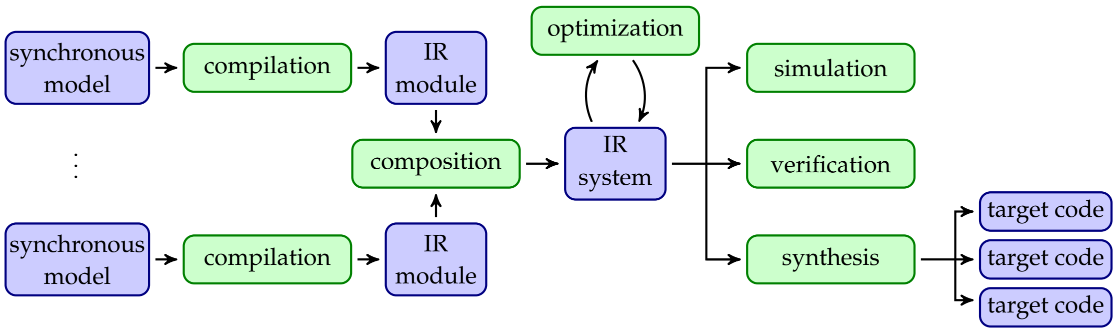


Fig. 1. Design flow

(globally asynchronous, locally synchronous) systems [99, 136], which perfectly match real-world systems. In a GALS system, a number of synchronous components are asynchronously connected, e. g. by sending messages over asynchronous 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. Moreover, to avoid expensive synchronizations between the components, these *parallel systems run asynchronously to each other*, and they are therefore distributed systems (even though they may still be contained in a small system).

Nevertheless, it is natural to develop a model-based design flow for embedded systems where one starts with a synchronous model. Figure 1 shows the design flow, which is considered in this thesis.

1.2 Contributions

This thesis is dedicated to this design flow, and it presents my contributions to it.

- System design starts with developers writing synchronous models, which are translated by a *compiler* to modules of an intermediate representation (IR). The compilation procedure proposed in this thesis makes it possible to compile modules separately, which enables the generation of libraries and IP-based design. Due to the orthogonal design of synchronous languages, which allows to nest preemption with all other statements, this is a very difficult problem. Furthermore, this thesis contributes a formal verification (inside an interactive theorem prover) of critical parts of the compiler.
- The compiler modules are subsequently composed with other modules to a complete system. While the synchronous composition of models is straightforward, non-trivial problems must be solved after the composition: as the synchronous abstraction of time implicitly schedules all computation in the system according to the data dependencies, the composition of several modules might introduce cyclic dependencies, which represent incorrect system. In the domain of synchronous languages, the static analysis to detect (and possibly remove) cyclic dependencies is known as *causality analysis*. This thesis contributes the first causality analysis for a imperative synchronous languages with non-Boolean data-flow. Furthermore, it explores its theoretical foundations, which show the effect of different variants of compilation to it.
- The next step in the design flow is the *optimization* of system models. This thesis presents the first data-flow analysis which is custom tailored for synchronous models. It allows to identify redundant computations (so-called passive code), which can be suppressed to improve run-time performance of the system.

- The intermediate representation is still a synchronous model so that we benefit from its advantages. In particular, the simulation of the model is deterministic and precise. Only the final *synthesis* step maps the synchronous model to a target architecture which does not directly support this model of computation. This thesis contributes some approaches to decompose and desynchronize a synchronous system so that it can be implemented by loosely coupled multi-threaded implementations or on a distributed set of processors.
- The synchronous abstraction of time simplifies *verification*. However, as the final implementation may be mapped to a distributed architecture, a natural question is which kind of properties are preserved and which verification results are still valid after desynchronization. This thesis answers this fundamental question and characterizes the set of preserved properties.
- This thesis also considers the problem how the modeling of asynchronous parts in synchronous systems can be supported in order to simplify the desynchronization and architecture mapping task. In particular, it proposes the introduction of a clock tree into an imperative synchronous language, which avoids the common problem of over-synchronization. In addition to the extension itself, it shows which effects to the design flow are caused by the extension.
- Finally, this thesis addresses the *integration* of models following a different model of computation in the design flow. As real systems are rather an amalgam of different descriptions than a clean set of synchronous models, a well-defined composition of heterogeneous models is a very important aspect. This thesis shows how synchronous, polychronous and asynchronous models can be integrated. To this end, it defines a common intermediate representation, which is targeted by newly developed compilation procedures. Furthermore, the relationship to discrete-event models, which are frequently used for system simulation, is discussed.

1.3 Structure

Each of the following chapters considers another activity in the design flow. Chapter 2 starts with the compilation of synchronous models to the target-independent intermediate representation, which is the starting point for the analyses, optimizations, verification and synthesis procedures of the following chapters. Chapter 3 addresses the important question of causality analysis, which is an important static analysis checking the schedulability of a synchronous system. Chapter 4 deals with custom optimizations in the domain of synchronous systems, and Chapter 5 shows how synchronous systems can be mapped to parallel target architectures. Chapter 6 summarizes recent efforts to the verification in the context of desynchronized systems. Chapter 7 shows how the modeling capabilities of synchronous programs can be extended to better target today's embedded systems. Chapter 8 explains how components following a different model of computation can be integrated with synchronous ones. Finally, Chapter 9 contains a structured and annotated list of all my publications, which describes my own contribution to all the publications contained in my habilitation.

2

Compilation

2.1 Context

The synchronous abstraction of time simplifies reactive programming in that developers do not have to bother about many low-level details related to timing, synchronization and scheduling. However, this abstraction is not for free: the compilation of synchronous languages is more difficult than the compilation of traditional sequential languages. The concurrency available in synchronous programs has to be translated such that the resulting code can be executed on ordinary sequential machines. To this end, special problems must be solved by compilers as this chapter and the following one will show. Furthermore, several aspects which are self-evident for other programming languages are very difficult – or even impossible to implement: modular compilation is such an example.

In this section, we first introduce the imperative synchronous language Quartz, which will the primary modeling language of this thesis. Then, we discuss the general issue of modular compilation in the context of synchronous languages before we briefly review other approaches of compiling imperative synchronous languages.

2.1.1 The Synchronous Language Quartz

In the following, we consider the synchronous language Quartz [229], which has been derived from Esterel [79, 81]. In the following, we give a brief overview of the language core, which is sufficient to define most other statements as simple syntactic sugar. For each statement, we will subsequently describe its behavior. For the sake of simplicity, we do not give a formal definition; the interested reader is referred to [229], which also provides a complete structural operational semantics. The Quartz core consists of the following statements, provided that S, S_1, and S_2 are also core statements, ℓ is a location variable, x and τ are a variable and an expression of the same type, σ is a Boolean expression, and α is a type:

`nothing`	(empty statement)
ℓ : `pause`	(start/end of macro step)
$x = \tau$ and `next`$(x) = \tau$	(assignments)
`if`(σ) S_1 `else` S_2	(conditional)
$S_1; S_2$	(sequence)
`do` S `while`(σ)	(iteration)
$S_1 \parallel S_2$	(synchronous concurrency)
`[weak] [immediate] abort` S `when`(σ)	(preemption: abortion)
`[weak] [immediate] suspend` S `when`(σ)	(preemption: suspension)
$\{\alpha\ x;\ S\}$	(local variable x of type α)
inst : *name*$(\tau_1, \ldots, \tau_n)$	(call of module *name*)

In imperative synchronous languages, the synchronous MoC is represented as follows: All statements are assumed to be executed in zero-time. The only exception is a special statement `pause`, which implements the end of the current macro step. A simplified view on the programs is therefore as follows: In each macro step, a synchronous program resumes its execution at the `pause` statements where the control flow has been stopped at the end of the previous macro step, then it reads new inputs and executes the following statements until the next `pause` statements are reached. A `pause` statement also defines a control flow location implemented by a unique Boolean valued label ℓ, which is assumed to be true iff the control flow is currently at the statement ℓ : `pause`. Since all other statements are executed in zero time, the control flow can only rest at these positions in the program.

$a = 1;$	$b = a;$	$a = 1;$
$b = a;$	$a = 1;$	`if`$(b = 1)$ $b = a;$
`pause;`	`pause;`	`pause;`
$a = b;$	$a = b;$	`if`$(a \neq 2)$ $a = b;$
(a)	(b)	(c)

Fig. 2. Three Quartz programs illustrating the synchronous MoC

Variables (or often called signals in the context of synchronous languages) of the synchronous program can be modified by assignments. They immediately evaluate the right-hand side expression τ in the currently environment/macro step. Immediate assignments $x = \tau$ instantaneously transfer the obtained value of τ to the left-hand side x, whereas delayed ones `next`$(x) = \tau$ transfer this value in the following macro step. If a variable is not set by an action in the current macro step, its value is determined by the so-called *reaction to absence*, which depends on the *storage type* of the variable. Quartz knows two of them: *memorized variables* keep the value of the previous step, while *event* variables are reset to a default value if no action sets their values.

The assumption that all Quartz statements are executed in zero-time has some consequences which might be confusing at a first glance: if a statement does not take time for its execution, it is evaluated in the same variable environment as another statement following it in a sequence. In principle, both statements may therefore be interchanged without changing the behavior of the program. So, the program in Figure 2 (b) has the same behavior as the program in Figure 2 (a). Thus, each statement knows and depends on the results of all operations in the current macro step. Obviously, this generally leads to the causally cycles described in the previous section, which are not present in traditional sequential programming languages. Hence, a Quartz statement may influence its own activation condition (see the program in Figure 2 (c)).

```
{                        ‖  {
  b = true;              ‖                              a = false, b = true, c = false
  ℓ1 : pause;            ‖    ℓ3 : pause;
  if(a) b = false;       ‖    if(¬b) c = true;
                         ‖    a = true;                 a = true, b = false, c = true
  ℓ2 : pause;            ‖    ℓ4 : pause;
                         ‖    b = true;                 a = true, b = true, c = true
}                        ‖  }
```

Fig. 3. Synchronous concurrency in Quartz

```
abort {          weak abort {     immediate abort {   weak immediate abort {
  a = 1;           a = 1;           a = 1;              a = 1;
  ℓ1 : pause;      ℓ1 : pause;      ℓ1 : pause;         ℓ1 : pause;
  b = 2;           b = 2;           b = 2;              b = 2;
  ℓ2 : pause;      ℓ2 : pause;      ℓ2 : pause;         ℓ2 : pause;
} when(true);    } when(true);    } when(true);       } when(true);
c = 3;           c = 3;           c = 3;              c = 3;
```

Fig. 4. Abort variants: strong, weak, immediate strong and immediate weak

In addition to the usual control flow statements known from typical imperative languages (conditionals, sequences and iterations), Quartz offers synchronous concurrency. The *parallel statement* $S_1 \parallel S_2$ immediately starts the statements S_1 and S_2. Then, both S_1 and S_2 run in lockstep, i. e. they automatically synchronize when they reach their next `pause` statements. The parallel statement runs as long as one of the sub-statements is active.

Figure 3 shows a simple example consisting of two parallel threads. In the first step, the program, and thus both threads, are started. The first thread executes the assignment to b and stops at location ℓ_1, while the second thread directly moves to location ℓ_3. In the second macro step, the program resumes are the labels ℓ_1 and ℓ_3. Since the second thread contains an immediate assignment to a, the action resetting b in the first threads is activated, which in turn activates the actions setting c in the second thread. The last step then resumes from ℓ_2 and ℓ_4, where the second thread performs the final assignment to variable b.

Preemption can be conveniently implemented by the `abort` and `suspend` statements. Their meaning is as follows: A statement S which is enclosed by an `abort` block is immediately terminated when the given condition σ holds. Similarly, the `suspend` statement freezes the control flow in a statement S when σ holds. Thereby, two kinds of preemption must be distinguished: strong (default) and weak (indicated by keyword `weak`) preemption. While strong preemption deactivates both the control and data flow of the current step, weak preemption only deactivates the control flow, but retains the current data flow of this macro step. The immediate variants check for preemption already at starting time, while the default is to check preemption only after starting time.

Figure 4 shows examples of all four abort variants. The execution of the first two statements requires two macro steps: in the first macro step, the abort block is entered without checking the condition, and a is set. In the second macro step, the condition is checked. While the strong variant immediately aborts and continues with the assignment to c, the weak one first completes the macro step by setting b and then moves to the assignment of c. The last two fragments feature immediate version of the abort statement. Since the abortion condition is checked when the control flow enters, both examples only need one step for their execution: in the strong immediate variant only c is set, while the weak immediate one additionally assigns a.

Finally, modular design is supported in imperative synchronous languages by the declaration of modules in the source code and by calling these modules in statements. Any statement can be encapsulated in a module, which further declares a set of input and output signals for interaction with its context statement. There are no restrictions for module calls, so that modules can be instantiated in every statement. In contrast to many other languages, a module instantiation can also be part of sequences or conditionals (and is therefore not restricted to be called as additional thread). Furthermore, it can be located in any abortion or suspension context, which possibly preempts its execution.

2.1.2 Modular Compilation

It is natural to split the overall compilation process into several phases, such that the first steps are independent of the final target code. For obvious reasons, it is desirable to store the intermediate results so that they can be reused for further compilation: First, expensive optimizations may have already been performed on the intermediate results. In particular, complex statements like the interaction of concurrent threads with abortion and suspension statements as well as instantaneous broadcast communication can already be reduced to simpler statements in a first compilation phase. Second, the intermediate code can be distributed in libraries without revealing its proprietary source (which leads to the ideas of IP-blocks used in hardware design).

The term *module* is also a very fuzzy term which we have to make more precise. In our case, modules are provided by module definitions of the source language, which are compiled to corresponding modules of the intermediate language. Modules can be used by 'calling' (or better instantiating) them at an arbitrary place in the program.

- The first view, which is used by most hardware description languages, considers modules as independent components that communicate over their interfaces. Most of these languages explicitly distinguish between behavior and structure. Modules are defined by the behavioral part of the language, and they are subsequently assembled by the structural part. Although there might be a structural hierarchy (which does not change over runtime), modules of systems created by this approach all run in parallel and thus, in the same context. As there are no context statements that could preempt or restart the behavior, this kind of symmetric linking is very simple. VHDL, SDL or even threads in all classical imperative programming languages follow this simple, but limited, approach.
- The second approach, which is followed by imperative synchronous languages, assumes that module instantiations are orthogonal to all other statements of the language. Thus, module instantiations can be used everywhere in the synchronous program. This gives the developer a higher level of abstraction, since sophisticated control flow as sequential/parallel execution or preemption is directly modeled in the program. A basic precondition for this approach is that linking is asymmetric since if one module instantiates another module, then the calling module defines the context of the called one. This is the view classic software languages usually have: A module is 'called' from another one.[1]

For the latter case, two degrees of modularity can be distinguished, which we call *incremental* and *separate* compilation:

- *Incremental compilation* requires that the compiled code for the inner module is available when the outer one is compiled. Then, the compiler is able to *simply include the already compiled code* to the remaining part of the system. All compilation procedures like [37, 41, 78] that are defined

[1] Obviously, the second approach comprises the first one: If two modules are run in parallel, the description of their connection can be seen as a separate module consisting of parallel module calls.

by a single recursive traversal over the syntax tree of the program can be easily generalized to implement incremental compilation by (1) simply storing all the compilation results of a compiled module in a file and (2) importing these results whenever a module instantiation of this module is compiled.

- In contrast, *separate compilation* is much more complex: It allows one to compile a module without having any knowledge about the called modules except for their interfaces, since *inclusion of called modules is deferred to a later linking step*. Hence, even modules can be compiled that call modules that are not yet implemented. This allows developers to compile all modules of the system in an arbitrary order and to link them at the end of the development process. Furthermore, separate compilation allows one to change a called module without the need to recompile the modules calling it (only new linking is required). Clearly, this is a very important feature to efficiently create and maintain systems consisting of many modules.

Separate compilation is a very difficult problem in the context of synchronous languages. If the compilation target is sequential code, it is even the case that modularity always has some price. Previous work has addressed this problem in detail. As shown in [191, 192, 211], there is a tradeoff between efficiency and reusability. The more the resulting code is sequentialized, the more constraints it imposes on its usage (by special interfaces). Alternatively, one may perform a partial causality analysis on single modules [254, 256]. However, it still has to be reworked and completed at the end, since the solution of causality problems requires information about all actions reading and writing to a variable that is only available at the end. In the context of Esterel or Quartz, separate compilation has to consider potential preemption contexts of a module and potential surrounding loops that may lead to further schizophrenia problems that did not exist when compiling a module on its own. For example, assume that some (inner) module has been compiled to intermediate code, and it is later on used in another (outer) module. A first problem that might appear is that it can be called within a preemption statement so that its behavior must be adapted to the potential preemptions. Another problem might be that its outputs are bound to local variables in the outer module. This may lead to schizophrenia problems, which did not exist when the inner module was compiled. Similarly, additional problems arise when compiling the outer module without any information about termination or instantaneity of the inner one (which is desirable so that changes on inner modules do not imply recompilation of outer modules). None of the publications mentioned above has addressed this problem.

2.1.3 Compilers for Imperative Synchronous Languages

As observed in [116, 121, 209] there are several fundamentally different techniques which have been developed to compile synchronous programs during the last two decades.

The first generation of compilers [81] used the *Structural Operational Semantics (SOS)* rules of Esterelto translate the program to an extended finite state machine whose transitions are endowed with corresponding code fragments. As it is well-known, this automaton may have exponentially many states in terms of the size of the given program, which makes this compilation technique applicable only to small programs. Nevertheless, this method generates the fastest code for software on a uniprocessor system, as the compiler generated exactly the code that has to be executed for each particular control state of the program.

Polynomial compilation was first achieved by a direct translation to *equation systems* [37, 75, 205, 225]. The idea behind this approach is to consider only control flow locations instead of entire control states[2]. The compiler determines (1) the value for every output, and (2) the value of all

[2] We distinguish between control flow *states* and control flow *locations*: A state is a vector of control flow locations, and a control flow location is a place in the program that can hold the control flow for an instant of time. In case of Esterel, control flow locations are essentially `pause` statements.

control flow locations at the next instant of time, both in terms of the current inputs and currently active control flow locations. The approach is particularly efficient for hardware synthesis, since hardware is able to execute all equations in parallel, and the obtained circuits' sizes are only quadratic in terms of the given program. Hence, this approach is successfully used for hardware synthesis and is still the core of commercial compilers. A software approach on a uniprocessor system, however, has to check all equations one after the other, even those that correspond with control flow locations that are currently not active, which leads to significant performance penalties for programs that contain only a limited degree of concurrency.

A third approach has been followed by the Saxo-RT compiler [103, 104] of France Telecom, which translates the program into an *event graph*. Hence, an event driven simulation scheme can be used to generate code, which is compiled into efficient C code. In contrast to our approach, this approach does not retain the original program structure and therefore is not able to exploit structural properties of the program like mutual exclusion of substatements in sequences or conditionals.

A fourth approach is based on the translation of programs into *concurrent control data flow graphs* [116, 117, 121, 209], whose sizes depend linearly on the given program. At each instant, the control flow graph is traversed until nodes are reached that correspond with active control flow locations. Then, the corresponding subtrees are executed which changes the values of the control flow locations (that are maintained in Boolean variables).

All these compilers use different intermediate formats [40, 116, 122, 142, 201, 209] in their design flow. The most important ones are:

- *Imperative/Intermediate Code (IC):* The IC format is used by imperative languages like Esterel, Argos and Statecharts. It is the compilation result obtained after parsing, expanding macro statements, and type-checking. Hence, no essential code generation is performed, so that the format still offers concurrency, exceptions, local variables and calls to other modules. Compared to AIF, the format is close to the kernel Esterel language, and thus, it still contains much of the complexity of the source language.
- *Object Code (OC):* Object code, also called automaton code, is a low-level format that describes a finite state automaton by explicitly listing its states and transitions (and the code that is to be executed along the transitions). Hence, concurrency and complex interaction of threads has already been eliminated by the compiler. Object Code can be also created from guarded actions by re-encoding the control states such that a single label (one-hot) represents each possible control flow position.
- *(Sorted) Sequential Circuit Code (SC and SSC):* These formats describe a hardware circuit that is obtained by compilation of Esterel programs. The unsorted version SC may contain combinatorial cycles, and the used gates are written in no particular order. In contrast, the SSC format makes use of a topological order and describes an acyclic circuit that may be obtained from a cyclic one by solving the corresponding causality problems. This intermediate format can be directly derived from guarded actions by grouping them according to their targets.
- *Declarative Code (DC):* The DC format is the privileged interchange format for hardware circuit synthesis, symbolic verification and optimization as well as distributed code generation. The format reflects declarative or data flow synchronous programs like Lustre, as well as the equational representations of imperative synchronous programs. The underlying idea is the definition of flows by equations governed by clock hierarchies. It is the successor of the previously used GC format [62] and is closely related to the sequential circuit code formats.
- *Event-Triggered Graphs:* Event-based simulation is a characteristic of hardware description languages like VHDL or Verilog. A compilation technique implemented in the Saxo-RT compiler of France Telecom [103, 104] also employed an event-triggered approach, and therefore relies

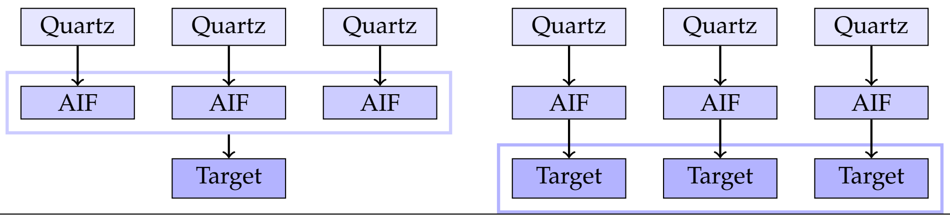


Fig. 5. Modular compilation and modular synthesis of Quartz programs

on an internal data structure called an *event graph* that keeps track of dependencies of events. Hence, an event driven simulation scheme can be used to generate sequential code.

