FunState—An Internal Design Representation for Codesign

L. Thiele and K. Strehl
Computer Engineering and Networks Lab (TIK)
Swiss Federal Institute of Technology (ETH)
Zurich, Switzerland

D. Ziegenbein and R. Ernst
Institute of Computer Engineering (IDA)
Technical University of Braunschweig
Braunschweig, Germany

J. Teich
Computer Engineering Lab (DATE)
University of Paderborn
Paderborn, Germany

Abstract

In this paper, an internal design model called *FunState* (functions driven by state machines) is presented that enables the representation of different types of system components and scheduling mechanisms using a mixture of functional programming and state machines.

It is shown here how properties relevant for scheduling and verification of specification models like boolean dataflow, cyclostatic dataflow, synchronous dataflow, marked graphs, and communicating state machines as well as Petri nets may be represented in the *FunState* model. Examples of methods suited for *FunState* are described, such as scheduling and verification. They are based on the representation of the model's state transitions in form of a periodic graph.

1 Introduction

In the design of complex embedded systems, the specification of the functional and timing behavior necessitates a mixture of different basic models of computation and communication which come from transformative or reactive domains. On the other hand, we are faced with an increasing heterogeneity in the implementation.

This heterogeneity caused a broad range of allocation, binding, and scheduling policies in hardware and software implementations. Recently, a methodology has been designed to deal with the modeling problem of complex embedded systems for the purpose of scheduling [38, 39]. This model called SPI (*System Property Intervals*) is a formal design representation internal to a design system. It combines the representation of communicating processes with correlated operation modes, the representation of non-determinate behavior, different communication mechanisms such as queues and registers, and scheduling constraints.

This paper is concerned with *FunState*, a major enhancement of the SPI model. While SPI is intended to capture the semantics of multiple input languages with different semantics, *FunState* has been defined to support verification and the representation of implementation decisions, such as scheduling strategies. *FunState* is a more complex model, but it allows to explicitly model control which, in SPI, is always linked to data tokens. So, one design scenario would be to start with a SPI notation capturing the design intent and then gradually extend the SPI processes for verification and implementation. SPI and *FunState* are semantically equivalent, as will be shown in this paper, and great care has been taken that SPI and *FunState* processes and channels are interface compatible. This approach supports incremental implementation and verification. Another application of a mixed representation is the inclusion of third party or legacy system parts where control information is incomplete.

So in conclusion, *FunState* is the preferred representation wherever the control of a process shall be exposed to a tool or to the user. A good example is the application of conflict-dependent scheduling, as will be demonstrated in this paper.

The role of such an internal model in a multi-language design setting is as follows. A specification of a system consists of different input formalisms. These different parts may be modeled and optimized independently. Then the information useful for methods like allocation of resources, partitioning the design, scheduling, and verification must be estimated or extracted and mapped onto internal representations which describe properties of the subsystems and their coordination (synchronization and communication). There may be different internal models for different tasks to be performed using system analysis and design. Methods like scheduling, abstraction, and verification work on these internal representations and eventually refine them by adding components and reducing non-determinism.

To justify the approach, efficient verification and scheduling methods are described and several examples are given. The following new results are described in the paper:

- The *FunState* representation is defined which serves as an internal representation of heterogeneous embedded systems for the purpose of scheduling and verification. Extensions are provided which enable hierarchical representations and support abstraction mechanisms.
- As the *FunState* model explicitly separates control and data flow, properties of many different models of computation can be represented, such as communicating finite state machines, marked graphs, synchronous, cyclo-static, dynamic dataflow graphs, and Petri nets. In contrast to other approaches, constraints and refinements as occurring in a typical design process can be represented directly in the model. Examples are different scheduling policies such as static scheduling, quasi-static scheduling, and constant-rate scheduling.
- The methods which will be described in this paper are based on the representation of a state space in form of a regular state transition graph, i.e., the state transition graph of a regular state machine. These dynamic or periodic graphs are theoretically well investigated. The simplicity of the underlying semantics distinguishes the presented representation from other approaches.