- *Concurrent Control Flow Graphs (CCFG):* The compiler of the University of Columbia translates Esterel programs to *concurrent control data flow graphs* [116, 117, 119, 122, 209]. At each instant, the control flow graph is traversed until nodes are reached that correspond with active control flow locations. Then, the corresponding subtrees are executed which changes the values of the control flow locations (that are maintained in Boolean variables). Hence, concurrency is still available, but sequential execution is also supported. Moreover, the format lends itself well for interpretation and software code generation for different architectures.

We decided against the usage of all these intermediate formats in work, since none of them contains the information needed for a separate compilation as described in the previous section.

2.2 Contribution

2.2.1 Separate Compilation [23, 27]

Several publications [20, 23, 27] address the problem of separate compilation of Quartz programs. The first contribution contained in these publications is the definition of a design flow.

As we target the design of embedded systems, where hardware-software partitioning and target platforms are design decisions that are frequently changed, persistent intermediate results in a well-defined and robust format are welcome. Therefore, we split up the design flow into two steps, which are bridged by the Averest Intermediate Format (AIF). This intermediate format models the system behavior in terms of synchronous guarded actions. Hence, complex control-flow statements need no longer be considered. We refer to *compilation* (considered in the following) as the translation of source code into AIF, while *synthesis* (considered in Chapter 5) means the translation from AIF to the final target code, which may be based on a different model of computation.

Figure 5 shows two approaches of generating target code from a set of Quartz modules. *Modular compilation*, which is shown on the left-hand side, translates each Quartz module to a corresponding AIF module. Then, these modules are linked on the intermediate level before the whole system is synthesized to target code. *Modular synthesis*, which is shown on the right-hand side, translates each Quartz module to a corresponding AIF module, which is subsequently synthesized to a target code module. Linking is then deferred to the target level. While modular synthesis simplifies the compilation (since all translation processes have to consider only a module of the system), it puts the burden on the runtime platform or the linker which have to organize the interaction of the target modules correctly.

Our contribution focuses on separate translation from Quartz modules to corresponding AIF modules, which results to a compilation procedure that supports the full modularity provided by the source language with unrestricted use of local variables, preemption statements, and delayed assignments – without any tradeoffs. In our approach, additional information is added to the intermediate format (about the context of a module and a new handling of incarnations) to achieve a modular compilation of imperative synchronous programs. This information is essential for binding. Otherwise, it is impossible to determine when the guarded actions of a called module should be activated, aborted and suspended, or how many times a called module must be replicated due to reincarnations.

As our separate translation does not generate sequential code, it does not have to deal with the scheduling and causality problems previous publications [191, 192, 211] considered. Instead, they can be solved later. Since the source and intermediate code are both based on the synchronous model of computation, they can model the same causality problems. For these reasons, we do not consider causality analysis in our approach, although we also believe that a modular approach to causality analysis is desirable. However, we postpone it to a later compilation phases.

The compilation procedure presented in [20, 23] is the first algorithm that supports fully separate compilation for a typical imperative synchronous language such as Esterel or Quartz. Modules of the source language are translated to individual modules in an intermediate format, which does not require any information about the called modules except for their names and the types of their inputs and outputs. Our contribution is based on two new developments: First, we define a new intermediate code format called AIF (Averest Intermediate Format), which is a target-independent representation of the system in terms of simple guarded actions. Second, we develop a compilation procedure to generate this AIF code.

A particular advantage of our separate compilation is that it also handles the *instantaneous reincarnation of local variable declarations* (often known as *schizophrenia problems* [37, 78, 205, 230, 243]) in a modular way: This means that even though module calls inside loops may generate additional reincarnations of local variables, the compiler does not have to take care about these reincarnations in an explicit way. Instead, the required reincarnations are implicitly made by generating copies of module calls in the different surfaces of the nested loops. This solution requires a sophisticated generation of names and related data structures in the intermediate code.

As a side effect, our separate compilation enables new possibilities for intellectual property (IP) design in synchronous languages. In general, using these IP cores gives rise to some problems, of which the most challenging one is the support of an adequate infrastructure for their integration. In general, there are several alternatives: First, it might be the case that the module is available as a *soft core* in a specific description language, and the user must use this language in order to use the module. Second, the IP provider may distribute a *hard core* and makes use of a high-level integration mechanism, which allows the integration of several modules written in different languages but using the same integration mechanism. This is based on the principle that there are two levels of a description language: the language in which the module is developed and a language that is merely used for the integration, i.e., a meta-language [141].

The approach presented in [27] provides a high-level integration of IP cores for synchronous languages. Developers can use the synchronous programming language both for the creation of new native modules *and* their integration with proprietary IP cores. This provides the developer with a high-level interface, since all modules can be used without any restrictions in the program (in particular, also in preemption statements).

2.2.2 Synchronous Guarded Actions [20]

As already mentioned in Section 2.2.1 another contribution of [20, 23] is the Averest Intermediate Format (AIF), an intermediate representation for the compilation of synchronous languages. The behavior of a system is described in AIF with the help of synchronous guarded actions. As the name suggests, they are designed in the spirit of classical *guarded commands* [98, 113, 168], which are a well-established formalism for the description of concurrent systems. However, note that in contrast to many other applications where guarded commands are used, the guarded actions considered here follow the *synchronous abstraction of time* as described above. The system is represented by a set of synchronous guarded actions of the form $\langle \gamma \Rightarrow \mathcal{A} \rangle$ defined over a set of variables $\mathcal{V}$. The Boolean condition γ is called the guard and $\mathcal{A}$ is called the action of the guarded action. In our case, guarded actions are either

- $\gamma \Rightarrow x = \tau$ (immediate assignment),
- $\gamma \Rightarrow \texttt{next}(x) = \tau$ (delayed assignment),
- $\gamma \Rightarrow \texttt{assume}(\sigma)$ (assumption), or
- $\gamma \Rightarrow \texttt{assert}(\sigma)$ (assertion).

```
module Example(
    nat ?i1, ?i2,
    nat !o1, event nat !o2)
{
  nat x = 0;

  i1 > 5 => o1 = i1 + x + 1;
  i1 < 5 => o1 = i1 + o2;
  o1 > 10 => next(x) = i1;
  o1 < 10 => next(x) = i2;
  i1 > 5 => o2 = i2 + o1;
  i1 < 5 => o2 = i2 + x + 1;

  true => assume(i1>0);
  true => assume(i2>0);
}
```

Fig. 6. Synchronous guarded actions

Both kinds of assignments evaluate the right-hand side expression τ in the current macro step. Immediate assignments $x = \tau$ write the obtained value of τ immediately to the variable x, whereas delayed ones $\texttt{next}(x) = \tau$ write the value in the following step. If there is not any action which determines the current value of a variable x (i. e. immediate ones of the current step and delayed ones of the previous step), the variable x will be set by the *default reaction*. The default reaction generally depends on the storage type of a variable: event variables (indicated by the modifier `event`) are reset to false/zero, while (ordinary) memorized variables keep the value from the previous step (and get the default value in the initial step).

Immediate assignments define a causal dependency within the instant from all the read variables (i. e. variables in the guard γ and on the right-hand side τ) to the written variable x. The former ones must be known before the value of x becomes known. In contrast, delayed assignments do not have causal dependencies within the instant since x is written in a different instant. Assumptions $\texttt{assume}(\sigma)$ provide a condition σ the developer guarantees, i. e. they restrict the set of states the user cares about. In contrast, an assertion $\texttt{assert}(\sigma)$ defines a verification goal σ, which has to be discharged. Both of them do not impose any causal dependencies since they do not change values.

An example for a synchronous system which is described by guarded actions is given in Figure 6. The system has the inputs `i1`, `i2`, the outputs `o1`, `o2`, and uses the local variable `x`. The guarded actions are synchronously evaluated based on the given inputs. An example execution

	1	2	3	4	5	...
`i1`	6	5	1	5	9	...
`i2`	2	4	6	8	10	...
`x`	0	2	4	1	5	...
`o1`	7	7	12	12	15	...
`o2`	9	0	11	0	25	...

Fig. 7. Trace of example in Figure 6

trace is given in Figure 7. The evaluation order of the guarded actions is based on the data dependencies. In the first instant, the input `i1` has the value `6`. Due to the guards of the actions, `o1` is written by the first guarded action and the value of `o2` is needed for that. In the third instant, the input `i1` is `1` and, due to the guards, the value of `o1` is needed to determine the value of `o2`. Thus, the variables are computed in a different order for those instants. The delayed assignments to `x` transfer the assigned value to the following instant. Notice the different behavior of `o1` and `o2` if there is not any action writing them as in the second and forth instants. Since `o1` is a memorized variable, it keeps its previous value, while `o2` is reset to zero.

2.2.3 Translation to Jobs [40, 41]

Another compilation approach is proposed in [40, 41], which is rather related to the concurrent control data flow graphs approach (see Section 2.1.3). However, we do not translate the program into control data flow graphs, and neither do we store the control flow in Boolean variables. Instead, we generate for each program location a corresponding job that consists of an instantaneous statement (one that does not contain macro steps). Hence, if we know the currently active control flow locations, we simply have to execute the corresponding jobs. In addition to generating outputs, the execution of these code fragments will also generate a set of control flow locations that are active at the next instant of time.

In contrast to control flow graph based approaches, we therefore have no need for a selection tree, i.e. to find in the control flow graph the subtrees that have to be executed next. Instead, we directly execute the jobs that have been computed by the compiler. Moreover, our compilation technique follows closely the given program structure, so that a formal verification as already done for hardware synthesis [37, 225, 226, 228] can be achieved to show the correctness of this approach. This can also be used to obtain a validating compiler that not only delivers compiled code, but additionally a formal proof of the correctness of the compilation.

2.2.4 Verification of Compilation [36, 37]

As it turned out that the compilation of synchronous languages is very challenging involving subtle problems, we believe that the research in this area greatly benefits from a verified compilation procedure. To this end, we embedded [226] Quartz in the HOL theorem prover [138]. This embedding allows us to reason not only about particular Quartz programs, but also about the semantics of Quartz. Previous work already addressed the correctness of the synthesis of equation systems [225] and the equivalence to SOS rules [228]. In [36, 37] we verified the compilation of Quartz to intermediate code.

3

Causality Analysis

3.1 Context

3.1.1 Causality Problems

Synchronous languages [68, 74, 142, 180, 180] follow the *paradigm of perfect synchrony:* the execution of a program is divided into macrosteps [146] that consist of finitely many microsteps. From a programmer's view, the execution of a microstep requires no time, and reactions (outputs) of the system respond immediately to actions (inputs) of the environment. In contrast, macrosteps all require the same *logical amount of time*.

Since synchronous languages allow programs to read their own outputs, mutual dependencies between actions and their trigger conditions may lead to so-called causality cycles [78]. Such cycles may lead to inconsistencies so that no code can be generated. In many cases, however, causality cycles can be resolved and deterministic code can be generated. In any case, the synchronous programming paradigm challenges the compilers [118]. Before code generation can be performed, the compilers have to solve *causality problems* [78, 85, 234, 237].

To analyze causality cycles, one has to check whether the considered program has a consistent and unambiguous behavior for all inputs and all reachable states. To this end, typically a formula of type $\forall \mathbf{x}.\exists_1 \mathbf{y}.\Phi(\mathbf{x},\mathbf{y})$ has to be proved, which expresses that for all inputs and current states, there must be uniquely determined values for the next states and the current outputs. In the worst case, this requires to consider the program for all inputs, outputs, and all reachable states, which makes this problem highly complex[1].

Note, however, that the above problems do not solve the more general problem to check the validity of a (propositional) formula of type $\forall \mathbf{x}.\exists_1 \mathbf{y}.\Phi(\mathbf{x},\mathbf{y})$. Instead, causality analysis may be viewed as a heuristic to solve this problem in the following sense: if causality analysis can prove that there are unique outputs for all inputs, the validity problem has been solved. However, if causality analysis fails, it may still be the case that $\forall \mathbf{x}.\exists_1 \mathbf{y}.\Phi(\mathbf{x},\mathbf{y})$ is valid. For this reason, programs with successful causality analysis, which are called *constructive programs*, form a strict subset of programs that have a unique behavior (these are called *logically correct programs* in [78]).

Malik [195] was the first who presented algorithms for eliminating cycles in Boolean equation systems $\mathbf{y} = f(\mathbf{x},\mathbf{y})$. He used a ternary extension of Boolean algebra as introduced by Yoeli and Rinon [252] and Eichelberger [126], and further refined by Brzozowski, Bryant, and Seger [87, 88, 89, 90, 91] to analyze the propagation of signal values in these circuits. The computation of a solution $\mathbf{y}$ depending on the inputs $\mathbf{x}$ is then reduced to the computation of a fixpoint of the function $f_{\mathbf{x}}(\mathbf{y}) := \mathbf{f}(\mathbf{x},\mathbf{y})$, where the inputs $\mathbf{x}$ are fixed. The existence of such fixpoints in the

[1] Actually, the precise complexity class is not known. However, the problem is at least in co-NP and at most in PSPACE.

ternary domain is guaranteed by the Tarski-Knaster theorem [179, 246] (see also the next section), and the number of iterations is limited by the number of equations $|\mathbf{y}|$. Having computed the fixpoint $\mathbf{y} = f_{\mathbf{x}}(\mathbf{y})$ in a symbolical form, i.e., depending on the inputs $\mathbf{x}$, one has an equivalent acyclic equation system $\mathbf{y} = \mathbf{f}'(\mathbf{x})$. As this is done in a three-valued setting, it finally remains to check if all equations evaluate to Boolean values. It can be shown that computing the ternary fixpoint and checking whether it is a Boolean one is co-NP-complete [194, 236].

Although the acyclic version may require more operations than the original cyclic one [160, 170, 216, 217, 218], the elimination of cycles is a popular way for generating single-threaded sequential code from multi-threaded synchronous programs in that a causal order (i.e. a schedule [120]) to evaluate the right hand sides of the equation system is determined[2].

Malik's approach has been generalized by Shiple et al. to sequential circuits with cyclic output functions [235, 236, 237]. Their analysis consists of two phases: In the first phase, Malik's procedure is used to transform the data flow $\mathbf{y} = \mathbf{\Phi}(\mathbf{x}, \boldsymbol{\ell}, \mathbf{y})$ into an equivalent acyclic version $\mathbf{y} = \mathbf{\Psi}(\mathbf{x}, \boldsymbol{\ell})$. Note that this is always possible in the ternary domain, but not always in the Boolean domain. Hence, it may be the case that the right hand side evaluates to a non-Boolean value. For this reason, the second phase of Shiple's analysis consists of checking whether a non-Boolean value can appear for one of the reachable states.

Besides a complete fixpoint computation, heuristics can be applied in a first instance to solve simple cases more efficiently [234]. Alternatives to the fixpoint computation were also considered: [102, 200] replaced the causality problem by the theoretically more difficult satisfiability problem and proposed new SAT solving techniques and temporal induction for its solution. However, causality analysis based on fixpoint computation is not only a heuristic for satisfiability checking, it moreover establishes a direct relationship between the causality of a program and its dynamic execution (stabilization of signals in circuits or existence of dynamic schedules in software). Hence, causality analysis may be viewed as a symbolic compile-time simulation of the program in order to guarantee its conflict-free execution.

3.1.2 Related Problems

Besides ternary simulation, there are many other problems that are equivalent to causality analysis and can be reduced to each other in polynomial time [78]:

- *Causality Analysis of Synchronous Programs* [78] Given a synchronous program, check without speculative reasoning whether there are unique outputs for all inputs.
- *Stabilization of Hardware Circuits with Combinational Feedback Loops* [91, 236] Given a combinational hardware circuit, check whether all outputs stabilize for all inputs after some finite time, independently of the delays of theused gates. Shiple [236] proved the equivalence to Brzozowski and Seger's timing analysis in the up-bounded inertial delay model [91]: Circuits derived from cyclic equation systems will stabilize for arbitrary gate delays iff the equation systems are causally correct. The algorithms used to analyze hardware circuits are basedon a ternary interpretation of Boolean logic.
- *Evaluation of Formulas in Intuitionistic Logic* [78, 196, 197, 237] Given a propositional formula, check whether incomplete truth assignments can be consistently completed by proof rules of intuitionistic logic. Berry [78] pointed out that causality analysis is equivalent to theorem proving in intuitionistic (constructive) propositional logic, since intuitionistic logic may be viewed in a ternary setting (1:provable, 0:disprovable, $\perp$: neither provable nor disprovable). Hence, he introduced the notion of constructive circuits [79, 196, 197, 237].

[2] In the meantime, alternative compilation techniques [104, 118, 193] were proposed. However, these approaches still need causality analysis to guarantee the existence of a dynamic schedule for mutually dependent actions.

- *Type Checking* [159] Check for a given functional program with certain data types whether it is correctly typed. By Berry's observation, the problem is equivalent to type-checking of certain functional programs due to the Curry-Howard isomorphism [159].
- *Existence of Dynamic Schedules* [78, 120] Given a set of mutually dependent guarded actions, check whether there is a dynamic schedule to execute these actions without deadlocks in all possible cases. Edwards reformulates the problem in that the existence of dynamic schedules must be guaranteed for the execution of mutually dependent actions [120].

Hence, causality analysis is a fundamental algorithm that has already found many applications in computer science.

3.2 Contribution

3.2.1 Causality Analysis for Quartz [34, 35]

In [35] we give the first causality analysis for Quartz programs. The added value of the publication is the extension of Malik's and Shiple's approaches to a causality analysis of systems with sequential data flow. This is necessary to compile synchronous programs with delayed actions. We show that similar to Shiple's extension, we can still rely on the fixpoint iteration used in [195]. However, we have to employ two fixpoint computations: one for the initialization part, and another one for the transition part. Similar to previous work, the fixpoints are computed in a lattice that extends the Boolean values by further elements $\perp$ and $\top$, so that their existence is guaranteed. For this reason, we have to check afterwards whether the fixpoints contain non-Boolean values. If they only contain Boolean values, we have transformed the equation system into an equivalent acyclic one and are therefore able to generate single-threaded code. Otherwise, it may be possible to generate single-threaded code, but an exact analysis is NP-complete, so we conservatively decide to reject the program (this is typical for compilers of synchronous languages).

In contrast to Shiple's equation systems, the data flow is split into initialization and transition parts that may both contain cyclic dependencies. Therefore, we say that the above equation system has a *sequential data flow*. In the sequel, we focus on such equation systems and describe an algorithm to check causality of such equation systems. In case the causality is given, the equation system is transformed into an equivalent acyclic one that can finally be used forcode generation.

However, [35] (and all other previous publications on causality analysis) still has some drawbacks, which are addressed by [34]. First, previous work only considers events in the data flow and not memorized values in the data flow. Broadcasting of events is the underlying communication mechanism in Esterel, however events are rather untypical for non-reactive systems. For this reason, Esterel additionally provides valued signals, where the value of a signal is memorized (in contrast to its status). However, ternary simulation of hardware circuits [78, 237] only considers stateful control flow. Some subtle issues, e. g. reincarnation of local variables [35], prevent a straightforward generalization of existing approaches to memorized variables. In particular, one has to additionally deal with write conflicts.

Second, higher data structures such as bitvectors and numbers are only supported by a decomposition to Booleans. While this approach is theoretically elegant, it has two significant drawbacks: The static analysis of these systems is commonly based on BDDs, which often suffer from the state explosion when examining programs with arithmetic computations. What is more problematic is that the analyzed model at the Boolean level may not correspond with the original program with higher data types: For instance, it may the case that the analysis at the Boolean level may be able to identify single bits of an integer variable so that the causality may be proved.

However, it may be the case that this analysis cannot be lifted to the used higher data types, since that abstraction level is coarser.

In [34], we therefore propose a different approach to analyze the causality of a synchronous program. In contrast to traditional compilation schemes, we do not compile to a macro step model of the system, but to a refined one that reflects the reactive behavior at the micro step level of execution. The information flow and causal dependencies are explicitly modeled so that they can be analyzed by general-purpose model checkers. Our method also deals both with events and memorized variables so that it does not suffer from the first of the above mentioned problems. Moreover, we treat atomic data types as unique values, so that we also avoid the second problem. Moreover, we show that the causality analysis can be even reduced to a satisfiability problem, since we can easily determine upper bounds for the fixpoint iterations of a given program at compile time.

3.2.2 Verification of Causality Analysis [19]

While previous work gave paper-and-pencil proofs for the equivalence of the causality analysis based on can-must analysis and the symbolic ternary simulation, there is no previous publication on a symbolic analysis for non-Boolean or non-event variables. In [19], we present such a symbolic analysis as a generalization of ternary simulation. To this end, we have to take into account the problem of write conflicts which does not appear for Boolean events (since disjunction is used as an implicit conflict resolution function). In addition, we prove the equivalence between the can-must causality analysis and our symbolic formulation based on extending each data type by the constants $\perp$ and $\top$.

In [19], we present the formalisation of the traditional causality analysis for synchronous systems as well as the definition of a fully symbolic version of a general causality analysis. We use the HOL4 theorem prover to prove the equivalence of these two variants of the causality analysis, so that we can guarantee that what is checked by means of model checking in a compiler with the symbolic analysis exactly matches the definition of causality analysis as given by the can-must analysis. Moreover, we extend the classical analysis by considering run-time errors like write conflicts, division by zero, or access to array elements outside the declared range. This is accomplished by generalizing the traditional three-valued analysis to a four-valued setting using a constant $\top$ for *run-time error* in addition to $\perp$ *yet unknown*.

3.2.3 Maximal Causality Analysis [38, 39]

All known procedures for causality analysis that are based on fixpoint computation require an extension of Boolean functions to a ternary domain. For such an extension, there are several possibilities. Depending on the chosen embedding, the resulting fixpoint may be different.

In previous work, the effect of different extensions is not considered. Instead, only basic Boolean operations like negation, conjunction, and disjunction are directly extended, and ternary extensions of other Boolean functions are obtained by composition of the ternary extensions of the basic functions. In [35], however, it has already been remarked that this is not optimal, and that different ternary extensions yield different results in causality analysis.

Thus, causality seems to be a *structural property:* Whether a program is causally correct or not depends on its syntax, and not only on its semantics. In the same way, there are logically equivalent hardware circuits where one has stable signal wires, while the other one may oscillate for some inputs and gate delays. Since causality depends on the syntax of a program, one may ask whether there are reasonable program transformations that turn non-constructive programs to equivalent constructive ones. Another way to achieve this is to directly integrate the mentioned transformations in the code generation.

In [38], we reconsider Boussinot proposals [85] to improve causality analysis. The problem with these proposals is that they are only given at the level of causality analysis and therefore might destroy the constructivity, which is obviously not desired. The main contribution of that paper is to *show how Boussinot's improvements can be integrated in the code generation so that constructivity is maintained.* This means that we can modify the intermediate code generation in such a way, that the usual causality analysis of this modified code becomes exactly the causality analysis that Boussinot proposed. The advantage of our approach is clear: we maintain the constructivity but are nevertheless able to analyze and to compile a broader class of programs. This suggests to have several *degrees of causality,* instead of only declaring a program to be constructive or not.

This also raises the question whether there is an optimal way to construct ternary extensions so that causality analysis can resolve as many causality cycles as possible. In [39], we answer this question to the positive: for every Boolean equation system, there is a (uniquely determined) maximal ternary extension that allows the transformation to an acyclic system, if this can be done by fixpoint computation at all. We show that this maximal ternary extension corresponds to the disjunction of all prime implicants, which gives a relationship to hazard elimination [126]. By this relationship, we derive a first algorithm for computing the maximal ternary extension, which, however, requires to compute all prime implicants of a Boolean function. We then present more efficient algorithms for computing the maximal ternary extension. In particular, we present an algorithm that can be easily implemented by means of binary ordered decision diagrams (BDDs).

4

Optimization

4.1 Context

Research over the last two decades has tackled various problems about the compilation of synchronous languages. As discussed in Chapter 2, various compilers have been developed [23, 37, 81, 103, 104, 116, 116, 121, 209] based on different code generation schemes like automaton-based code, (Boolean) equation systems and concurrent control/data-flow graphs. While the first articles on compilation of synchronous programs focused on the *correctness* of the compilers considering mainly semantic issues like schizophrenia and causality problems, this chapter now considers the *efficiency* of the generated code.

4.1.1 Data-Flow Analysis

Data-flow analysis has been a static analysis tool for code optimization in classical compiler design for decades. Early work has been presented in [60, 152, 153, 221] and considers different kinds of data-flow analyses like the computation of live variables, busy variables, available expressions (to detect shared expressions), use-def chains and many more. However, these analyses cannot be directly applied to synchronous programs due to the different underlying model of computation and its different definition of equivalent computations. In particular, changes of the variable's values are synchronously done at the level of macro steps.

Hence, even though static data-flow analysis is a well-established tool for classic compiler optimization, it has not yet been used for the compilers of synchronous languages. The static analysis described in [242] is used to compute conditions for instantaneous execution of a synchronous program that is done similarly by the control flow predicates in [227]. In this thesis, we instead develop the first static data-flow analysis based on fixpoint computations similar to the classic data-flow analyses. However, our data-flow analysis takes care of the synchronous model of computation and aims at the identification of passive code as explained above. The results of this work can be used to optimize the code that is finally generated, e.g., to reduce the reaction time of programs, or to reduce the energy consumption since less computations are performed in the macro steps.

Hence, even though static data-flow analysis is a well-established tool for classic code optimization, it has not yet been widely used for the compilers of synchronous languages. The static analysis described in [242] is used to compute conditions for instantaneous execution of a synchronous program that is done similarly by the control flow predicates in [227]. In [21], we developed a static data-flow analysis based on fixpoint computations similar to the classic data-flow analyses, which will be the basis for one stage of the optimization technique presented in this thesis.

4.1.2 Satisfiability Modulo Theories

The dependency analysis and corresponding optimization techniques described in this thesis are based on *Satisfiability Modulo Theories* (SMT). The concept of SMT is to check the satisfiability of logical formulas over one or more theories. While a SAT solver computes an answer to the question *"Is there an assignment to the variables of a propositional formula such that the formula evaluates to true?"*, an SMT solver is more powerful. It can answer for a formula over first-order predicates of a particular decidable theory (or a combination of such theories) whether there exists an assignment to the variables such that the whole formula evaluates to *true*. Such decidable theories (commonly called *background theories* in the context of SMT) are for example integer linear arithmetic, real linear arithmetic, theories of bit-vectors or arrays [174].

As a result, an SMT solver is capable of more powerful logics than a SAT solver. For example, deciding the satisfiability of the formula $(x > 3) \wedge (x < 0)$ where x is an integer, a SAT solver has to abstract each of the inequations by a Boolean variable and derives the formula $(a \wedge b)$. This formula is obviously satisfiable, as assigning a and b both to *true* makes the whole formula evaluate to *true*. However, for an SMT solver, it can capture the semantics of the relations $>$ and $<$ and interpret 3, 0 and x in the domain of integers. A decision procedure for the corresponding background theory will then deliver the appropriate answer that both clauses of the formula above cannot be fulfilled at the same time. Hence the SMT solver returns the result *invalid*, which means that there is no assignment to the free variables of a formula that makes it satisfiable. Other answers an SMT solver usually gives are *valid*, which means that the formula is always *true*, whatever values are chosen for the free variables, while *satisfiable* indicates that the truth of the formula depends on the assignment of the free variables. If the background theory is undecidable and if the (thereby incomplete) decision procedure of the background theory fails, it can also return *unknown*.

4.2 Contribution

The execution of the guarded actions obeys the synchronous model of computation: As already explained in the introduction, each variable has a unique value in each macro step. The set of possible values of a variable is determined by its type, and its current value in a macro step is determined either by the immediate assignments executed in the current macro step, the delayed assignments executed in the previous macro step, or the default reaction as explained in the next subsection. In each macro step i, the system reads inputs and computes outputs, in principle, by evaluating *all* guarded actions. In general, a system will also have internal variables whose values are also determined by guarded actions in the same way as done for output variables, i.e., either by immediate assignments enabled in the current macro step, by delayed actions enabled in the previous macro step, or by the default reaction if no action determines the value. Internal variables may stem from locally declared variables or from control flow locations. Furthermore, internal variables are generated by the compiler in order to abbreviate common sub-expressions to avoid their re-computations.

As already outlined in previous chapters, a synchronous program cannot be executed like a usual sequential program due to the paradigm of perfect synchrony: all enabled actions must be concurrently executed according to their data dependencies. Clearly, only a subset of the guarded actions of the program are enabled within a macro step. Efficiently identifying this part of the program can significantly improve the run-time performance, which was one of the main motivations for code generations schemes based on the discrete event model [103, 104] or concurrent control/data-flow graphs [209, 255].

Obviously, guarded actions which are not enabled in the current macro step (their guards evaluate to false) do not contribute to the final result. In addition to disabled guarded actions, there is still more room for optimization: the paradigm of perfect synchrony demands that unique values have to be determined for *all* variables (in particular also for *all* local variables) in each step. However, some local values may not be needed to compute the final outputs. The key to a simple code optimization is therefore to determine actions that do not contribute to the final result, which will be called *passive code* in the following.

The notion of passive code is also known for sequential programs, e.g. for code whose execution is not required to compute the return value of a function. Note that *dead code* is different to passive code, since dead code is not reachable, i.e., not executed in any run of the program. Hence, dead code can be safely removed without modifying the semantics of the program. In contrast, passive code is reachable, and may be required in certain macro steps, but may not contribute to the output values in other macro steps. Hence, passive code must not be removed, but its execution may sometimes be suppressed without modifying the overall behavior of the program. Clearly, we can suppress its execution by an appropriate modification of the guards.

The optimizations that we consider in the following have a specific realization (hardware or software) or target architecture in mind, nor do they consider a particular source language. Instead, we aim at optimizing an intermediate code that is based on synchronous guarded actions, the target-independent intermediate format used in the compilation procedure of Chapter 2. This also gives us the possibility to apply our analysis to several source languages and makes it also independent of the target architecture. We implemented all procedures within our Averest system.

4.2.1 Definition of Passive Code [21]

The code optimization we consider in this chapter is based on the identification of *passive code*, which is the key to various optimizations in the course of the further code generation. The contribution of [21] consists of the definition and implementation of a static data-flow analysis that determines the passive code in synchronous systems. To this end, we introduce for each variable x a new variable req_x, which should hold iff the value of x is required for the computations of the current or future macro steps. These expressions are called required *conditions* in the following.

The *required* conditions obviously depend on the externally visible behavior of the system. Usually, all outputs of the system should be computed in all steps. Hence, we set $\mathit{req}_x = \mathsf{true}$ for all output variables $x \in \mathcal{V}_O$[1]. In contrast, all other variables of the system $x \in \mathcal{V} \setminus \mathcal{V}_O$ are generally not interesting as long as they do not carry information that is later used to compute the outputs. Hence, ideally their *required* conditions should be as strong as possible to gain as much as possible space for optimization.

Thus, the main task of our data-flow analysis is to compute program expressions for the *required* conditions req_x for local variables x, which preserve the behavior of the original system. Thereby, our main idea is to follow the dependencies in reverse direction and to construct the conditions for req_x. Unfortunately, this naive computation is not possible, since the conditions req_x mutually depend on each other, and since pseudo-cycles have to be excluded:

First, there are data dependencies across macro steps, which result from delayed actions $\langle\gamma \Rightarrow \mathtt{next}(x) = \tau\rangle$. Hence, if such a variable x is required in step i, all variables in γ are required in the preceding step, as well as all variables in τ if γ holds in the preceding step. As an example, consider the variable wa in Figure 8: it is set by a delayed action (dashed edge), which is always the case for control-flow variables of synchronous programs.

Second, additional data dependencies across macro steps which result from the default reaction of memorized variables impose further problems: The required value of a local memorized variable might be computed several steps before it is actually used in the computation of an output. Since

[1] These definitions of the *required* variables of the outputs can be weakened if the system is accompanied by a specification that implies that not all outputs are needed in all steps.

```
module ABRO
(event bool ?a,?b,?r,!o) {
  loop
    abort {
        wa: await(a);
      ||
        wb: await(b);
      emit(o);
      wr: await(r);
    } when(r);
}
```

$$\left[\begin{array}{rcl}
\mathsf{true} & \Rightarrow & t_1 = wa \wedge wb \wedge a \wedge b \\
\mathsf{true} & \Rightarrow & t_2 = \neg wa \wedge wb \wedge b \\
\mathsf{true} & \Rightarrow & t_3 = \neg wb \wedge wa \wedge a \\
(t_1 \vee t_2 \vee t_3) \wedge \neg r & \Rightarrow & o = \mathsf{true} \\
\mathrm{start} \vee r \vee (wa \wedge \neg a) & \Rightarrow & \mathtt{next}(wa) = \mathsf{true} \\
\mathrm{start} \vee r \vee (wb \wedge \neg b) & \Rightarrow & \mathtt{next}(wb) = \mathsf{true} \\
t_1 \vee t_2 \vee t_3 \vee (wr \wedge \neg r) & \Rightarrow & \mathtt{next}(wr) = \mathsf{true}
\end{array}\right.$$

Fig. 8. Quartz program and its synchronous guarded actions

these dependencies are not encoded in the ADG, the data-flow analysis must explicitly handle these storage properties.

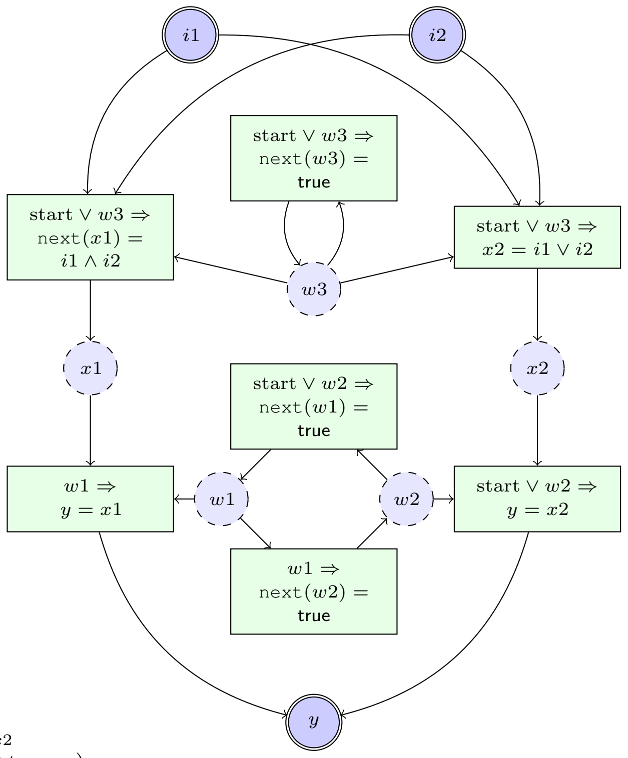
Third, the dependency graph is generally cyclic, even for programs which are commonly referred to as acyclic in the synchronous languages community. For example, variable wa in Figure 8 occurs on the left-hand side and on the right-hand side of the same action, which leads to a pseudo-cycle in the dependency graph. For the execution of the synchronous program, this cycle does not cause any problems, since the new value is fed back only in the following macro step, but the definition of its *required* condition becomes recursive.

In order to determine req_x in the general case, we therefore need the vector μ-calculus. If the dependency is across a macro step, we use the diamond operator of the μ-calculus: $\Diamond \mathit{req}_x$ states that there is a successor state in which x is required. To cope with the cyclic dependencies, we use the fixpoint operator $\mu x.f(x)$, which returns the least fixpoint of the given function f. This is exactly the operation that we need in our computation, since we want to minimize the number of states where req_x for some variable x holds.

The *required* conditions and their approximations can then be used in the subsequent code generation step, which transforms the intermediate code to the target code. Our definitions are the basis for the removal of passive code, which improves the run-time of the software code or the energy efficiency of the hardware circuit. Depending on the realization, the information gained by the data-flow analysis can be used as follows:

- The most obvious optimization is to eliminate passive code by using the *required* conditions to strengthen the guarded actions. Any immediate action that writes x can be strengthened by req_x, and any delayed action that writes $\mathtt{next}(x)$ can be strengthened by $\Diamond \mathit{req}_x$ (which has to be replaced by a program expression by solving the fixpoint equation system). This may speed up the execution time of the system, but a potential drawback is the additional complexity of the guards due to the additional effort for evaluation of the *required* conditions at runtime: One has to carefully balance the additional effort against the savings.
- This directly leads to the second optimization: If the evaluation of guards is very complex, one can weaken them by allowing the action to fire even if a variable is not needed. Any guard γ' that is implied by $\gamma \wedge \mathit{req}_x$ and still implies γ leads to a correct behavior.
- The elimination of passive code can be used to optimize the resource requirements of a given system. Generally, a synchronous system needs dedicated resources for all actions that can execute within the same macro step, e.g. if three guarded actions can be potentially run in parallel, and each one of them needs a multiplier, three multipliers must be statically allocated

```
module Interleaved
(bool ?i1,?i2,!y) {
  event x1, x2;
  loop {
    y = x2;
    w1: pause;
    y = x1;
    w2: pause;
  }
  ||
  loop {
    next(x1) = i1 & i2;
    x2 = i1 | i2;
    w3 : pause;
  }
}
```



$i1$
$i2$
start $\vee$ $w3 \Rightarrow$ next($w3$) $=$ true
start $\vee$ $w3 \Rightarrow$ next($x1$) $=$ $i1 \wedge i2$
start $\vee$ $w3 \Rightarrow$ $x2 = i1 \vee i2$
$w3$
$x1$
$x2$
start $\vee$ $w2 \Rightarrow$ next($w1$) $=$ true
$w1 \Rightarrow$ $y = x1$
$w1$
$w2$
start $\vee$ $w2 \Rightarrow$ $y = x2$
$w1 \Rightarrow$ next($w2$) $=$ true
y

$$\left[\begin{array}{ll}
req_y & \overset{\mu}{=} \text{true} \\
req_{x1} & \overset{\mu}{=} w1 \wedge req_y \\
req_{x2} & \overset{\mu}{=} (\text{start} \vee w2) \wedge req_y \\
req_{w1} & \overset{\mu}{=} \Diamond req_{w2} \vee req_y \\
req_{w2} & \overset{\mu}{=} \Diamond req_{w1} \vee req_y \\
req_{w3} & \overset{\mu}{=} \Diamond req_{w3} \vee \Diamond req_{x1} \vee req_{x2} \\
req_{i1} & \overset{\mu}{=} (\text{start} \vee w3) \wedge (\Diamond req_{x1} \vee req_{x2}) \\
req_{i2} & \overset{\mu}{=} (\text{start} \vee w3) \wedge (\Diamond req_{x1} \vee req_{x2})
\end{array}\right.$$

Fig. 9. Source program, data dependency graph and equation system for requirement conditions

for the hardware realization in principle. If the strengthened guards exclude potential conflicts, the common resource can be safely shared.

- Just as in traditional liveness analysis, the condition req_x can be exploited to determine an optimal register allocation for the variables. Two variables x_1 and x_2 can be mapped to the same register, if they are never required at the same time, which can be simply checked by $\neg(req_{x_1} \wedge req_{x_2})$. Apparently, the formula covers the special case to map two variables of distinct scopes to the same register. This optimization is feasible for both hardware and software realizations.

The example in Figure 9 illustrates the computation of the *required* conditions. The compiler extracts from the source program given in the upper left corner the guarded actions which are shown in the dependency graph on the right-hand side. Then, the guarded actions are used to determine an equation system that defines the fixpoint to compute for the solution of the *required* variables as shown of the left-hand side.

4.2.2 Semi-Symbolic Analysis [1]

The approach of the previous section is a general approach to find redundant computations in a synchronous system. However, it turned our that it has two significant drawbacks: first, the runtime of the analysis as presented in [21] (Section 4.2.1) may exceed an acceptable amount for realistic examples, while the conservative approximations given in [21] may be too coarse. Second, even after a successful computation of the *required* conditions, an aggressive strategy which adds these conditions to all guards can worsen the performance of the system significantly, and the tradeoff between savings and additional effort of strengthened guards is generally very difficult to analyze.

Therefore, [1] introduces an alternative approach, which is a conservative approximation of the data-flow analysis of [21]. It exploits structural information of the synchronous systems by first generating an extended finite state machine (EFSM) from the guarded actions of the synchronous program, which are also the best starting point for the generation of *fast* sequential code from synchronous programs.

EFSMs explicitly represent the state space of the control-flow: each state s represents a subset $\mathsf{Labels}(s) \subseteq \mathcal{L}$ of the control-flow labels, and edges between states are labelled with conditions that must be fulfilled to move from the source state to the target state. Thus, EFSMs are a representation where the control-flow of a program state is explicitly represented, while the data flow is still represented symbolically (while synchronous guarded actions represent control flow and data flow symbolically). Note that, in order to obtain efficient code, compilers also determine a control-flow state for data-flow languages (i. e. Lustre or Opal) by converting a subset of the Boolean variables [145].