2 Related Work

In many applications such as embedded systems, the transformative domain (data processing, stream processing) and the reactive domain (reaction to discrete events, control flow) are tightly interwoven. Application examples include mode and parameter control of dataflow processing systems, system configuration and initialization, e.g., in packet-based transmission systems [14], wireless modems [6], etc.

It is not possible to give here an overview of all specification models which have been proposed in this area. Many of them will be covered in later sections when we relate *FunState* to other models of computation. An overview and classification of different models of computation including discrete-event, reactive, and dataflow models is given in [23].

In the SPI model [38, 39], the control information is communicated using data tokens. Two similar approaches are Huss' codesign model CDM [4] and Eles' conditional process graph [10]. Many other research groups independently proposed models that separate data and control flow. These are, for example, SDL [31], codesign finite state machines (CFSMs) [2], combining *synchronous dataflow* (SDF) [25] with finite state machines (FSMs) [29, 14] and program

state machines [36]. Most of these approaches have limited composability as control and data flow cannot be mixed arbitrarily in the hierarchical levels.

Also in this area, graphical formalisms based on extensions of classical FSMs like hierarchical, concurrent FSMs as introduced by Harel [16] with many variants [26, 37] have been developed. In the implementation of i-Logix Inc., the dataflow aspect is covered in a separate domain called activity-chart. In *charts [6, 13], unlike statecharts, CFSMs, and other concurrent hierarchical FSMs, no model of concurrency is defined a priori. Instead, the goal is to show how to embed FSMs within a variety of concurrency models, i.e., dataflow models, discrete-event models, and the synchronous/reactive model.

Whereas these authors favor the systematic combination of disjoint semantics often combined with abstract graphical models (block diagrams), e.g. [6], others seek for a consistent semantics for specification of the complete system, for example the COSYMA system [12] and the OLYMPUS system [9].

Complementary to the above approaches, the *FunState* internal model attempts to reduce the design complexity by representing only those characteristics of a heterogeneous input specification which are relevant to certain design methods, in particular scheduling and verification. Therefore, the primary purpose is not to provide a unifying algorithm specification.

Besides the usual requirements for specification models such as composability, hierarchical structure, well-defined semantics and adaptation to the heterogeneity present in the application domain, we require four further properties. (1) The properties of different specification models (computer languages, block diagrams) relevant to certain design methods should be representable in the internal model. (2) The internal model must support abstraction mechanisms as necessary for the design of complex systems. (3) The internal model should support refinement such that results in the design process can be incorporated into the model, e.g., scheduling decisions reducing the degree of non-determinism or back-annotation of computation times of tasks.

3 The Basic *FunState* Model

At first, the basic non-hierarchical *FunState* model is explained. The activation of functions in a network is controlled by a finite state machine, similar to the semantics of activity-charts in statecharts implementations, see [17]. In contrast to dataflow models of computation, functions (or actors) are not autonomous.

Definition 3.1 *The basic untimed FunState component consists of a network N and a finite state machine M. The network* $N = (\mathbf{F}, \mathbf{S}, \mathbf{E})$ *itself contains a set of storage units* $s \in \mathbf{S}$*, a set of functions* $f \in \mathbf{F}$*, and a set of directed edges* $e \in \mathbf{E}$ *where* $\mathbf{E} \subseteq (\mathbf{F} \times \mathbf{S}) \cup (\mathbf{S} \times \mathbf{F})$.

Data is represented by *valued tokens*. Storage units and functions form a bipartite graph. In other words, there are no edges connecting two storage units or two functions.

Fig. 1 shows an example of a simple *FunState* model. The upper part represents the network N containing storage units q_1, q_2, q_3, and q_4 with 1, 2, 0, and 3 tokens, respectively, and functions f_1, f_2, and f_3. The lower part contains a finite state machine, in this example with just one state and three transition edges. Details concerning the behavior of the *FunState* model are described below.