4.2.3 SMT-Based Optimization [2]

This builds upon two pillars: (1) For modeling and verification of the embedded system, we use synchronous programs that have many advantages, in particular, a formally defined semantics with a deterministic concurrency, which is a necessary requirement for verification and static analysis. Moreover, the absence of dynamic data structures allows a much better static analysis of the runtime requirements. Thus, synchronous programs provide a good basis for our desired combination of verification and optimization. (2) For the verification part, we use a SMT (satisfiability-modulo-theory) prover [64, 109] as a backend. SMT solvers can efficiently check the satisfiability of first-order formulas over so-called background theories.

The contribution of [2] consists in an optimization technique that employs verification for the optimization of synchronous languages. This is not only done for verification of a performed optimization step. In addition, we also generate verification goals during the compilation and optimization process, and hand them over to a tightly coupled SMT-solver. The compilation and optimization process continues according to the decisions made by the SMT solver. We will demonstrate that we can this way achieve a significant improvement of the run-time performance of the generated code. Our code optimization is not based on a particular source language. Instead, it is based on a general intermediate code (synchronous guarded actions) so that our optimization techniques can be applied to several source languages and are also independent of the later on chosen target architecture.

4.2.4 Integration of Contributions [24]

This publications integrates all the publications of this section. Additionally, it gives a formalization of passive code and the analysis so that the correctness of the presented optimizations can be verified. Furthermore, it presents an optimized semi-symbolic analysis, which integrates the computation of passive code with previous code generation scheme and thereby significantly improves the practicability of our approach. The proposed approach is evaluated with the help of several practical examples. Our F# implementation is based on our Averest framework, and we use the SMT solver Z3 [198], which was directly connected via its *.NET* managed API.

5

Synthesis

5.1 Context

In the area of synchronous languages, research over the last two decades has put its focus on creating sequential code from the synchronous languages. Various compilers have been developed (see Section 2.1.3): except for a translation to hardware circuits, they all target *deterministic single-threaded code* in order to directly execute synchronous programs on simple micro-controllers without using complex operating systems. Hence, generation of multi-threaded code is still an open problem, which has not yet been considered in this area with only a few exceptions.

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.

Thus, in addition to usual code generation, the synthesis of synchronous programs involves the following 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.

The following sections review some related work in this context.

5.1.1 GALS Systems

As the name suggests, globally asynchronous locally synchronous (GALS) [99, 136] systems consist of components, which follow internally the synchronous model of computation [74, 142], but that work together in an asynchronous manner. The execution of a synchronous system consists of a sequence of reactions $\mathcal{R} = \langle R_0, R_1, \ldots \rangle$ where all inputs are read and and all outputs are immediately written by the synchronous system. Thus, it basically implements a function mapping input tuples to output values depending on some local state. In contrast, communication between components is asynchronous: there is no shared memory, and information is only exchanged with the help of FIFO buffers. Thereby, the reactions of the individual components are decoupled, each one triggering its own computation according to the availability of suitable input values.

GALS systems can be seen as a special case of dataflow process networks (DPN), which we will use to describe our examples. The behavior of a DPN is usually described with the help of *firing rules*, which are a simple operational description of a single local component. In each step, the component basically matches the current content of the input buffers to the patterns given in the left half of the table. Among the rows with matching input patterns, an arbitrary one is selected, the matched input values are removed from the input buffers, and output values are written to the output buffers according to the right half of this row.

In the case of GALS systems, the patterns on the left-hand side and the output list always contain at most one token, since fully synchronous components are only allowed to read and write at most one value for each signal in each reaction. For technical reasons, we add to the set of data values $\mathcal{D}$ a special symbol $\boxdot$, which is used if no value is read or written in a particular step. This will simplify the presentation in the following since we achieve this way that always exactly one value is consumed and produced for each input and output signal even though the realized system may not read or write the values denoted as $\boxdot$. Obviously, this special value $\boxdot$ will never be *never transmitted* in the implementation of the GALS system: matching $\boxdot$ on the left-hand side means to read nothing from the corresponding input channel, and a $\boxdot$ in the output list means that no value is produced for the corresponding output channel.

5.1.2 Endochrony and Isochrony

Encoding and transmitting empty messages ($\boxdot$) is not desired since it involves a large overhead for implementations. Instead, as already mentioned above, these tokens representing the absence of values are mapped to absence of transmission, i. e. we do not want to send and receive the $\boxdot$ tokens. However, in the following formalizations, we keep them to study the effect of delays on the asynchronous communication channels. Obviously, as these $\boxdot$ values align the streams in a particular way, it is an interesting question, which systems still show the same behavior if we introduce additional $\boxdot$ values or remove them completely.

This fundamental problem of desynchronization has been addressed in previous work [67, 71, 72, 73, 206, 207, 208], and the notions of endochrony and isochrony are intended to solve this problem. Before we continue with a discussion of these concepts, we first make some auxiliary definition over streams, which we need to define these concepts.

Stretch and Flow Equivalence

In a GALS systems, local components generally run at different speeds, and some components might have to wait for inputs to perform a reaction step. With the help of the $\boxdot$ tokens, this can be modeled by so-called silent reactions, i. e. reaction steps produced by firing rules where each pattern has the form $(\boxdot :: L)$ and each output is $\boxdot$. Thus, when a silent firing rule is fired, neither inputs are consumed nor outputs are produced. As these silent reactions do not contribute to the real behavior, we often want to omit them.

Let $x = \mathsf{Bhv}(P)$ be a behavior of process P, i. e. x is a tuple of streams $x \in \mathsf{Stream}^m$ for all variables in P. Then, the behavior $\mathsf{stfree}(x)$ is the one where all silent reactions of s have been removed. Furthermore, we say that x and y (both in Stream^m) are *stretch-equivalent* ($x =_{st} y$) if and only if $\mathsf{stfree}(x) = \mathsf{stfree}(y)$. For example, $(x_1, x_2, x_3) =_{st} (y_1, y_2, y_3)$:

x_1	0	2	$\boxdot$	6	$\boxdot$	10	...
x_2	1	$\boxdot$	$\boxdot$	3	$\boxdot$	2	...
x_3	0	2	$\boxdot$	8	$\boxdot$	7	...
y_1	0	$\boxdot$	2	6	$\boxdot$	10	...
y_2	1	$\boxdot$	$\boxdot$	3	$\boxdot$	2	...
y_3	0	$\boxdot$	2	8	$\boxdot$	7	...

As already stated above, we want that components of the GALS system only consume and produce real values (of domain $\mathcal{D}$), and that $\boxdot$ is never transmitted. To this end, we define for a stream $x \in \mathsf{Stream}^m$ another stream $\mathsf{flows}(x)$ where all $\boxdot$ have been removed from x (i. e. the stream that is visible in the actual implementation). Furthermore, we say that x and y are flow-equivalent ($x =_{fl} y$) if and only if $\mathsf{flows}(x) = \mathsf{flows}(y)$. For example, $\mathsf{flows}(x_1, x_2, x_3) = \mathsf{flows}(y_1, y_2, y_3) = (z_1, z_2, z_3)$ is:

z_1	0	2	6	10	...
z_2	1	3	2	...	
z_3	0	2	8	7	...

Obviously, stretch equivalence implies flow equivalence, i. e. $x =_{st} y \rightarrow x =_{fl} y$, whereas the other direction does not hold in general.

Endochrony

Now assume that the components work synchronously (i. e. still use the $\boxdot$ tokens) and that the communication is asynchronous (i. e. the $\boxdot$ values are not transmitted). In order to perform a synchronous reaction step correctly, each component must be able to reconstruct the inputs of that step from the asynchronous communication, i. e. it must be able to reinsert the deleted $\boxdot$ values. Endochronous components have this property: they reconstruct reaction steps (up to stretch-equivalence) in a unique way from the asynchronous input streams.

Following the formal definition of [241], we say that a synchronous component P is *endochronous* if and only if for any behaviors $\rho_1, \rho_2 \in \mathsf{Bhv}(P)$

$$\rho_1 {=_{fl}}^{in} \rho_2 \Rightarrow \rho_1 =_{st} \rho_2$$

where $\rho_1 {=_{fl}}^{in} \rho_2$ is an abbreviation of $\rho_1|_{\mathcal{V}_{in}} =_{fl} \rho_2|_{\mathcal{V}_{in}}$ and $\rho|_{\mathcal{V}_{in}}$ is the projection of ρ to the input streams $\mathcal{V}_{in}$ of P. In words, the component P is endochronous if two flow-equivalent inputs lead to stretch-equivalent behaviors of P.

In order to understand this definition, consider a variant of Berry's Gustave function as given in Figure 10. This component is endochronous, i. e. flow-equivalence of inputs implies the stretch-equivalence of process behaviors. It reconstructs the alignment of streams by looking at all present values at the inputs (without removing them), and then chooses the appropriate firing rule. For example, if we have the input streams $((1 :: A), (0 :: B), (0 :: C))$, the process glimpses all the three inputs streams to finally see that the first firing rule will be chosen. For the next synchronous reaction, the component takes the 1 for x_1, the 0 for x_0, and introduces a $\boxdot$ for x_3.

x_1	x_2	x_3	y
$(1 :: A)$	$(0 :: B)$	$(\boxdot :: C)$	$[1]$
$(\boxdot :: A)$	$(1 :: B)$	$(0 :: C)$	$[1]$
$(0 :: A)$	$(\boxdot :: B)$	$(1 :: C)$	$[1]$
$(0 :: A)$	$(0 :: B)$	$(0 :: C)$	$[\boxdot]$
$(1 :: A)$	$(1 :: B)$	$(1 :: C)$	$[\boxdot]$

Fig. 10. Example: Gustave

If the component is represented by firing rules, as in our case, it is simple to check its endochrony. For any pair of rules, we must check whether they do not overlap, i. e. if v_1 and v_2 are the first values in the patterns of the rules, $v_1 \neq \boxdot$, $v_2 \neq \boxdot$ and $v_1 \neq v_2$. Intuitively, this input can then be used to distinguish the situation where either rule is fired. The requirement that v_1 and v_2 are not $\boxdot$ ensures that we do not fire different rules depending on the delays in the streams: two flow equivalent input behaviors should lead to the same result.

Endochrony is not compositional, i. e. if we merge two endochronous components, the resulting component may not be endochronous. To see this, consider the trivial example cpy1, which only forwards data values.

x		y
$(a :: A)$	$a \neq \boxdot$	$[a]$

Obviously, cpy1 is endochronous. Now, we merge two of these components to a single component cpy2:

$$\begin{cases} y_1 = \mathsf{cpy1}(x_1) \\ y_2 = \mathsf{cpy1}(x_2) \end{cases}$$

This composition destroys endochrony. We see that for cpy2 it is now the case that $\rho_1 =^{in}_{fl} \rho_2$ holds, but we do not have $\rho_1 =_{st} \rho_2$.

ρ_1

x_1	1	2	3	$\boxdot$	5	...
y_1	1	2	3	$\boxdot$	5	...
x_2	8	4	2	$\boxdot$	3	...
y_2	8	4	2	$\boxdot$	3	...

ρ_2

x_1	1	2	3	$\boxdot$	5	...
y_1	1	2	3	$\boxdot$	5	...
x_2	8	$\boxdot$	4	2	3	...
y_2	8	$\boxdot$	4	2	3	...

As a consequence, we are not able to establish a unique synchronous execution for several endochronous components.

Isochrony

As we are generally not able to reconstruct a synchronous reaction for several endochronous components, we need another useful criterion in the context of desynchronization. Essentially, a desynchronized system only needs to ensure that the reconstructed synchronization for a component complies to one of the other components in the network. If several components agree on their reconstructions, they are said to be *isochronous*. Formally, several components $P_1, \ldots, P_n$ are isochronous [241] iff

$$\mathsf{flows}(\mathsf{Bhv}(P_1 \parallel \ldots \parallel P_n)) = \mathsf{flows}(\mathsf{Bhv}(P_1)) \parallel \ldots \parallel \mathsf{flows}(\mathsf{Bhv}(P_n))$$

Note that the composition of flows on the right-hand side is an asynchronous composition, but semantically it is the same as the synchronous composition (as the only difference is that the streams that are composed are already free from $\boxdot$ values). Isochronous compositions guarantee that the asynchronous composition generates a behavior that is flow-equivalent to the one of the synchronous composition. Note that isochrony is not composable, i. e. composing two sets of isochronous components is not necessarily isochronous. This is obvious since a single component is always isochronous by definition so that compositionality would make all systems isochronous.

In example cpy2 (see Section 5.1.2), the composition of both cpy1 components is apparently isochronous. Checking the property in general can be done by model checking [72, 241]. For all reactions of the endochronous components $P_1, \ldots, P_n$, we must check that, whenever they agree on the present shared variables (shared variables that are different to $\boxdot$ in the current reaction), then they also agree on all the shared variables. This guarantees that the receiver inserts $\boxdot$ tokens at exactly the same points of the stream where the sender at the other end of the asynchronous communication channels has discarded them.

OpenMP is an API based on compiler directives for shared-memory parallel programming in C/C++ and Fortran on multiple platforms. In our translation, we make use of some of its directives. First, to create parallel code, we use the `parallel sections` directive, which marks a fork in the current thread and thereby creates a new team. To describe each of its members, we use the `section` directive. Second, to control race conditions between a team of threads, we use the OpenMP directives `shared` and `reduction`. The `shared` directive declares variables to be shared among all threads. In contrast, the `reduction` directive takes two parameters, a variable and an operation, and attributes each thread its own local copy of the variable. Thus, accesses do not require synchronization during the execution of the thread. Only when a thread terminates, it requests exclusive write access to the global variable. Then, it performs the reduction operation on the current value of the global variable and its local copy.

With these directives, generating OpenMP-based code from the task graph can be achieved fully structurally: *Task actions* are translated into blocks in the C-code. If the task action is a simple guarded action, a conditional statement is generated. If the task action is a team, we first generate code for it and insert it at this point. For *task teams*, we first generate code for all its members and then put each of them into a `section` directive of OpenMP, marking it as a separate thread. These parts are concatenated and finally enclosed by a `sections` directive. For all variables, which are shared among two or more members of the team, we need to add a *reduction* clause. *Tasks*, which consist of a list a task actions, are simply translated to a sequence of C statements. The final *task structure* just wraps the master task.

5.1.3 Pipelining

Pipelining is a simple principle for parallel computation that dates back to the late 1950s [172, 213]. It can be applied to all kinds of processes that repeatedly apply a sequence of actions $\alpha_1; \ldots; \alpha_p$ to an incoming stream of objects. Instead of processing single actions for each object one after the other, a pipelined system processes p objects $o_{t+p}, \ldots, o_{t+1}$ at every point of time t in parallel and applies thereby the actions $\alpha_1; \ldots; \alpha_p$ to these objects in parallel (i.e., action α_{i+1} is applied to object o_{t+p-i} at time t). Pipelining is generally used to increase the throughput of the system: Using p pipeline stages, a theoretical speed-up by a factor p can be obtained [176] without increasing the number of actors that perform the actions α_i. Essentially all microprocessors are nowadays implemented with pipelines to speed-up the processing of their instruction streams. However, pipelining is a much more general parallel processing technique and can therefore also be used to create pipelines of software threads. A very good survey on the architecture and analysis of pipelines (which is absolutely worth reading for its historical viewpoint) can be found in [213].

To apply pipelining to a system description, we assume that the considered description consists of a single loop whose loop body $\alpha_1; \ldots; \alpha_p$ can be pipelined as outlined above [213]. Clearly, this is a strong requirement that can, however, be always achieved by appropriate program transformations. As we will explain in the next section, our compiler for synchronous languages is already able to perform this required program transformation which is one of our motivations for choosing a synchronous language as starting point of our model-based design flow. A further argument for using synchronous languages is that we typically find more concurrency in the programs that allows us to generate more powerful pipelines compared to sequential programs. Finally, the precise notion of time given by synchronous languages allows a formal analysis of the correctness of the generated pipelines.

The generation of software pipelines as presented in this thesis should not be confused with *software pipelining* [177]. The latter is a well-known technique to increase the degree of instruction-level parallelism of loops in sequential programs by unrolling the loops and overlapping a couple of loop bodies (if the data dependencies allow this). This approach is frequently applied by compilers targeting VLIW or superscalar processor architectures [177] — especially for VLIW/EPIC processors [131, 132, 133]. The difference between this software pipelining and our approach is that our approach is applied to a level of abstraction much higher than the instruction level considered typically in software pipelining: our pipeline stages are entire threads extracted from a system description. Furthermore, we do not unroll loops to increase the level of concurrency, and instead only partition the given set of synchronous actions into pipeline stages.

5.1.4 OpenMP

OpenMP is an API based on compiler directives for shared-memory parallel programming in C/C++ and Fortran on multiple platforms. In our translation, we make use of some of its directives. First, to create parallel code, we use the `parallel sections` directive, which marks a fork in the current thread and thereby creates a new team. To describe each of its members, we use the `section` directive. Second, to control race conditions between a team of threads, we use the OpenMP directives `shared` and `reduction`. The `shared` directive declares variables to be shared among all threads. In contrast, the `reduction` directive takes two parameters, a variable and an operation, and attributes each thread its own local copy of the variable. Thus, accesses do not require synchronization during the execution of the thread. Only when a thread terminates, it requests exclusive write access to the global variable. Then, it performs the reduction operation on the current value of the global variable and its local copy.

With these directives, generating OpenMP-based code from the task graph can be achieved fully structurally: *Task actions* are translated into blocks in the C-code. If the task action is a simple guarded action, a conditional statement is generated. If the task action is a team, we first generate code for it and insert it at this point. For *task teams*, we first generate code for all its members and then put each of them into a `section` directive of OpenMP, marking it as a separate thread. These parts are concatenated and finally enclosed by a `sections` directive. For all variables, which are shared among two or more members of the team, we need to add a *reduction* clause. *Tasks*, which consist of a list a task actions, are simply translated to a sequence of C statements. The final *task structure* just wraps the master task.

5.2 Contribution

5.2.1 Desynchronizing Synchronous Programs [11]

Several researchers have already considered the desynchronization of synchronous programs, as in particular, the pioneering work of Potop et al. [72, 206, 207, 208]. Their definitions of endochrony and isochrony, which are introduced in [72, 207], 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 [206], which is simpler to check, but still sufficient in practice. Finally, [208] 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 [208]. This also transfers the whole theory in a causal context, and sending and receiving of messages is explicitly visible.

In [11], 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. 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. 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.

5.2.2 Generation of Multi-Threaded Code [5, 6, 7]

In [6] we consider the problem of generating multi-threaded C-code from synchronous guarded actions. Our contribution is a translation procedure from purely synchronous guarded actions to multi-threaded OpenMP-based C-code. This is the first approach to translate imperative synchronous programs to multi-threaded code for general-purpose multi-core processors. Thereby, our implementation is still deterministic – for the same inputs we get the same outputs – independent of the scheduling of the individual OpenMP threads. As the synchronization of threads usually requires a significant amount of time, which can easily exceed the time for actual computation, we try to keep the synchronization effort as low as possible in order to get a significant speed-up from multi-threaded code. To this end, our code generation algorithm uses the data dependencies as a starting point for a task structure, which represents a possible thread structure of the final OpenMP-based C-code.

In [7], we consider the partitioning of system descriptions into threads, which are not necessarily given by the original description. Instead, we show how threads can be automatically generated from such a description to form a pipeline (see Section 5.1.3) whose stages are implemented by the identified software threads. In this approach, we construct a data dependency graph of the guarded actions and split it *horizontally* to form independent threads, whereas [6] split it *vertically* into threads. Another important difference to [6] is that the threads generated by the horizontal partitioning of the data-dependency graph communicate via FIFO buffers instead of a shared memory which allows us to run them asynchronously in contrast to the threads in [6] (which implement parts of one reaction step). The approach presented is therefore closer to Sutherland's micro-pipelines [240]. As the generated threads run asynchronously to each other, we moreover change the model of computation: we generate delay insensitive [248] threads from a synchronous program.

In [5], we integrate and extends our the approaches presented in [6] and [7]: [6] extracts independent parts of a synchronous program to extract concurrent threads, whereas [7] slices chains of dependencies to create a pipelined system. 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.

The contribution of [5] is 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.

There is some previous work which has already considered the automatic distribution of synchronous programs to an asynchronous network of processing elements: In [95, 137], 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 [92, 93, 94] and synchronous elastic systems [105, 106, 175] 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.

5.2.3 Translation to Data-Flow Process Networks [8, 9, 10]

In the publications above we showed how independent computations [6] and pipelining [7] can be used to generate multi-threaded programs from synchronous systems. However, our translation immediately created C code based on OpenMP, which imposes tight restrictions on the source code structure. More closer is the approach presented in [5], which partitions a synchronous program into concurrent components. Although the structure of the partitioned program resembles a DPN, its semantics is different so that the nodes – in particular the designated *merge* nodes – cannot be translated to DPN nodes without a heavy encoding. Moreover, we do not only translate synchronous systems to DPNs, but we do also translate the resulting DPNs to a data-flow language, showing the applicability of our approach.

The main contribution of [8] is a translation from synchronous systems to data-flow process networks, thereby bridging the gap between two fundamentally different models of computations. This makes it simpler to map synchronous systems onto distributed target architectures. Furthermore, we propose to use DPNs as an abstract intermediate representation before final synthesis to concrete data-flow languages or multithreaded software. On the one hand, this separates concerns, since the conceptual part, the switch of the model of computation, is clearly separated from the technical part, which deals with the concrete APIs or differences on the target platforms. On the other hand, this also gives us the possibility to reuse this translation as a common step for various architectures.

We show the feasibility of our approach by a translation from DPNs to CAL [128], which is a programming language of the OpenDF package. It provides many useful features, which will be also accessible to our synchronous systems: including the ability to create efficient multithreaded software and VHDL code from CAL programs [83, 166]. Furthermore, the OpenDF package also contains a simulator to investigate the behavior of single nodes and the complete system. A highlight of OpenDF is an approach for HW/SW co-design [220]. In particular, [220] describes how one can create code for heterogenous systems by translating nodes of a system separately to different platforms.

In [10], we study how out-of-order execution, which is a well known technique from computer architecture, can be used in the context of DPNs. As their nodes are executed many times, it might be the case that the next execution must wait due to unresolved dependencies or lack of resources, where the following one has no such problems and could be immediately executed. This out-of-order execution avoids idle time and can speed up the program execution. As data is now reordered, the approach involves additional effort to reorder the correct flow of output data.

Finally, [9] considers the problem to reduce the communication costs in DPNs that deal with arrays. The main idea is thereby that we define for each array of the given program one process node (called the writer of the array) in the DPN that is responsible for all write updates of the array. Each node reading the array will maintain a local copy of the array and will receive from the writer of the array the corresponding updates. While the array may be large, the updates typically contain only a few assignments to array elements instead of the entire array. Depending on the application, the presented approach *automatically* reduces the amount of data that has to be sent through a DPN's communication channels. It is not difficult to see that our proposed algorithm is correct, so that we are able to automatically translate synchronous programs to deterministic DPNs having this optimized use of arrays.

Typical examples of applications that benefit from the mentioned optimizations are all kinds of embedded systems that deal with arrays. In particular, this covers multimedia applications or more general applications that deal with digital signal processing. It is also not hard to see that the approach can be applied to other non-scalar data types as well, but our experimental results are currently limited to arrays, since these are the most important compound data types in embedded applications.

6

Verification

6.1 Context

As already discussed in Chapter 5, the desynchronization of synchronous systems has gained attention [72, 94, 107, 207, 210]. Its basic idea is to use existing methods and tools for synchronous systems throughout the whole development process, in particular, for modeling and verification. A subsequent desynchronization step decomposes the system into several components to implement a GALS system. Thereby, some synchronization logic is added so that the behavior of the synthesized system corresponds to the original model. Previous publications have considered the desynchronization of synchronous programs, as in particular, the pioneering work in [72, 206, 207, 208] (see Section 5.1.2). Their definitions of (weak) endochrony and isochrony are intended to answer the question of which systems can be safely desynchronized without additional effort while *preserving the original behavior with respect to flow-equivalence of each signal*, i. e. each asynchronous channel will still transport the same sequence of tokens. Their main result is that systems composed of endochronous components which are isochronous to each other can be safely desynchronized without additional effort.

In Chapter 5, the effect of desynchronization to synthesis is addressed. In contrast, this chapter shows the effects to verification.

6.1.1 Runtime Verification

Runtime verification [63, 148, 149, 151, 171, 187] is often a good compromise between formal verification and testing. In runtime verification, one adds special components called monitors to a system whose task is to check whether required properties are satisfied during the execution of the system. For most standard temporal logics (at least for safety properties), a property φ can be automatically translated to an equivalent monitor [130, 149, 151, 204]. The resulting component can then either check the system *online*, i. e. the monitor and the system run in parallel (the typical case), or *offline*, where the monitor checks a finite set of previously recorded executions.

A particular problem for runtime verification is that the decisions of monitors can only be made on the finite history of the executions seen so far, while the semantics of temporal logic formulas requires to consider the unknown infinite future as well. For this reason, many *special temporal logics* have been developed [65, 66, 199, 204]. These logics extend the Boolean values true and false by further logic values that denote some further information about the unknown future. All of these logics still assume a synchronous composition with the monitors.

By definition of the semantics of temporal logics, the monitors run synchronously in parallel to the monitored system. For this reason, one is forced to synchronize the monitor and the monitored system so that both subsystems influence each other. Since the monitor often checks

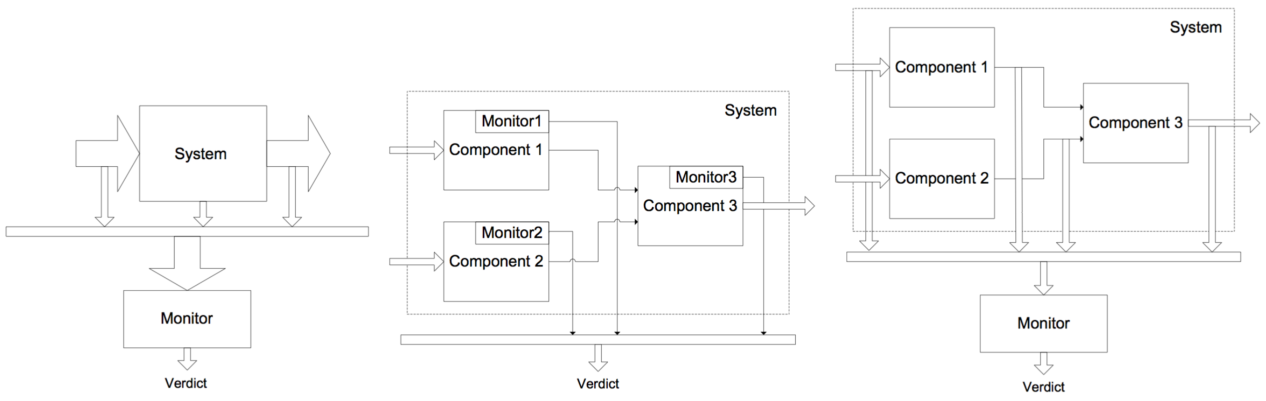


Fig. 11. Variants of monitors: centralized (left), decentralized (middle), and separate (right) monitors

the consistency with a less efficient but more readable specification, this typically slows down the overall performance and therefore worsens non-functional properties of the system. The approach presented later in this chapter removes this interaction even though the considered temporal logics are still based on a synchronous semantics.

The mentioned problem has already been recognized by others who proposed less efficient solutions. In particular, centralized and decentralized implementations of monitors (see Figure 11) have been distinguished. A *centralized monitor* [110, 149, 150] runs as a separate component of the monitored system, as shown on the left-hand side of Figure 11. If a synchronous composition is used, the system inputs and outputs are synchronously transferred to the monitor and form a synchronous state of the entire system. As already mentioned, a synchronous composition may slow down the monitored system as the system might have to wait for the monitor for synchronization. If asynchronous communication is used, then the monitored system has to send besides the values of inputs and outputs additional information (like the presence and absence of values or alternatively their tags) to the monitor to allow it to reconstruct the synchronous computations. Thus, an asynchronous implementation doubles the communication of the data transferred between the monitor and the system. Furthermore, the monolithic view of the monitored system makes it impossible to inspect the system components inside the system.

For monitoring distributed systems, a methodology generating *decentralized monitors*[101, 233] has been proposed. The idea of decentralized monitors is to decouple a global property over the system into local properties that are monitored in single components. As the system components are running asynchronously, tagged messages are used for keeping the communicating processes synchronized on system events. Some dedicated local data structures are used for keeping the executions of components that are relevant to the monitored local properties. Although this kind of monitoring is called decentralized, the monitor is tightly coupled to the monitored system so that there is a potential interference that can again influence non-functional properties: For example, the additional resources required by the monitor might violate the real-time requirements, thereby possibly changing the functional behavior of the system. Moreover, in industrial systems, e. g. in the automotive domain, many parts are supplied by third parties, IP designs, which cannot be modified in general. Hence, neither centralized nor decentralized monitoring can be applied in this context.

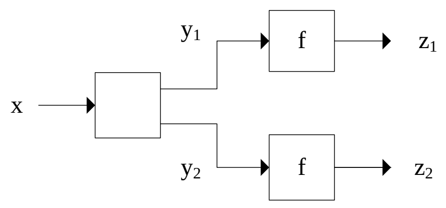


Fig. 12. Simple desynchronization example

6.2 Contribution

6.2.1 Verification vs. Desychronization [10]

In [10], we addressed the problem that sometimes the guarantee that all signal flows are preserved may not be sufficient – in particular, if the original synchronous systems has been proved correct with respect to specifications given in temporal logics. As these properties rely on an alignment of the signal flows, previously verification results are generally not valid any more. To illustrate the problem, consider a very simple example (see Figure 12). Assume that we have a system which consists of three parts: the first one only forwards its inputs to its outputs, whereas the other two parts implement the same function f. Obviously, in this synchronous system the LTL property $\mathsf{G}(z_1 = z_2)$ holds. However, after a desynchronization of the three parts, this is no longer the case since z_1 and z_2 are generally delayed differently. However, there are other global properties, which can be still guaranteed after the desynchronization. Assume that all signals in our example are Boolean, and the function f is a simple negation. Then, the LTL property $[z_1 \mathsf{U} x]$ will still hold in the desynchronized system.

Hence, in contrast to previous work [72, 206, 207, 208], where desynchronization under flow-preservation was explored, we consider a different problem in [10]: we aim at identifying *which LTL properties formally verified for a synchronous system are still preserved by desynchronized implementations*. Obviously, this is a very important question for model-based design which clarifies the compatibility of verification and desynchronization.

Note that our approach is also different to other previous ones [114, 214], where the verification is carried out on the GALS model (desynchronized model in our context). Our approach avoids the analysis on the desynchronized model, which is generally more complex. In contrast, we want to benefit from the synchronous abstraction of time by using the verification results from the synchronous level, and our theory will guarantee that the gained results also hold in the desynchronized implementation.

The main contribution of [10] is a method to check the preservation of LTL properties in desynchronized implementations of a synchronous system. The core consists of a theorem which gives us sufficient conditions to guarantee the preservation of already verified LTL properties. In particular, this enables us to answer two questions: (1) Given a synchronous system S, an already verified LTL property φ and a desynchronization of S, will φ hold in the desynchronization? (2) Given a synchronous system S and an already verified LTL property, will φ be preserved in *any* desynchronized implementation of S?

6.2.2 Runtime Verification vs. Desychronization [3]

Section 6.1.1 already showed the deficiencies of runtime verification in the context of desynchronized systems. In [3], we therefore propose a different approach, which we call separate monitoring, to the runtime verification of reactive systems that were originally based on a synchronous composition, but are then implemented as asynchronous/distributed reactive systems. We create separate components for the monitors (see Figure 11, right hand side) that run asynchronously to the remaining components (that might also run asynchronously to each other). The components of the system are not modified at all so that the real-time behavior of the system and most other non-functional properties are preserved. In real systems, e. g. in an automotive application, the monitor can be implemented on an additional control unit, snooping the vehicle bus for runtime-verification. Obviously, to cope with asynchronous communication, we need to ensure additional system properties so that the monitor always has a consistent view of the system state.

The main contribution of [3] is a methodology for separate monitoring of distributed reactive systems that were obtained by desynchronization of an original synchronous composition of modules that now run asynchronously to each other. Based on the theory of endo/isochronous systems, we define the subclass of synchronous systems that can be separately monitored, and we show how to construct an asynchronous monitor component for a given property. Surprisingly, it turns out that even systems can be separately monitored that are not endo/isochronous.

7

Modeling

7.1 Context

The synchronous model of computation imposes tight restrictions that lead to an unfortunate inflexibly of already created systems: An apparent drawback is the single abstraction layer provided by micro and macro steps, which may lead to a possible *over-synchronization in a synchronous program*. For example, compilers (at the back-end of the language) are challenged when generating efficient code for programs consisting of sporadically communicating threads, since all parts of the program implicitly synchronize after each step, even if there are no data dependencies. While a static clock and data-flow analysis may be able to detect this effect and to desynchronize such programs [11, 21], adding an explicit notion of independence makes it possible for compilers to create desynchronized code without expensive analyses. Additionally, spurious synchronization can be prevented by construction. Similarly, developers (at the front-end of the language) are limited by the single temporal abstraction layer of the synchronous model of computation. It abstracts from the causality and scheduling of the operations within a single macro-step but there is no support for a more coarse-grained structure of logical time. For example, this immediately causes problems if several existing *modules of different abstraction levels* should be combined.

Using a hierarchy of clocks in the system description is an apparent approach to tackle these problems, while still preserving all advantages of the synchronous model of computation. A crucial point in the design of a derived multi-clock model of computation is the construction of the clock hierarchy. In general, one can distinguish three different alternatives for this: (1) new clocks are created independently from each other and subsequently related by *explicit clock constraints* to form a clock hierarchy, (2) new clocks are created by *downsampling* already existing clocks (bottom-up hierarchy), and (3) new clocks are created by *upsampling* already existing clocks (top-down hierarchy). While all alternatives seem to be equivalent at first sight, a closer look reveals that there are significant differences. In particular, the last alternative bears an enormous potential, which has not been used so far due to the lack of full support by state-of-the-art synchronous languages.

In this chapter, we present an extension of the imperative synchronous programming language Quartz which gives developers the possibility to declare so-called *subclocks*. These subclocks are the first full support of a top-down hierarchy of clocks. They are an appropriate means to refine the temporal behavior of synchronous systems to avoid over-synchronization and to allow one to exchange components with others having a different internal temporal behavior. Hence, clock refinements provide additional degrees of freedom for synthesis and design space exploration. At the same time, we preserve all desired advantages of the synchronous languages: fundamental properties such as the input-output determinism are maintained as well as the fully orthogonal structure of the programming language, which allows developers to arbitrarily nest all kinds of statements.

7.1.1 Multiple Clocks in Synchronous Models

The *multi-clock* extension of Esterel presented in [82] assumes a set of independent clock domains communicating via signals. The semantics are given by a transformation to single-clocked Esterel, where all clocks occur as *additional inputs* to the whole system. In this way, the determinism of the system is achieved by relocating the clock triggers to the environment. For synthesizing distributed systems, problems of clock synchronization still occur for the environment. Such multi-clock systems are clock driven for each clock. In contrast, our subclocked systems are still driven by a single clock and provide a logical refinement of clocks and steps. Furthermore, the multi-clock extension is not orthogonal to all other statement, but bound to modules.

Multi-clocked systems can be also described by the synchronous language *Lustre*[143]. Each Lustre program basically consists of a set of equations over data streams. In addition to functional symbols and delays, there are two operators to change the rate of a stream. The downsampling operator *when* takes a stream of arbitrary type and a Boolean stream and only keeps the events of the first one at those instants in which the second one is true. Upsampling is done by the *current* operator. This undoes a previous sampling operation and inserts in the missing locations the last actual value. Thus, all clocks in a Lustre program are subsets of the so-called base clock. Since upsampling only undoes the last downsampling, there is no mean to refine this base clock. It contains all instants at which any computation or communication may happen. Lustre specifications are completely deterministic due to their bottom-up design from the base clock. At each of the sequentially ordered instants of the base clock, the clock calculus determines which values must be computed.

In contrast, the polychronous language *Signal*[180, 181] has a different underlying model of computation: while the syntax looks almost like Lustre, its semantic foundations are very different due to its assumption that there is no base clock. In consequence, Signal specifications are relational and not functional like Lustre: they do not describe a single behavior but several ones, which differ in the clocks. Hence, Signal solves all the above problems. However, the price one has to pay for this powerful model is that input-output determinism is lost in general. It can be guaranteed if the program is shown to be endochronous or weakly endochronous. While endochrony proves determinism by the existence of a base clock (usually called master trigger in this context), weak endochrony also reveal some internal nondeterminism that can be safely exploited for a more efficient execution.

7.2 Contribution

7.2.1 Clock Refinement [28, 30, 33]

In [28], we present an extension of the imperative synchronous programming language Quartz which gives developers the possibility to declare so-called *subclocks*. These subclocks are the first full support of a top-down hierarchy of clocks. They are an appropriate means to refine the temporal behavior of synchronous systems to avoid over-synchronization and to allow one to exchange components with others having a different internal temporal behavior. Hence, clock refinements provide additional degrees of freedom for synthesis and design space exploration. At the same time, we preserve all desired advantages of the synchronous languages: fundamental properties such as the input-output determinism are maintained as well as the fully orthogonal structure of the programming language, which allows developers to arbitrarily nest all kinds of statements.

To overcome these problems, we propose the introduction of so-called subclocks, which allow developers to divide macro steps on the system level into smaller steps — thus, macro steps are

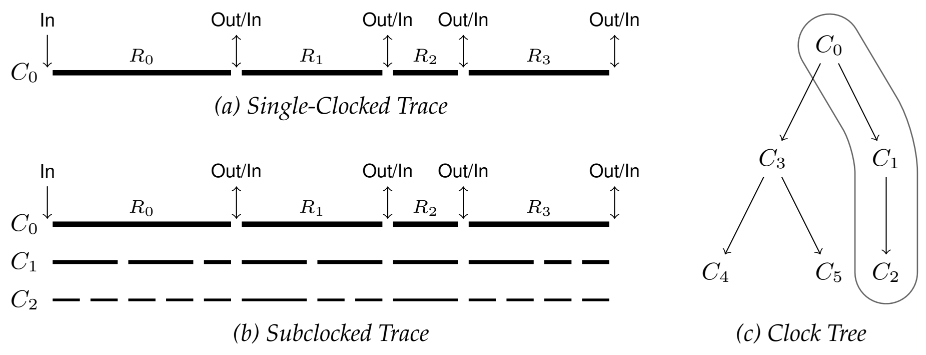


Fig. 13. Timing of single clocked and subclocked systems

no longer atomic. However, *we do not make this abstraction visible to the environment of modules*. The system interface has the same timing behavior, while its internal implementation has more freedom due to its internal subclocks. One instant on a clock level is divided into some smaller steps of the lower clock level as shown in Figure 13 (b). Thereby, variables of subclocks can have multiple values during a step on a higher level, but the variables are not visible to the higher level. Similar to the distinction of micro- and macro steps, the computation which is done in one step, is hidden to the higher level and only the result is visible. The advantage compared to micro- and macro steps is that the clock hierarchy provided hereby can be arbitrarily deep nested.

It is possible to refine a clock by multiple unrelated subclocks. This leads to a tree of clocks shown in Figure 13 (c), whereas only the marked branch of the tree provides the clocks shown in the trace in Figure 13 (b). Unrelated clock domains must not share variables, but they can communicate over variables declared on common superclocks. Thereby, they can run independently until a clock tick of the common superclock arises, which enforces a synchronization of its derived clock levels.

Finally, the main contribution of [30] is a *structural operational semantics* of our language extension and a theorem that states that refined clocks preserve the input/output determinism of the single-clocked synchronous model, while providing additional degrees of freedom for the internal implementation and execution.

7.2.2 Adapting the Design Flow [29, 31, 32]

In [29], we consider the compilation of our imperative synchronous programs with refinement by subclocks. The contribution of that paper is thereby twofold: first, we define a new intermediate language based on synchronous guarded actions. This intermediate language resolves the complex interaction of different statements at the source code level, so that back-end tools can better concentrate on efficient analysis and code generation procedures. Second, we present a preliminary compilation algorithm that translates the extended synchronous programs to the intermediate language. Thereby, we focus on the compilation of complex control flow statements (such as different kinds of preemption) in the context of different clock levels and do not elaborate on special issues such as schizophrenic local variables or modular compilation.

While [29] presents a preliminary compilation scheme, it can only translate the control-flow of a program completely. In particular, schizophrenia and causality problems could not be handled, so far. We close this gap in [32], where these classical problems are considered in the context of refined clocks. Its main contribution is a compilation procedure that copes with both problems and translates a Quartz program to synchronous guarded actions with sub-clocks. To support the practicability of the translation, we show how hardware synthesis can benefit from these sub-clocks to generate a synchronous circuit.

We generalize previous work on causality analysis of synchronous languages in [31]. Whereas previous work relies on a single clock, i. e. logical time is divided into a sequence of equal instants, our model of computation is more general: it allows the refinement of steps into substeps so that we have a hierarchical partition of the program execution. Obviously, this requires a more difficult causality analysis as events on different levels must be analyzed.

Our contribution is twofold: (1) we give a formal definition of causality for systems with refined clocks, and (2) we list conservative approximations to quickly check the causality of a system at compile-time.

7.2.3 Syntactic Sugar [16, 17]

These papers demonstrates the expressive power of the synchronous programming model in that we show how different aspects like the description of the modular architecture, complex properties as given by ω-regular properties similar to PSL and assume-guarantee reasoning can be incorporated in a synchronous language. In particular, we show that regular expressions and temporal logics can be easily translated to even more readable synchronous programs. Besides the better readability of the specification, the main advantage is the possibility to use existing tools for synchronous languages to simulate, verify, and debug the specified properties as well as the program. Moreover, all translations to hardware and software already offered by compilers for synchronous languages can still be used for hardware-software codesign. The article [17] is an extended version of the paper [16].