3.1 Elements of the Network

3.1.1 Storage Units

For the sake of simplicity, only two sorts of storage elements are introduced here, namely queues and registers.

Queues have FIFO (first in first out) behavior and unbounded length. They store tokens which are added (removed) via incoming (outgoing) edges. The tokens represent data flowing through the network.

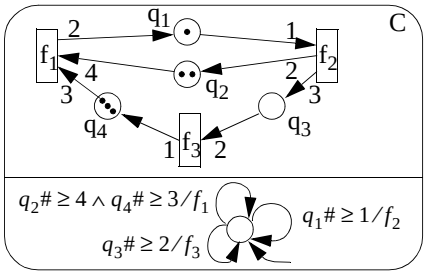

Figure 1: Example of a simple *FunState* model.

The numbers of tokens $q\# \in \mathbf{Z}_{\geq \mathbf{0}}$ in queues q are part of the system state. $q\$1$, $q\$2$, ... denote the values of the first, second, ... token in queue q, respectively. The assignment of initial values to tokens is not considered here.

Registers are linear arrays of pairs $(address, value)$. In contrast to tokens in a queue, the number of values in a register is constant. These values $r\$1$, $r\$2$, ..., $r\$n$ of a register r can be replaced via tokens on incoming edges or read non-destructively via outgoing edges. Registers are modeled in order to represent the flow of information, e.g., for estimating the necessary communication bandwidth and imposing timing constraints.

3.1.2 Functions

The function objects $f \in \mathbf{F}$ of a *FunState* model are uniquely named and operate on tokens or values when firing. Inputs and outputs of functions have associated variables $c_i \in \mathbf{Z}_{\geq \mathbf{0}}$ and $p_i \in \mathbf{Z}_{\geq \mathbf{0}}$ which denote the number of consumed tokens (read values) and the number of produced tokens (replaced values), respectively. The variables represent expressions which evaluate to constants or random processes. If required, additional constraints, for example intervals, involving these variables may restrict the numbers of consumed or produced tokens. In case of firing, the function f shown in Fig. 2 consumes c tokens from queue q_1 and reads 3 values from register r_1. It adds to q_2 some non-deterministically chosen number of tokens in the interval $[1, 4]$ and replaces p values in r_2.

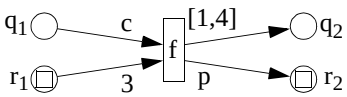

Figure 2: Example of a function.

3.2 State Machine

There are many different possibilities to specify the finite state machine M which controls the activation of embedded components or functions. In order to facilitate analysis, scheduling, and the concept of hierarchy, a synchronous/reactive model is chosen. In particular, the model is similar to ARGOS [26] developed at IMAG (Grenoble). It resembles the statecharts formalism by Harel [16, 17] but resolves circular dependencies using fixed point semantics.

Transitions are labeled with conditions and actions. Conditions are predicates on storage units $s \in \mathbf{S}$ in the network. These predicates very often only concern the number of tokens in a queue, e.g., $q\# \geq v$ for some integer variable v. Again, this variable may represent a deterministic value or a random process, possibly constrained. A transition is enabled if the corresponding predicate is *true*. The action consists of a set of the names of those functions which are activated when the transition is taken.

Fig. 3 shows the example of a simple automaton. The transition t is taken if the automaton is in its initial state, if there are at least 3 tokens in queue q, and if the value of the second token is less

than 1.5. At the same time instant, functions named f_1 and f_2 are activated.

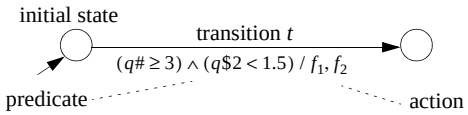

Figure 3: Part of a simple automaton M of some component.

3.3 Operational Semantics of the Flat Model

Until now, we have not described how the state machine and the network interact. At first, the current state x_c of the state machine is set to its initial state x_0. All queues and registers are filled with initial tokens and values. Then the following steps are executed in turn:
(1) Predicate evaluation: All predicates over storage units used in conditions of transitions originating from the currently active state of the state machine are evaluated. **(2) Check for possible progress:** If there is no enabled transition, i.e., no one with a satisfied predicate, the execution is stopped. **(3) State machine reaction:** One non-deterministically chosen enabled state transition is taken of which the source is the current state. All functions in the corresponding action set are activated. **(4) Function firing:** All activated functions are fired in non-deterministic order. A fired function removes tokens (reads values) from its input storage units and adds tokens (writes values) to its output storage units[1].

4 Model Extensions

In order to allow for refinement, abstraction, and hierarchy, the basic model can be extended using the following principles:

- The state machine M can be hierarchical, similar to ARGOS [26]. To this end, XOR and AND decompositions as well as signals are introduced.
- The network N can also contain embedded components and interfaces through which a component exchanges values and tokens with the component into which it is embedded. The state machine can send signals to embedded components.

The hierarchy can be nested arbitrarily deep. The semantics of the hierarchical model can be given through a simple flattening operation [35]. The concept of hierarchy does not complicate the semantics. In Fig. 4, part of a hierarchical *FunState* model and its equivalent flat model are shown.

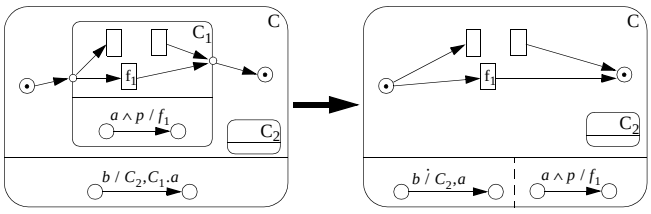

Figure 4: Part of a hierarchical *FunState* model and its equivalent flat model.

If predicate b in Fig. 4 is *true*, the state machine of C makes a transition. It also activates the components C_1 and C_2. The condition a for the transition of the state machine of C_1 is *true* also, resulting in a reaction of the state machine of C_1 in the same execution cycle if p is satisfied, too. The flat model has two parallel state machines (AND decomposition).

[1]It may happen that a function tries to remove tokens from an empty queue. There are many possibilities to deal with such a case. One may even apply the verification methods described later on in order to statically prove that this cannot happen at runtime.

5 Regular State Machines

The purpose of this section is to introduce the underlying computational model of *FunState*. It serves as the basis for the methods derived in this paper, i.e., verification and scheduling. Because of the simplicity of this model and its thorough investigation in combinatorial mathematics, many further results can be expected in the future.

The model is introduced in its simplest form. In particular, we start from the following class of *FunState* models: (1) The conditions in the *FunState* model do not contain data dependencies, i.e., the free variables in predicates denote numbers of tokens in queues only. (2) We suppose that the hierarchy of components has been unfolded using the techniques described above. (3) The functions have constant consumption and production rates c and p, respectively.

An example for the relation between a *FunState* model and its computational model is given in Fig. 5. The numbers of tokens in queues q_k correspond to the respective vector elements i_k.

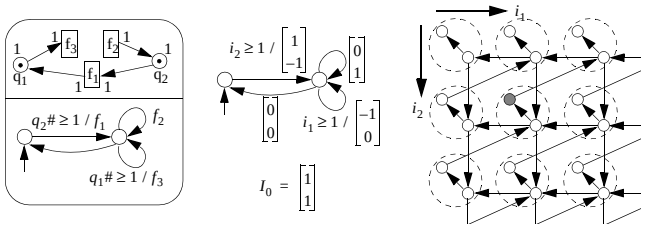

Figure 5: A basic *FunState* model, its equivalent static state transition graph, and its dynamic state transition diagram.