8

Integration

8.1 Context

8.1.1 Models of Computation

For the design of embedded systems, a plethora of languages based on different models of computation (MoC) [70, 135, 167, 185, 186] have been proposed over the years. For example, languages like Verilog [161], VHDL [163], and SystemC [162] are based on an event-driven paradigm [97], synchronous languages [74, 142] such as Esterel [77, 80, 86], Lustre [96, 144], Quartz [229], and some statechart variants are based on clock-driven paradigms, polychronous languages like Signal [69, 134, 180] are based on a declarative and non-deterministic paradigm using several clocks, data flow languages like CAL [127] are based on data-driven paradigms [182, 183, 184], and others like SHIM [123] or most multi-threaded languages are based on asynchronous threads with a rendezvous-style communication [155, 156].

Depending on a particular application domain such as digital signal processing or reactive controllers, depending on the design task such as modeling, simulation, analysis or synthesis, and depending on the synthesis target such as digital hardware circuits, multithreaded software or software for heterogeneous MPSoCs, the one or the other language might be preferable. For this reason, many existing components are given in different languages using different MoCs. The co-simulation of such heterogeneous systems has been widely considered [84, 202, 222, 247, 258] and covers also languages with different MoCs [100]. Co-simulation is also used to create virtual prototypes that are required to achieve hard time-to-market constraints.

However, co-simulation alone is not sufficient for a seamless design flow. Formal verification and a common synthesis of the different components require an *integration* that has gained a lot of interest in recent years [129, 140, 147, 154, 173, 203, 223, 224, 253]. Moreover, having a common description for the different components at hand, one can easily combine the components, e.g. one can create new components by using existing ones in a hierarchical way as known in block/schematic-oriented languages such as Simulink or those used by many tools for digital hardware circuit design. This way, one is no longer restricted in a parallel composition of the heterogeneous components by using appropriate wrappers. Instead, one can create a hierarchy of modules that combine modules based on different MoCs. This allows one to establish a design flow using several steps of refinement where asynchronous descriptions can become synchronous by adding a schedule to clocks, and synchronous ones may become asynchronous again when one considers the actions scheduled to one clock tick. Finally, one can re-use existing backend tools for synthesis and verification and of course, simulation would be greatly simplified: Instead of coupling different simulators, a single one could consider the entire system in a way it will later on be verified and synthesized.

8.1.2 Concurrent Action-Oriented Specifications

Concurrent Action-Oriented Specifications (CAOS) describe hardware circuits at a level higher than RTL. They abstract from concrete timing and model the data-flow and causalities in the system. In the following, we do not introduce their full syntax and semantics, we focus on simple asynchronous guarded actions in the form of rules and methods, which are the language core and sufficient to define other statements as simple syntactic sugar. We first give an overview of the syntax before we describe their semantics in the next subsection.

Each CAOS model is defined over a set of explicitly declared variables $\mathcal{V}$, which represent the state of the modeled component. The behavior is described by a set of *rules*, which are guarded atomic actions of the form r_i : $\texttt{when}(\gamma_i)\ S_i$, where γ_i is called the guard and S_i is called the body of rule r_i.

$$\begin{array}{c}\texttt{rule}\ r_1\ \texttt{when}(\gamma_1)\ S_1\\ \vdots\\ \texttt{rule}\ r_n\ \texttt{when}(\gamma_n)\ S_n\end{array}$$

Provided that σ is a Boolean expression, τ an expression of appropriate type over the readable variables, and x a writeable variable, the body S is one of the following statements:

`nothing;`	(no operation)
$x = \tau;$	(wire assignment)
$\mathsf{next}(x) = \tau;$	(register assignment)
$S_1\ S_2$	(parallel composition)
$\texttt{if}(\sigma)\ S_1\ \texttt{else}\ S_2$	(alternative)

CAOS provides two kinds of assignments: while wire assignments are immediately visible, register assignments are committed with the current state update. Other CAOS formalisms (like Bluespec) often distinguish these two variants by variable declarations, i. e. variables are declared either as wires or as registers. We use the more general approach in which appropriate assignment types distinguish between a state variable and a wire type variable. Several assignments can be combined in the body by parallel composition, which is simply written as the concatenation. Finally, alternatives in rules can be given by the `if` operator.

For the interaction with the environment, CAOS uses so-called methods, which are parameterized rules. In addition to the local variables, the action of a method has access to the variables specified in its parameter list. Traditionally, CAOS distinguishes so-called *action methods*, *value methods* and *action-value methods*: as the name suggests, an action method executes an action, which only transports data given by the parameters into the system, while a value method does not change the system state and simply returns a value — thus, they only transport data from the system to the outside. Action-value methods are just combinations of them. Frequently, it is required that the outputs of these methods do not depend on the inputs, since this can lead to cyclic dependencies. We do not make this assumption here, since our translation target (synchronous guarded actions) can deal with cyclic dependencies, and well-established analysis tools will spot problematic situations.

$$\begin{array}{c}\texttt{method}\ m_1(p_{11}, p_{12}, \ldots)\ \texttt{when}(\gamma_1)\ S_1\\ \vdots\\ \texttt{method}\ m_n(p_{n1}, p_{n2}, \ldots)\ \texttt{when}(\gamma_n)\ S_n\end{array}$$

The CAOS semantics is very simple. After the initialization of all the variables, the following two steps are repeated forever: first, the guards of all actions are evaluated with respect to the current state. Among the actions whose guards evaluate to true, an arbitrary one is chosen and its body is executed. The execution generally modifies the system state so that other actions will be possibly activated in the following iteration. If no action is activated, the loop may be also aborted, since no state change will occur from there on.

8.1.3 Polychronous Specifications

In contrast to synchronous systems, polychronous specifications [134, 180] are based on a partially ordered model of time. Partially ordered time model allows one to express asynchronous computations which possibly need to synchronize intermittently. As the name suggests, polychrony makes use of several clocks, which means that signals do not need to be present at all instants. Since the used clocks may not imply each other, polychronous models are not based on a linear model of time, so that the reactions of a polychronous system are only partially ordered. Two instants can be only compared on the time scale if both contain events of a shared signal x.

Another aspect of polychronous specifications is that they are *relational*, rather than *functional*. A polychronous behavior is not described in an operational way, but rather, it is constrained by relational clauses. Obviously, due to the relational approach, polychronous specifications are generally nondeterministic, when constraints do not sufficiently specify the suited functional behavior. Even in the presence of the same input values, various temporal alignments, which comply to the constraints, may lead to different output values. In contrast, synchronous modules deterministically react to any possible input configuration. The primary concern of a polychronous system are the constraints to interface the system with possibly asynchronous inputs. This problem is solved by, first, providing the specification of (possibly non-deterministic) input/output constraints and, second, determining a solution by the automatic synthesis of a controller enforcing the specified input/output timing constraints. Hence, polychronous models may be seen as specifications, which describe a set of acceptable implementations. There are three different types of clauses, which restrict the overall behavior:

- *equations* define the *values* of signals in terms of each other,
- *clock constraints* define the presence and absence of signals in an instant, i. e. how signals are temporally aligned (We denote the clock of x by $\widehat{x}$, which holds if and only if x is present (i.e. x has an event) in a given instant.), and
- *causal dependencies* describe the order in which the values of the signals are determined within an instant ($x \xrightarrow{\phi} y$ means that there is a dependency from x to y in all instants where ϕ holds.)

Each signal implicitly defines the dependency $\widehat{x} \xrightarrow{\widehat{x}} x$, i. e. the status (presence or absence) of a signal x must be known before we can determine its value. In the same way, operators and equations also impose clock constraints *and* causal dependencies.

In the following, we use Signal programs as polychronous specifications, which generally consist of a composition of several nodes. Each node has an input interface consisting of input signals, an output interface consisting of output signals and several possible internal signals. Its body is given by the composition of other nodes and/or a set of basic equations, which can be built from one of the following four primitive operators:

Function. A general function $\langle y := \texttt{f}(x_1, \ldots, x_n) \rangle$ can have an arbitrary number of inputs $x_1, \ldots, x_n$ and an arbitrary number of outputs $y_1, \ldots, y_m$. The output values are determined by applying the given function to the input values. This node requires that all inputs have the same clock, and it produces the outputs also at the same instant, i. e. $\widehat{x_1} = \ldots = \widehat{x_n} = \widehat{y_1} = \ldots = \widehat{y_m}$.

Obviously, there are causal dependencies from the inputs to the output of the node (each time there are values), i. e. $x_1 \xrightarrow{\widehat{y_j}} y_j, \ldots, x_n \xrightarrow{\widehat{y_k}} y_k$.

Delay. The delay operator $\langle y := x$ `$` **`init`** $c\rangle$ has exactly one input x and one output y. Its behavior consists of two micro steps: Each time a new incoming value arrives, it outputs the previously stored value and stores the new value. For the initial value, the buffer simply returns the given value c. By definition, the input and the output have the same clock, i. e. $\widehat{x} = \widehat{y}$. Since the output never depends on the input of the same instant, this node does not impose any causal dependencies.

When. The downsampling operator $\langle y := x_1$ **`when`** $x_2\rangle$ has two inputs, x_1 of arbitrary type and x_2 of Boolean type, and one output y. Each time a new x_1 arrives, it checks whether there is an input at x_2. If there is one and if it is true, a new output event with the value of x_1 is emitted for y. In all other cases, i. e. if x_1 or x_2 is absent or x_2 has the value false, no event will be produced. Thus, we obtain the following clock constraint $\widehat{y} = \widehat{x_1} \wedge \widehat{x_2} \wedge x_2$. As the input is immediately forwarded, there is a causal dependency from x_1 to y: $x_1 \xrightarrow{\widehat{y}} y$. Note that there is no dependency from the second input x_2 to the output y since it only influences its status - not its value.

Default. The merge operator $\langle y := x_1$ **`default`** $x_2\rangle$ has two inputs x_1 and x_2 and a single output y. Each time an input arrives at x_1, it will be forwarded to y. If there are events present at both inputs in a particular instant, the value of x_1 will be forwarded, and the value of x_2 will be discarded. If x_1 is absent, and there is only a value for x_2, x_2 will be forwarded. Hence, the operator always gives priority to its first input, i. e. we have the clock constraint $\widehat{y} = \widehat{x_1} \vee \widehat{x_2}$, and the dependencies $x_1 \xrightarrow{\widehat{x_1}} y$ and $x_2 \xrightarrow{\widehat{x_2} \wedge \neg\widehat{x_1}} y$.

In addition to these basic nodes, programs may contain additional clock constraints to restrict the behavior. For example, clocks can be declared to be equal $\widehat{x} = \widehat{y}$, mutually exclusive $\widehat{x} \oplus \widehat{y}$, or a clock can be declared to be a subclock of another one $\widehat{x} \to \widehat{y}$.

```
process Counter =
(? integer n;
 ! integer o;)
(| c := o $ init 0
 | o := n default (c-1)
 | n ^= (when (c=0))
 |)
where
    integer c;
end;
```

Fig. 14. Signal Example: `Counter`

The idea of Signal is illustrated with the help of an example, which is given in Figure 14. It implements a simple counter which has one input `n` and one output `o`. The intention of the process is that for each input value n, the output values $n, n-1, \ldots, 0$ are produced. To this end, the local signal `c` stores the last value of the produced output, whereas `o` is produced by subtraction of 1 from `c`. If a new value for the input `n` arrives, the output is updated by this value. The clock constraint `n ^= (`**`when`**` (c = 0))` ensures that new inputs is read when the local signal `c` reaches 0. Thus, the countdown is never aborted, and the initial value for the next countdown is guaranteed to be read in time.

8.2 Contribution

8.2.1 Interfacing SystemC [12, 18]

As a synchronous language, the execution of an Esterel program is divided into macro steps that correspond with single reactions that are triggered by a common clock of a hardware circuit. Each macro step is divided into finitely many microsteps that are all executed in zero time and within the same variable environment. Hence, the execution of Esterel programs are driven in a cycle-based fashion. Due to the instantaneous reaction of microsteps, causality problems may occur if actions modify variables whose values are responsible for triggering the action. In order to analyze the causality of programs, a fixpoint iteration may be performed to compute the reaction of a macrostep. It is well-known that this fixpoint iteration is the ternary simulation [91] of the corresponding hardware circuits. However, it has to be remarked that Esterel compilers usually perform this fixpoint analysis at compile time, so that (1) more efficient code is generated and (2) it is known at compile time that the iteration finally terminates with known values.

SystemC follows the discrete-event semantics that are well-known from hardware description languages like VHDL [164] and Verilog [165]. A SystemC program consists of a set of processes that run in parallel. SystemC distinguishes thereby between three classes of processes, namely 'methods', asynchronous processes and synchronous processes. Methods are special cases of asynchronous processes that do not have **wait** statements. Asynchronous processes are triggered by events, i.e., by changes of the variables on which the process depends, and they are executed as long as variable changes are seen. For this reason, the execution of the asynchronous processes is also a fixpoint computation that terminates as soon as a fixpoint of the variables' values is found. After this, the synchronous processes are executed once to complete the simulation cycle.

In [18], we outline the differences and similarities of synchronous languages like Esterel and SystemC. In particular, we define classes of systems that can be easily described in both languages in a way that allows one to structurally translate these descriptions into each other. This is the result of the similarities that we have identified between the two languages. On the other hand, the differences we will outline in the following may be interesting for those who work on later versions of both languages. With [18], we therefore hope to stimulate the discussion between the communities of SystemC and synchronous languages.

We address a broader scope in [12]. It provides a complete translation of synchronous programs to SystemC so that modules written in synchronous languages like Esterel can be smoothly integrated with SystemC. We implemented this translation for our synchronous language Quartz [229] and applied it successfully to some standard benchmarks [38, 39, 78, 229] to reason about programs with difficult causality cycles and schizophrenia problems.

The benefits of our translation are manyfold. First, it gives a detailed insight in the relationship of both models of computation so that the research done in these communities can be combined, and research/engineering results can be shared. For example, solutions that have been developed to solve problems for synchronous languages (like causality problems) can be transferred to solve corresponding problems of SystemC. In addition, these problems can now also be analyzed by means of fast simulation using the SystemC simulation. Conversely, certain runtime problems of SystemC programs can be solved by means of a fixpoint analysis that has been originally developed for synchronous languages.

Of course, verification results obtained for a synchronous program are still valid after its translation to SystemC, since our translation preserves the semantics. From a practical perspective, a major advantage of our translation is the integration of synchronous components in a SystemC description while preserving the synchronous semantics. For example, this can be used for fast simulation or the generation of virtual prototypes. In addition to a very efficient simulation,

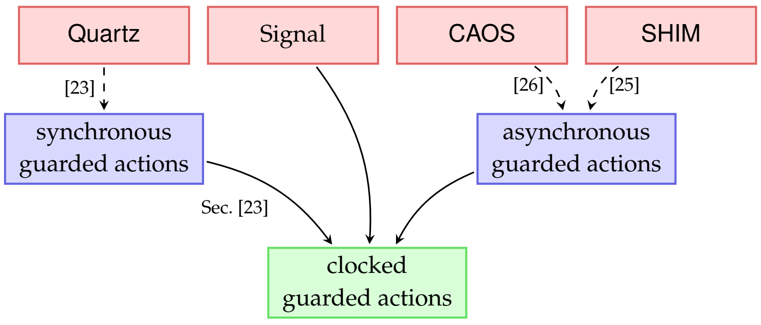


Fig. 15. Design flow for clocked guarded actions

the rich library of SystemC components including many microprocessors becomes available for synchronous programming.

8.2.2 Clocked Guarded Actions [13, 14, 15, 42]

In [14], we propose *clocked guarded actions* as an intermediate representation to cover various MoCs used for the design of embedded systems. This representation is in the spirit of guarded commands, a well-established concept for the description of concurrent systems. With a theoretical background in conditional term rewriting systems [111, 139, 169], guarded actions have been not only used in many specification and verification formalisms (e. g. Dijkstra's guarded commands [112], Unity [98], Murphi [113]) but they have also shown their power in hardware and software synthesis (e. g. Bluespec [61, 157] and Concurrent Action-Oriented Specifications [158]).

To demonstrate the power of our approach, we sketch a design flow based on our intermediate representation (see Figure 15): In particular, we show how different languages based on different MoCs can be translated to clocked guarded actions (see Section 8.2.3).

The article [14] is an extension of a previous paper [13], which introduced clocked guarded actions as a common intermediate representation. This is extended by introducing control-flow contexts, which allows us to nest components in an arbitrary behavioral hierarchy, i. e. we can *call* components from other components. Obviously, this has not only consequences for the intermediate representation itself but also related tasks: the translation to the intermediate format must be revised, and the procedure to link several components must be generalized.

Clocked guarded actions (CGA), which we propose in [14], provide a basis to integrate both variants. As the name suggests, they are defined over a set of explicitly declared *clocks* $\mathcal{C}$, which define logical timescales the system uses to synchronize its computations so that asynchrony and synchrony in the system can be precisely described. The basis of the whole temporal model are so-called instants, i. e. the points of time where some event in the system occurs.

The actions of programs are scheduled to a set of instants (also called reactions or macro steps [146]). The actions that take place within an instant (sometimes called micro steps) are not explicitly ordered. Instead, micro steps are assumed to happen simultaneously, i. e. in the same variable environment. Hence, variables seem to be constant during the execution of the micro steps and only change synchronously at macro steps. From the semantical point of view, which postulates that a reaction is atomic, neither communication nor computation take time in this sense. In reality, all actions within an instant are executed according to their data dependencies to establish the illusion of zero-time computations.

8.2.3 Translations [22, 25, 26]

The design flow sketched 15 requires several translations, which have been addressed in several publications.

The contribution of [26] is the provision of a compilation procedure from CAOS to synchronous guarded actions. In contrast to previous work, which always considered the translation of the model of computation and the scheduling simultaneously, our approach separates these two tasks. From the original CAOS model, we do not construct one particular synchronous implementation, but we create synchronous guarded actions that cover all possible implementations. Since this model is deterministic, we encode the choice between all the possible schedules as additional inputs to the system which are constrained by generated assertions. Thus, linking a scheduler to the system which complies with the assertions results in one of the possible deterministic synchronous implementations that implement the original CAOS model.

In order to construct the scheduler constraints, we formally define the correctness preservation criterion, i. e. we fix the properties of the system which have to be obeyed by an implementation so that it is considered to be valid with respect to the original CAOS model. In principle, all previous definitions of conflict-free transitions had the same goal, but they were always made with consequences for a specific implementation in mind. Since we do not want to make any restrictions in advance, we aim at providing very general constraints, which are only implied by the CAOS semantics.

The separation of concerns does not only provide a better theoretical understanding of existing CAOS languages and synthesis tools such as Bluespec, but it also has many practical advantages: The representation as synchronous guarded actions allows us to carry out many analyses and optimizations known from the synchronous domain. In particular, specification and verification based on temporal logics becomes possible. Moreover, we can reuse synthesis tools for synchronous guarded actions, which support hardware and software as targets.

The contribution of [25] is a translation algorithm that takes a SHIM-like program and generates asynchronous guarded actions for it. By translating SHIM to guarded actions, we leverage many existing techniques for hardware/software codesign and formal verification. The approach presented in [25] complements existing compilation techniques for SHIM, which focus on software generation. Furthermore, by using the conflict analysis presented by Brandt et al. [26], the SHIM approach can harness all the implementation and verification techniques developed for the synchronous world.

Previous work focused mainly on software generation. Edwards and Tardieu introduced initial ideas [123, 244, 245] and later refined them. For example, Edwards and Tardieu [124] showed how threads can be statically scheduled to minimize the synchronization overhead in generated code. Later, Edwards, Vasudevan, and Tardieu [125] use Pthreads as a target for software synthesis. Edwards and Tardieu [123] sketch a rudimentary hardware generation, but only cover a subset of the language. While deadlock detection [250] and buffer sharing [251] has been considered for SHIM, much work remains to be done on formal verification and static analysis.

In [22], we focus on synchronous and asynchronous guarded actions. As already sketched above, *synchronous systems* are based on the notion of steps. In each step, they read inputs, update their state and produce outputs. These steps are atomic in the sense that synchronous systems do not describe any relationship between the actions that are executed in a step; they abstract from the causality, which is implicitly given by the data dependencies (e. g. one does not need to give an evaluation order for the gates in a synchronous circuit). In contrast, *asynchronous systems* do not have a predefined notion of time: they simply describe what actions follow another one, i. e. they precisely describe the causality. *Hence, the synchronous and the asynchronous models focus on two different aspects: timing and causality.*

The main contribution of [22] consists of the presentation of transformations between these two MoCs, which pave the way for a better exchangeability of synchronous and asynchronous models in a flexible model-based design flow. As the previous explanations suggest, the transformation from the synchronous to the asynchronous domain extracts causality, while the transformation in the opposite direction maps actions to steps according to their dependencies.

9 Bibliography

9.1 Publications of this Habilitation

The following list contains all the publications that are part of this habilitation thesis.

[1] Y. Bai, J. Brandt, and K. Schneider. Data-flow analysis of extended finite state machines. In B. Caillaud, J. Carmona, and K. Hiraishi, editors, *Application of Concurrency to System Design (ACSD)*, pages 163–172, Newcastle Upon Tyne, England, UK, 2011. IEEE Computer Society.

[2] Y. Bai, J. Brandt, and K. Schneider. SMT-based optimization for synchronous programs. In S. Stuijk, editor, *Software and Compilers for Embedded Systems (SCOPES)*, pages 11–20, St. Goar, Germany, 2011. ACM.

[3] Y. Bai, J. Brandt, and K. Schneider. Monitoring distributed reactive systems. In *High Level Design Validation and Test (HLDVT)*, Huntington Beach, California, USA, 2012. IEEE Computer Society.

[4] Y. Bai, J. Brandt, and K. Schneider. Preservation of LTL properties in desynchronized systems. In S. Shukla, L. Carloni, D. Kroening, and J. Brandt, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 53–63, Arlington, Virginia, USA, 2012. ACM.

[5] D. Baudisch, J. Brandt, and K. Schneider. Dependency-driven distribution of synchronous programs. In M. Hinchey, B. Kleinjohann, L. Kleinjohann, P.A. Lindsay, F.J. Rammig, J. Timmis, and M. Wolf, editors, *Distributed and Parallel Embedded Systems (DIPES)*, pages 169–180, Brisbane, Queensland, Australia, 2010. International Federation for Information Processing (IFIP).

[6] 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)*, pages 949–952, Dresden, Germany, 2010. EDA Consortium.

[7] D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Generating software pipelines for OpenMP. In M. Dietrich, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 11–20, Dresden, Germany, 2010. Fraunhofer Verlag.

[8] D. Baudisch, J. Brandt, and K. Schneider. Translating synchronous systems to data-flow process networks. In S.-S. Yeo, B. Vaidya, and G.A. Papadopoulos, editors, *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 354–361, Gwangju, Korea, 2011. IEEE Computer Society.

[9] D. Baudisch, J. Brandt, and K. Schneider. Efficient handling of arrays in dataflow process networks. In *International Conference on Embedded Software and Systems (ICESS)*, Liverpool, UK, 2012. IEEE Computer Society.

[10] D. Baudisch, J. Brandt, and K. Schneider. Out-of-order execution of synchronous data-flow networks. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (ICSAMOS)*, Samos, Greece, 2012. IEEE Computer Society.

[11] J. Brandt, M. Gemünde, and K. Schneider. Desynchronizing synchronous programs by modes. In S. Edwards, R. Lorenz, and W. Vogler, editors, *Application of Concurrency to System Design (ACSD)*, pages 32–41, Augsburg, Germany, 2009. IEEE Computer Society.

[12] J. Brandt, M. Gemünde, and K. Schneider. From synchronous guarded actions to SystemC. In M. Dietrich, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 187–196, Dresden, Germany, 2010. Fraunhofer Verlag.

[13] J. Brandt, M. Gemünde, K. Schneider, S. Shukla, and J.-P. Talpin. Integrating system descriptions by clocked guarded actions. In A. Morawiec, J. Hinderscheit, and O. Ghenassia, editors, *Forum on Specification and Design Languages (FDL)*, pages 1–8, Oldenburg, Germany, 2011. IEEE Computer Society.

[14] J. Brandt, M. Gemünde, K. Schneider, S.K. Shukla, and J.-P. Talpin. Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions. *Design Automation for Embedded Systems (DAEM)*, July 2012. DOI 10.1007/s10617-012-9087-9.

[15] J. Brandt, M. Gemünde, K. Schneider, S.K. Shukla, and J.-P. Talpin. Embedding polychrony into synchrony. *IEEE Transactions on Software Engineering (TSE)*, 39(7):917–929, July 2013.

[16] J. Brandt and K. Schneider. System description aspects as syntactic sugar. In *Forum on Specification and Design Languages (FDL)*, pages 293–301, Darmstadt, Germany, 2006. Electronic Chips and Systems Design Initiative (ECSI).

[17] J. Brandt and K. Schneider. Different kinds of system descriptions as synchronous programs. In S.A. Huss, editor, *Advances in Design and Specification Languages for Embedded Systems*, pages 243–263. Springer, 2007. ISBN: 978-1-4020-6147-9.

[18] J. Brandt and K. Schneider. How different are Esterel and SystemC? In *Forum on Specification and Design Languages (FDL)*, pages 98–103, Barcelona, Spain, 2007. Electronic Chips and Systems Design Initiative (ECSI).

[19] J. Brandt and K. Schneider. Formal reasoning about causality analysis. In O. Ait Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 5170 of *LNCS*, pages 118–133, Montréal, Québec, Canada, 2008. Springer.

[20] 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.

[21] J. Brandt and K. Schneider. Static data-flow analysis of synchronous programs. In R. Bloem and P. Schaumont, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 161–170, Cambridge, Massachusetts, USA, 2009. IEEE Computer Society.

[22] J. Brandt and K. Schneider. Round trip to asynchrony and synchrony. In F. Oppenheimer, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 239–248, Oldenburg, Germany, 2011. OFFIS-Institut für Informatik.

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

[24] J. Brandt, K. Schneider, and Y. Bai. Passive code in synchronous programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 2013. (to appear).

[25] J. Brandt, K. Schneider, and S.A. Edwards. Translating SHIM to guarded actions. Technical Report 387/12, University of Kaiserslautern, Kaiserslautern, Germany, February 2012.

[26] J. Brandt, K. Schneider, and S.K. Shukla. Translating concurrent action oriented specifications to synchronous guarded actions. In J. Lee and B.R. Childers, editors, *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 47–56, Stockholm, Sweden, 2010. ACM.

[27] J. Brandt, K. Schneider, and A. Willenbücher. Using IP cores in synchronous languages. In C. Gremzow and N. Moser, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 97–106, Berlin, Germany, 2009. Universitätsbibliothek Berlin, Germany. ISBN 9783798321182.

[28] M. Gemünde, J. Brandt, and K. Schneider. Clock refinement in imperative synchronous languages. In A. Benveniste, S.A. Edwards, E. Lee, K. Schneider, and R. von Hanxleden, editors, *SYNCHRON'09: Abstracts Collection of Dagstuhl Seminar 09481*, Dagstuhl Seminar Proceedings, pages 3–21, Dagstuhl, Germany, 2010. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany. ISSN 1862-4405, http://www.dagstuhl.de/09481.

[29] M. Gemünde, J. Brandt, and K. Schneider. Compilation of imperative synchronous programs with refined clocks. In L. Carloni and B. Jobstmann, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 209–218, Grenoble, France, 2010. IEEE Computer Society.

[30] M. Gemünde, J. Brandt, and K. Schneider. A formal semantics of clock refinement in imperative synchronous languages. In L. Gomes, V. Khomenko, and J.M. Fernandes, editors, *Application of Concurrency to System Design (ACSD)*, pages 157–168, Braga, Portugal, 2010. IEEE Computer Society.

[31] M. Gemünde, J. Brandt, and K. Schneider. Causality analysis of synchronous programs with refined clocks. In *High Level Design Validation and Test Workshop (HLDVT)*, pages 25–32, Napa, California, USA, 2011. IEEE Computer Society.

[32] M. Gemünde, J. Brandt, and K. Schneider. Schizophrenia and causality in the context of refined clocks. In A. Morawiec, J. Hinderscheit, and O. Ghenassia, editors, *Forum on Specification and Design Languages (FDL)*, pages 1–8, Oldenburg, Germany, 2011. IEEE Computer Society.

[33] M. Gemünde, J. Brandt, and K. Schneider. Clock refinement in imperative synchronous languages. *EURASIP Journal on Embedded Systems*, April 2013.

[34] K. Schneider and J. Brandt. Performing causality analysis by bounded model checking. In *Application of Concurrency to System Design (ACSD)*, pages 78–87, Xi'an, China, 2008. IEEE Computer Society.

[35] 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, District of Columbia, USA, 2004. ACM.

[36] K. Schneider, J. Brandt, and T. Schuele. A verified compiler for synchronous programs with local declarations (proceedings version). In *Synchronous Languages, Applications, and Programming (SLAP)*, Barcelona, Spain, 2004.

[37] 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.

[38] 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.

[39] 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, Saint-Malo, France, 2005. IEEE Computer Society.

[40] K. Schneider, J. Brandt, and E. Vecchié. Efficient code generation from synchronous programs. In J.C. Hoe and J. Palsberg, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 165–174, Napa, California, USA, 2006. IEEE Computer Society.

[41] K. Schneider, J. Brandt, and E. Vecchié. Modular compilation of synchronous programs. In B. Kleinjohann, L. Kleinjohann, R.J. Machado, C. Pereira, and P.S. Thiagarajan, editors, *Distributed and Parallel Embedded Systems (DIPES)*, volume 225 of *IFIP Advances in Information and Communication Technology*, pages 75–84, Braga, Portugal, 2006. Springer.

[42] J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S. Shukla. Constructive polychronous systems. In S.N. Artëmov and A. Nerode, editors, *Logical Foundations of Computer Science (LFCS)*, volume 7734 of *LNCS*, pages 335–349, San Diego, California, USA, 2013. Springer.

9.2 Other Own Publications

The following list contains all my publications that are *not* part of this habilitation thesis.

[43] J. Brandt. 2D-Polygon-Clipping Algorithmen für eingebettete Echtzeitsysteme. Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany, September 2003.

[44] J. Brandt. *A Layered Approach to Polygon Processing for Safety-Critical Embedded Systems*. PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, Kaiserslautern, Germany, June 2007.

[45] J. Brandt, R. Gotzhein, R. Grammes, and B. Schürmann. Chatroom over WLAN-systematic development of an QoS-integrated distributed system. In *European Wireless 2004*, Barcelona, Spain, 2004. Technical University of Catalonia.

[46] J. Brandt and K. Heljanko, editors. *Application of Concurrency to System Design (ACSD)*, Hamburg, Germany, 2012. IEEE Computer Society.

[47] J. Brandt and A. Rettberg, editors. *Electronic System Level Synthesis Conference (ESLsyn)*, San Francisco, California, USA, 2012. IEEE Computer Society.

[48] J. Brandt and K. Schneider. Dependable polygon-processing algorithms for safety-critical embedded systems. In L.T.Yang, M. Amamiya, Z. Liu, M. Guo, and F.J. Rammig, editors, *Embedded and Ubiquitous Computing (EUC)*, volume 3824 of *LNCS*, pages 405–417, Nagasaki, Japan, 2005. Springer.

[49] J. Brandt and K. Schneider. Using three-valued logic to specify and verify algorithms of computational geometry. In K.-K. Lau and R. Banach, editors, *International Conference on Formal Engineering Methods (ICFEM)*, volume 3785 of *LNCS*, pages 405–420, Manchester, England, UK, 2005. Springer.

[50] J. Brandt and K. Schneider. Efficient map overlay for safety-critical embedded systems. In *Industrial Embedded Systems (IES)*, Antibes, France, 2006. IEEE Computer Society.

[51] J. Brandt and K. Schneider. *Embedded Systems: Status and Perspective*, chapter A Verified Polygon-Processing Library for Safety-Critical Embedded Systems. American Scientific Publishers, 2008.

[52] J. Brandt and K. Schneider, editors. *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, Kaiserslautern, Germany, 2012. Verlag Dr. Kovač.

[53] J. Brandt, K. Schneider, S. Ahuja, and S.K. Shukla. The model checking view to clock gating and operand isolation. In L. Gomes, V. Khomenko, and J.M. Fernandes, editors, *Application of Concurrency to System Design (ACSD)*, pages 181–190, Braga, Portugal, 2010. IEEE Computer Society.

[54] J. Brandt, K. Schneider, and A. Willenbücher. Hardware acceleration for model checking. In C. Scholl and S. Disch, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 179–187, Freiburg, Germany, 2008. Shaker.

[55] K. Schneider and J. Brandt. Theorem proving in higher order logics and applications – emerging trends. Internal Report 364/07, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, September 2007.

[56] K. Schneider and J. Brandt, editors. *Theorem Proving in Higher Order Logics (TPHOL)*, volume 4732 of *LNCS*, Kaiserslautern, Germany, 2007. Springer.

[57] K. Schneider, B. Jobstmann, L. Carloni, and J. Brandt, editors. *Formal Methods and Models for Codesign (MEMOCODE)*, Grenoble, France, 2010. IEEE Computer Society.

[58] S. Shukla, D. Kroening, L. Carloni, and J. Brandt, editors. *Formal Methods and Models for Codesign (MEMOCODE)*, Arlington, Virginia, USA, 2012. IEEE Computer Society.

[59] S. Singh, B. Jobstmann, M. Kishinevsky, and J. Brandt, editors. *Formal Methods and Models for Codesign (MEMOCODE)*, Cambridge, England, UK, 2011. IEEE Computer Society.

9.3 Publications by Other Autors

[60] F.E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, 1970.

[61] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification invited talk. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 249–254, Mont Saint-Michel, France, 2003. IEEE Computer Society.

[62] P. Aubry and T. Gautier. GC: the data-flow graph format of synchronous programming. *ACM SIGPLAN Notices*, 30(3):83–93, March 1995.

[63] K. Bai, D. Lu, and A. Shrivastava. Vector class on limited local memory (LLM) multi-core processors. In R.K. Gupta and V.J. Mooney, editors, *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 215–224, Taipei, Taiwan, 2011. ACM.

[64] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.

[65] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In O. Sokolsky and S. Tasiran, editors, *Runtime Verification (RV)*, volume 4839 of *LNCS*, pages 126–138, Vancouver, British Columbia, Canada, 2007. Springer.

[66] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, February 2010.

[67] A. Benveniste. Some synchronization issues when designing embedded systems from components. In T.A. Henzinger and C. Meyer Kirsch, editors, *Embedded Software (EMSOFT)*, volume 2211 of *LNCS*, pages 32–49, Tahoe City, California, USA, 2001. Springer.

[68] A. Benveniste and G. Berry. The synchronous approach to reactive real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[69] A. Benveniste, P. Bournai, T. Gautier, and P. Le Guernic. SIGNAL: A data flow oriented language for signal processing. Research Report 378, INRIA, Rennes, France, March 1985.

[70] A. Benveniste, B. Caillaud, L.P. Carloni, and A.L. Sangiovanni-Vincentelli. Tag machines. In W. Wolf, editor, *Embedded Software (EMSOFT)*, pages 255–263, Jersey City, New Jersey, USA, 2005. ACM.

[71] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, *Concurrency Theory (CONCUR)*, volume 1664 of *LNCS*, pages 162–177, Eindhoven, The Netherlands, 1999. Springer.

[72] 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.

[73] A. Benveniste, L.P. Carloni, P. Caspi, and A.L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In R. Alur and I. Lee,

editors, *Embedded Software (EMSOFT)*, volume 2855 of *LNCS*, pages 35–50, Philadelphia, Pennsylvania, USA, 2003. Springer.

[74] 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.

[75] G. Berry. A hardware implementation of pure Esterel. In *Formal Methods in VLSI Design*, Miami, Florida, USA, 1991.

[76] G. Berry. A hardware implementation of pure Esterel. *Sadhana*, 17(1):95–130, March 1992.

[77] 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.

[78] G. Berry. The constructive semantics of pure Esterel, July 1999.

[79] G. Berry. The Esterel v5 language primer, July 2000.

[80] G. Berry and L. Cosserat. The Esterel synchronous programming language and its mathematical semantics. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency (CONCUR)*, volume 197 of *LNCS*, pages 389–448, Pittsburgh, Pennsylvania, USA, 1985. Springer.

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

[82] G. Berry and E. Sentovich. Multiclock Esterel. In T. Margaria and T.F. Melham, editors, *Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 110–125, Livingston, Scotland, UK, 2001. Springer.

[83] S.S. Bhattacharyya, G. Brebner, J.W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet. OpenDF-a dataflow toolset for reconfigurable hardware and multicore systems. *ACM SIGARCH Computer Architecture News*, 36(5):29–35, December 2009.

[84] M. Bombana and F. Bruschi. SystemC-VHDL co-simulation and synthesis in the HW domain. In *Design, Automation and Test in Europe (DATE)*, pages 20101–20105, Munich, Germany, 2003. IEEE Computer Society.

[85] F. Boussinot. SugarCubes implementation of causality. Research Report 3487, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia Antipolis, France, September 1998.

[86] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[87] R.E. Bryant. A switch level model and simulator for MOS digital systems. *IEEE Transactions on Computers (T-C)*, C-33(2):160–177, February 1984.

[88] R.E. Bryant, D.L. Beatty, and C.H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Design Automation Conference (DAC)*, pages 397–402, San Francisco, California, USA, 1991. ACM.

[89] J.A. Brzozowski and C.-J.H. Seger. Advances in asynchronous circuit theory part I. Bulletin of the European association of Theoretical Computer Science, October 1990.

[90] J.A. Brzozowski and C.-J.H. Seger. Advances in asynchronous circuit theory part II. Bulletin of the European Association of Theoretical Computer Science, March 1991.

[91] J.A. Brzozowski and C.-J.H. Seger. *Asynchronous Circuits*. Springer, 1995.

[92] 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.

[93] 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.

[94] 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 (T-CAD)*, 20(9):1059–1076, 2001.

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

[96] P. Caspi, N. Halbwachs, D. Pilaud, and J.A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Principles of Programming Languages (POPL)*, pages 178–188, Munich, Germany, 1987. ACM.

[97] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2 edition, 2008.

[98] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Austin, Texas, USA, May 1989.

[99] D.M. Chapiro. *Globally-Asynchronous, Locally-Synchronous Systems*. PhD thesis, Department of Computer Science, Stanford, California, USA, 1984.

[100] A. Chatelain, Y. Mathys, G. Placido, A. La Rosa, and L. Lavagno. High-level architectural co-simulation using Esterel and C. In *Hardware-Software Codesign (CODES)*, Copenhagen, Denmark, 2001.

[101] F. Chen and G. Roşu. Parametric and sliced causality. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 240–253, Berlin, Germany, 2007. Springer.

[102] K. Claessen. Safety property verification of cyclic synchronous circuits. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 88:55–69, 2003. Workshop on Synchronous Languages, Applications, and Programming (SLAP).

[103] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, and S. Yovine. TAXYS: A tool for the development and verification of real-time embedded systems. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 391–395, Paris, France, 2001. Springer.

[104] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil. SAXO-RT: Interpreting Esterel semantics on a sequential execution structure. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 65(5):80–94, 2002. Workshop on Synchronous Languages, Applications, and Programming (SLAP).

[105] J. Cortadella, M. Kishinevsky, and B. Grundmann. SELF: Specification and design of synchronous elastic circuits. In *Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, 2006.

[106] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In E. Sentovich, editor, *Design Automation Conference (DAC)*, pages 657–662, San Francisco, California, USA, 2006. ACM.

[107] J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C.P. Sotiriou. From synchronous to asynchronous: An automatic approach. In *Design, Automation and Test in Europe (DATE)*, pages 1368–1369, Paris, France, 2004. IEEE Computer Society.

[108] N. Dave, Arvind, and M. Pellauer. Scheduling as rule composition. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 51–60, Nice, France, 2007. IEEE Computer Society.

[109] L. de Moura, B. Dutertre, and N. Shankar. A tutorial on satisfiability modulo theories. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 20–36, Berlin, Germany, 2007. Springer.

[110] D.Y. Deng, D. Lo, G. Malysa, S. Schneider, and G.E. Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In H. Kim, A. Raman, F. Liu, J.W. Lee, and D.I. August, editors, *Microarchitecture (MICRO)*, pages 137–148, Atlanta, Georgia, USA, 2010. IEEE Computer Society.

[111] N. Dershowitz, M. Okada, and G. Sivakumar. Canonical conditional rewrite systems. In E.L. Lusk and R.A. Overbeek, editors, *Conference on Automated Deduction (CADE)*, volume 310 of *LNCS*, pages 538–549, Argonne, Illinois, USA, 1988. Springer.

[112] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM (CACM)*, 18(8):453–457, 1975.

[113] 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, New Jersey, USA, 1996. Springer.

[114] F. Doucet, M. Menarini, I.H. Krüger, R. Gupta, and J.-P. Talpin. A verification approach for gals integration of synchronous components. In M. Singh and J.-P. Talpin, editors, *Formal Methods for Globally Asynchronous Locally Synchronous Design (FMGALS)*, Verona, Italy, 2005.

[115] A. Douillet and G.R. Gao. Software pipelining on multi core architectures. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 39–48, Brasov, Romania, 2007. IEEE Computer Society.

[116] S. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 21(2):169–183, February 2002.

[117] S. Edwards. ESUIF: An open Esterel compiler. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 65(5):79, 2002. Workshop on Synchronous Languages, Applications, and Programming (SLAP).

[118] S. Edwards. Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(2):141–187, 2003.

[119] S.A. Edwards. Compiling Esterel into sequential code. In *Hardware-Software Codesign (CODES)*, pages 147–151, Rome, Italy, 1999. ACM.

[120] S.A. Edwards. Making cyclic circuits acyclic. In *Design Automation Conference (DAC)*, pages 159–162, Anaheim, California, USA, 2003. ACM.

[121] S.A. Edwards, V. Kapadia, and M. Halas. Compiling Esterel into static discrete-event code. In *Synchronous Languages, Applications, and Programming (SLAP)*, Barcelona, Spain, 2004.

[122] S.A. Edwards, V. Kapadia, and M. Halas. Compiling Esterel into static discrete-event code. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(4):117–131, 2006.

[123] S.A. Edwards and O. Tardieu. SHIM: a deterministic model for heterogeneous embedded systems. In W. Wolf, editor, *Embedded Software (EMSOFT)*, pages 264–272, Jersey City, New Jersey, USA, 2005. ACM.

[124] S.A. Edwards and O. Tardieu. Efficient code generation from SHIM models. In M.J. Irwin and K. De Bosschere, editors, *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 125–134, Ottawa, Ontario, Canada, 2006. ACM.

[125] S.A. Edwards, N. Vasudevan, and O. Tardieu. Programming shared memory multiprocessors with deterministic Message-Passing concurrency: Compiling SHIM to pthreads. In *Design, Automation and Test in Europe (DATE)*, pages 1498–1503, Munich, Germany, 2008. IEEE Computer Society.

[126] E.B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9:90–99, 1965.

[127] J. Eker and J.W. Janneck. CAL actor language – language report. EECS Department, University of California at Berkeley, May 2002.

[128] J. Eker and J.W. Janneck. CAL language report. ERL Technical Memo UCB/ERL M03/48, EECS Department, University of California at Berkeley, Berkeley, California, USA, December 2003.

[129] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[130] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? Research Report TR-2010-5, Verimag, January 2010.

[131] J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers (T-C)*, C-30(7):478–490, July 1981.

[132] J.A. Fisher. Very long instruction word architectures and the ELI-512. In *International Symposium on Computer Architecture (ISCA)*, pages 140–150, Stockholm, Sweden, 1983. ACM/IEEE Computer Society.

[133] J.A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[134] A. Gamatié, T. Gautier, P. Le Guernic, and J.P. Talpin. Polychronous design of embedded real-time applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2), April 2007.

[135] A. Girault, B. Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 18(6):742–760, June 1999.

[136] A. Girault and C. Ménier. Automatic production of globally asynchronous locally synchronous systems. In A.L. Sangiovanni-Vincentelli and J. Sifakis, editors, *Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 266–281, Grenoble, France, 2002. Springer.

[137] A. Girault and X. Nicollin. Clock-driven automatic distribution of Lustre programs. In R. Alur and I. Lee, editors, *Embedded Software (EMSOFT)*, volume 2855 of *LNCS*, pages 206–222, Philadelphia, Pennsylvania, USA, 2003. Springer.

[138] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[139] B. Gramlich and C. Wirth. Confluence of terminating conditional rewrite systems revisited. In H. Ganzinger, editor, *Rewriting Techniques and Applications (RTA)*, volume 1103 of *LNCS*, pages 245–259, New Brunswick, New Jersey, USA, 1996. Springer.

[140] G. Gössler and A. Sangiovanni-Vincentelli. Compositional modeling in Metropolis. In A.L. Sangiovanni-Vincentelli and J. Sifakis, editors, *Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 93–107, Grenoble, France, 2002. Springer.

[141] A. Habibi and S. Tahar. A survey on system-on-a-chip design languages. In *IEEE 3rd International Workshop on System-on-Chip (IWSOC'03)*, Calgary, Canada, 2003. IEEE Computer Society.

[142] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[143] N. Halbwachs. A synchronous language at work: the story of Lustre. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 3–11, Verona, Italy, 2005. IEEE Computer Society.

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

[145] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming (PLILP)*, volume 528 of *LNCS*, pages 207–218, Passau, Germany, 1991. Springer.

[146] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[147] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich. A SystemC-based design methodology for digital signal processing systems. *EURASIP Journal on Embedded Systems*, 2007(1):15–15, January 2007.

[148] K. Havelund. Using runtime analysis to guide model checking of Java programs. In K. Havelund, J. Penix, and W. Visser, editors, *Model Checking Software (SPIN)*, volume 1885 of *LNCS*, pages 245–264, Stanford, California, USA, 2000. Springer.

[149] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design (FMSD)*, 24(2):189–215, March 2004.

[150] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 55(2):200–217, 2001.

[151] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2280 of *LNCS*, pages 342–356, Grenoble, France, 2002. Springer.

[152] M.S. Hecht and J.D. Ullman. Analysis of a simple algorithm global data flow problems. In *Principles of Programming Languages (POPL)*, pages 207–217, Boston, Massachusetts, USA, 1973. ACM.

[153] M.S. Hecht and J.D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM (JACM)*, 21(3):367–375, July 1974.

[154] F. Herrera and E. Villar. A framework for heterogeneous specification and design of electronic embedded systems in SystemC. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):1–31, August 2007.

[155] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM (CACM)*, 21(8):666–677, 1978.

[156] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM (CACM)*, 26(1):100–106, January 1983.

[157] J.C. Hoe and Arvind. Hardware synthesis from term rewriting systems. Technical Report CSG-MEMO 421-a, Computer Science and Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA, 1999.

[158] J.C. Hoe and Arvind. Operation-centric hardware description and synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 23(9):1277–1288, September 2004.

[159] W.A. Howard. To H.B. Curry: Essays on combinatory logic, lambda-calculus and formalism. chapter The formulas-as-types notion of construction, pages 479–490. Academic Press, 1980.

[160] D.A. Huffman. Combinational circuits with feedback. In A. Mukhopadhyay, editor, *Recent Developments in Switching Theory*, pages 27–55. Academic Press, 1971.

[161] IEEE. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, 2005. IEEE Std. 1394-2005.

[162] IEEE. *IEEE Standard SystemC Language Reference Manual*. New York, New York, USA, December 2005. IEEE Std. 1666-2005.

[163] IEEE. *IEEE Standard VHDL Language Reference Manual*, 2008. IEEE Std. 1076-2008.

[164] IEEE Computer Society. *IEEE Standard VHDL Language Reference Manual*. New York, USA, 2000. IEEE Std. 1076-2000.

[165] IEEE Computer Society. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. New York, USA, 2001. IEEE Std. 1394-2001.

[166] J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. In *Signal Processing Systems (SiPS)*, pages 287–292, Washington, District of Columbia, USA, 2008. IEEE Computer Society.

[167] A. Jantsch. Models of computation for networks on chip. In *Application of Concurrency to System Design (ACSD)*, pages 165–178, Turku, Finland, 2006. IEEE Computer Society.

[168] 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.

[169] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science (TCS)*, 33:175–193, 1984.

[170] W.H. Kautz. The necessity of closed circuit loops in minimal combinational circuits. *IEEE Transactions on Computers (T-C)*, C-19(2):162–166, February 1970.

[171] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 114–122, York, England, UK, 1999. IEEE Computer Society.

[172] P.M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill, 1981.

[173] T.J. Koo, J. Liebman, C. Ma, B. Horowitz, A. Sangiovanni-Vincentelli, and S. Sastry. Platform-based embedded software design for multi-vehicle multi-modal systems. In A.L. Sangiovanni-Vincentelli and J. Sifakis, editors, *Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 32–45, Grenoble, France, 2002. Springer.

[174] D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.

[175] 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.

[176] S.R. Kunkel and J.E. Smith. Optimal pipelining in supercomputers. In *International Symposium on Computer Architecture (ISCA)*, pages 404–414, Tokyo, Japan, 1986.

[177] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In R.L. Wexelblat, editor, *Programming Language Design and Implementation (PLDI)*, pages 318–328, Atlanta, Georgia, USA, 1988. ACM.

[178] L. Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Cooperation, 1991.

[179] J.-L. Lassez, V.L. Nguyen, and E.A. Sonenberg. Fixed point theorems and semantics – a folk tale. *Information Processing Letters*, 14(3):112–116, 1982.

[180] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[181] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers (JCSC)*, 12(3):261–304, June 2003.

[182] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers (T-C)*, 36(1):24–35, January 1987.

[183] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[184] E.A. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[185] E.A. Lee and A. Sangiovanni-Vincentelli. Comparing models of computation. In *International Conference on Computer-Aided Design (ICCAD)*, pages 234–241, San Jose, California, USA, 1996. ACM/IEEE Computer Society.

[186] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 17(12):1217–1229, December 1998.

[187] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, May/June 2009.

[188] X. Li, M. Boldt, and R. von Hanxleden. Compiling Esterel for a multi-threaded reactive processor. Technical Report 0603, Christian-Albrechts-Universität Kiel, Department of Computer Science, 2006.

[189] Y.-T.S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer, 1999.

[190] 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.

[191] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams: modularity vs. code size. In Z. Shao and B.C. Pierce, editors, *Principles of Programming Languages (POPL)*, pages 78–89, Savannah, Georgia, USA, 2009. ACM.

[192] R. Lublinerman and S. Tripakis. Modularity vs. reusability: Code generation from synchronous block diagrams. In *Design, Automation and Test in Europe (DATE)*, pages 1504–1509, Munich, Germany, 2008. IEEE Computer Society.

[193] J. Lukoschus and R. von Hanxleden. Removing cycles in Esterel programs. In *Synchronous Languages, Applications, and Programming (SLAP)*, Edinburgh, Scotland, UK, 2005.

[194] S. Malik. Analysis of cyclic combinational circuits. In *International Conference on Computer-Aided Design (ICCAD)*, pages 618–625, Santa Clara, California, USA, 1993. IEEE Computer Society.

[195] S. Malik. Analysis of cycle combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 13(7):950–956, July 1994.

[196] M. Mendler. Timing analysis of combinational circuits in intuitionistic propositional logic. *Formal Methods in System Design (FMSD)*, 17(1):5–37, August 2000.

[197] M. Mendler and M. Fairtlough. Ternary simulation: A refinement of binary functions or an abstraction of real-time behaviour. In *Designing Correct Circuits (DCC)*, Electronic Workshops in Computing, Båstad, Sweden, 1996. Springer.

[198] L. Mendonça de Moura and N. Bjørner. Z3: An efficient SMT solver. In C.R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 337–340, Budapest, Hungary, 2008. Springer.

[199] A. Morgenstern, M. Gesell, and K. Schneider. An asymptotically correct finite path semantics for LTL. In N. Bjørner and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 7180 of *LNCS*, pages 304–319, Mérida, Venezuela, 2012. Springer.

[200] K.S. Namjoshi and R.P. Kurshan. Efficient analysis of cyclic definitions. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 394–405, Trento, Italy, 1999. Springer.

[201] J.-P. Paris, G. Berry, F. Mignard, P. Couronne, P. Caspi, N. Halbwachs, Y. Sorel, A. Benveniste, T. Gautier, P. Le Guernic, F. Dupont, and C. Le Maire. The common format of synchronous languages: The declarative code DC, 1998.

[202] C. Passerone, L. Lavagno, M. Chiodo, and A.L. Sangiovanni-Vincentelli. Fast Hardware/Software co-simulation for virtual prototyping and trade-off analysis. In *Design Automation Conference (DAC)*, pages 389–394, Anaheim, California, USA, 1997. ACM.

[203] H.D. Patel, S.K. Shukla, and R.A. Bergamaschi. Heterogeneous behavioral hierarchy extensions for SystemC. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 26(4):765–780, April 2007.

[204] A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods (FM)*, volume 4085 of *LNCS*, pages 573–586, Hamilton, Ontario, Canada, 2006. Springer.

[205] A. Poigné and L. Holenderski. Boolean automata for implementing pure Esterel. Arbeitspapiere 964, GMD, Sankt Augustin, Germany, 1995.

[206] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Application of Concurrency to System Design (ACSD)*, pages 48–57, Saint-Malo, France, 2005. IEEE Computer Society.

[207] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. In *Application of Concurrency to System Design (ACSD)*, pages 67–76, Hamilton, Ontario, Canada, 2004. IEEE Computer Society.

[208] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design (FMSD)*, 28(2):111–130, 2006.

[209] D. Potop-Butucaru and R. de Simone. Optimizations for faster execution of Esterel programs. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 227–236, Mont Saint-Michel, France, 2003. IEEE Computer Society.

[210] D. Potop-Butucaru, R. de Simone, and Y. Sorel. Necessary and sufficient conditions for deterministic desynchronization. In C.M. Kirsch and R. Wilhelm, editors, *Embedded Software (EMSOFT)*, pages 124–133, Salzburg, Austria, 2007. ACM.

[211] M. Pouzet and P. Raymond. Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation. In S. Chakraborty and N. Halbwachs, editors, *Embedded Software (EMSOFT)*, pages 215–224, Grenoble, France, 2009. ACM.

[212] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization (SICON)*, 25(1):206–230, 1987.

[213] C.V. Ramamoorthy and H.F. Li. Pipeline architecture. *ACM Computing Surveys (CSUR)*, 9(1):61–102, 1977.

[214] S. Ramesh, S. Sonalkar, V. D'Silva, N. Chandra, and B. Vijayalakshmi. A toolset for modelling and verification of GALS systems. In R. Alur and D.A. Peled, editors, *Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 506–509, Boston, Massachusetts, USA, 2004. Springer.

[215] R. Rangan, N. Vachharajani, M. Vachharajani, and D.I. August. Decoupled software pipelining with the synchronization array. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 177–188, Antibes Juan-les-Pins, France, 2004. IEEE Computer Society.

[216] M.D. Riedel and J. Bruck. Cyclic combinational circuits: Analysis for synthesis. In *International Workshop on Logic and Synthesis (IWLS)*, Laguna Beach, California, USA, 2003.

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

[218] R.L. Rivest. The necessity of feedback in minimal monotone combinational circuits. *IEEE Transactions on Computers (T-C)*, C-26(6):606–607, 1977.

[219] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits: A hardware implementation of LUSTRE. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 195–208, Mook, The Netherlands, 1992. Springer.

[220] G. Roquier, C. Lucarz, M. Mattavelli, M. Wipliez, M. Raulet, J.W. Janneck, I.D. Miller, and D.B. Parlour. An integrated environment for HW/SW co-design based on a CAL specification and HW/SW code generators. In *International Symposium on Circuits and Systems (ISCAS)*, pages 799–799, Taipei, Taiwan, 2009. IEEE Computer Society.

[221] B.K. Rosen. High-level data flow analysis. *Communications of the ACM (CACM)*, 20(10):712–724, October 1977.

[222] J.A. Rowson. Hardware/Software co-simulation. In *Design Automation Conference (DAC)*, pages 439–440, San Diego, California, USA, 1994. ACM.

[223] I. Sander, A. Jantsch, and Z. Lu. Development and application of design transformations in ForSyDe. In *Design, Automation and Test in Europe (DATE)*, pages 10364–10369, Munich, Germany, 2003. IEEE Computer Society.

[224] A.L. Sangiovanni-Vincentelli, L.P. Carloni, F. De Bernardinis, and M. Sgroi. Benefits and challenges for platform-based design. In S. Malik, L. Fix, and A.B. Kahng, editors, *Design Automation Conference (DAC)*, pages 409–414, San Diego, California, USA, 2004. ACM.

[225] K. Schneider. A verified hardware synthesis for Esterel. In F.J. Rammig, editor, *Distributed and Parallel Embedded Systems (DIPES)*, pages 205–214, Schloß Ehringerfeld, Germany, 2000. Kluwer.

[226] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Application of Concurrency to System Design (ACSD)*, pages 143–154, Newcastle Upon Tyne, England, UK, 2001. IEEE Computer Society.

[227] K. Schneider. Improving automata generation for linear temporal logic by considering the automata hierarchy. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2250 of *LNCS*, pages 39–54, Havana, Cuba, 2001. Springer.

[228] 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, Virginia, USA, 2002. Springer.

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

[230] K. Schneider and M. Wenz. A new method for compiling schizophrenic synchronous programs. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 49–58, Atlanta, Georgia, USA, 2001. ACM.

[231] T. Schuele and K. Schneider. Exact runtime analysis using automata-based symbolic simulation. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 153–162, Mont Saint-Michel, France, 2003. IEEE Computer Society.

[232] T. Schuele and K. Schneider. Abstraction of assembler programs for symbolic worst case execution time analysis. In S. Malik, L. Fix, and A.B. Kahng, editors, *Design Automation Conference (DAC)*, pages 107–112, San Diego, California, USA, 2004. ACM.

[233] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *International Conference on Software Engineering (ICSE)*, pages 418–427, Edinburgh, Scotland, UK, 2004. IEEE Computer Society.

[234] E.M. Sentovich. Quick conservative causality analysis. In *International Symposium on System Synthesis (ISSS)*, pages 2–8, Antwerp, Belgium, 1997. IEEE Computer Society.

[235] T. Shiple, V. Singhal, R. Brayton, and A. Sangiovanni-Vincentelli. Analysis of combinational cycles in sequential circuits. In *International Symposium on Circuits and Systems (ISCAS)*, pages 592–595, 1996.

[236] T.R. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California, Berkeley, California, USA, 1996.

[237] T.R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design Automation Conference (EDAC)*, pages 328–333, Paris, France, 1996. IEEE Computer Society.

[238] G. Singh and S.K. Shukla. Algorithms for low power hardware synthesis from concurrent action oriented specifications CAOS. *International Journal of Embedded Systems (IJES)*, 3(1/2):83–92, 2007.

[239] G. Singh and S.K. Shukla. Verifying compiler based refinement of Bluespec specifications using the SPIN model checker. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *Model Checking Software (SPIN)*, volume 5156 of *LNCS*, pages 250–269, Los Angeles, California, USA, 2008. Springer.

[240] I.E. Sutherland. Micropipelines. *Communications of the ACM (CACM)*, 32(6):720–738, June 1989.

[241] J. Talpin, J. Ouy, L. Besnard, and P. Le Guernic. Compositional design of isochronous systems. In *Design, Automation and Test in Europe (DATE)*, pages 928–933, Munich, Germany, 2008. IEEE Computer Society.

[242] O. Tardieu and R. de Simone. Instantaneous termination in pure Esterel. In R. Cousot, editor, *Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 91–108, San Diego, California, USA, 2003. Springer.

[243] O. Tardieu and R. de Simone. Curing schizophrenia by program rewriting in Esterel. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 39–48, San Diego, California, USA, 2004. IEEE Computer Society.

[244] O. Tardieu and S.A. Edwards. R-SHIM: deterministic concurrency with recursion and shared variables. In J.C. Hoe and J. Palsberg, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 202–202, Napa, California, USA, 2006. IEEE Computer Society.

[245] O. Tardieu and S.A. Edwards. Scheduling-independent threads and exceptions in SHIM. In S.L. Min and W. Yi, editors, *Embedded Software (EMSOFT)*, pages 142–151, Seoul, South Korea, 2006. ACM.

[246] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[247] A.R.W. Todesco and T.H.Y. Meng. Symphony: A simulation backplane for parallel mixed-mode co-simulation of VLSI systems. In *Design Automation Conference (DAC)*, pages 149–154, Las Vegas, Nevada, USA, 1996. ACM.

[248] J.T. Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, University of Eindhoven, The Netherlands, Eindhoven, The Netherlands, 1984.

[249] K. van Berkel. *Handshake Circuits*. Cambridge University Press, Cambridge, Great Britain, 1993.

[250] N. Vasudevan and S.A. Edwards. Static deadlock detection for the SHIM concurrent language. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 49–58, Anaheim, California, USA, 2008. IEEE Computer Society.

[251] N. Vasudevan and S.A. Edwards. Buffer sharing in CSP-like programs. In R. Bloem and P. Schaumont, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 151–160, Cambridge, Massachusetts, USA, 2009. IEEE Computer Society.

[252] M. Yoeli and S. Rinon. Application of ternary algebra to the study of static hazards. *Journal of the ACM (JACM)*, 11:84–97, 1964.

[253] C. Zebelein, J. Falk, C. Haubelt, and J. Teich. Classification of general data flow actors into known models of computation. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 119–128, Anaheim, California, USA, 2008. IEEE Computer Society.

[254] J. Zeng and S.A. Edwards. Separate compilation for synchronous modules. In L.T.Yang, X. Zhou, W. Zhao, Z. Wu, Y. Zhu, and M. Lin, editors, *International Conference on Embedded Software and Systems (ICESS)*, volume 3820 of *LNCS*, pages 129–140, Xi'an, China, 2005. Springer.

[255] J. Zeng, C. Soviani, and S.A. Edwards. Generating fast code from concurrent program dependence graphs. In D. Whalley and R. Cytron, editors, *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 175–181, Washington, District of Columbia, USA, 2004. ACM.

[256] Y. Zhou and E.A. Lee. A causality interface for deadlock analysis in dataflow. In S.L. Min and W. Yi, editors, *Embedded Software (EMSOFT)*, pages 44–52, Seoul, South Korea, 2006. ACM.

[257] R. Ziller and K. Schneider. Combining supervisor synthesis and model checking. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(2):331–362, May 2005.

[258] V. Zivojnovic and H. Meyr. Compiled HW/SW co-simulation. In *Design Automation Conference (DAC)*, pages 690–695, Las Vegas, Nevada, USA, 1996. ACM.