Definition 5.1 *A static state diagram is a directed edge-labeled graph $G = (\mathbf{V}, \mathbf{A}, D, P, v_0, I_0)$ with a set of nodes $\mathbf{V}$, a set of directed edges $\mathbf{A}$ where $a = (v_1, v_2)$ denotes an edge with source $v_1 \in \mathbf{V}$ and target $v_2 \in \mathbf{V}$, a function $D : \mathbf{A} \to \mathbf{Z}^m$ which associates to each edge $a = (v_1, v_2) \in \mathbf{A}$ an integer distance vector $d(a) = d(v_1, v_2) \in \mathbf{Z}^m$ of dimension m, a predicate function $P : \mathbf{A} \times \mathbf{Z}^m \to \{true, false\}$, and a node $v_0 \in V$ and a vector with non-negative elements $I_0 \in \mathbf{Z}^m$ which are called the initial state.*

The static state diagram as defined above is a shorthand notation for the (infinite) state transition diagram of a *regular state machine*, denoted as a dynamic state transition diagram.

Definition 5.2 *The dynamic state diagram $G_d = (\mathbf{X}, \mathbf{T}, x_0)$ of a given static state diagram $G = (\mathbf{V}, \mathbf{A}, D, P, v_0, I_0)$ is an infinite directed graph defined as follows: The nodes $\mathbf{X}$ are called the states of the regular state machine. We have $\mathbf{X} = \mathbf{V} \times \mathbf{I}$ where $\mathbf{I} = \mathbf{Z}^m_{\geq 0}$ denotes the index set of the regular state machine and $x = (v, I) \in \mathbf{X}$ denotes a state for all $v \in \mathbf{V}$ and $I \in \mathbf{I}$. The state $x_0 = (v_0, I_0)$ is the initial state. The edges $\mathbf{T}$ are called transitions of the dynamic state diagram. There is an edge $t = (x_1, x_2) \in \mathbf{T}$ with $x_1 = (v_1, I_1) \in \mathbf{X}$ and $x_2 = (v_2, I_2) \in \mathbf{X}$ iff $a = (v_1, v_2) \in \mathbf{A}$, $I_2 - I_1 = d(v_1, v_2)$, and $P(a, I_1) = true$.*

A given *FunState* model can be transformed into a static graph by a simple syntactic operation. In particular, the nodes of the finite state machine in the *FunState* model are the nodes $\mathbf{V}$, the transitions the edges $\mathbf{A}$, the predicates on the transitions are P, and the initial state is v_0. The dimension n is the number of queues in the *FunState* model, I_0 is a vector containing the numbers of initial tokens, and $d(a)$ denotes the change in the number of tokens caused by the transition corresponding to a.

The state transition diagram of a *FunState* model is given by its dynamic state transition diagram. Therefore, the *FunState* model is in state $x_0 = (v_0, I_0)$ initially. A state transition via some edge $a = (v_1, v_2) \in E$ with source v_1 and target v_2 may happen iff the state machine is in a state $x_1 = (v_1, I_1)$ for some index point I_1 and

$P(a, I_1) = true$. After the transition, the *FunState* model is in state $x_2 = (v_2, I_1 + d(a))$.

The model is similar to that of vector addition systems or Petri nets. But in our case, there are several nodes for each index point I. Moreover, many results from combinatorial mathematics are known for the class of periodic graphs considered here, e.g., [1, 19, 28].

6 Relationship to Other Models

As the *FunState* model serves as an internal representation, properties relevant to scheduling and verification of different input specifications should be easily representable.

The modeling power of *FunState* is coming neither from the concept of hierarchical or parallel automata (as they can be transformed to simple automata without signals) nor from the concept of embedded components (as they can be flattened). Instead, the partition into a purely reactive part (state machine) without computations and a passive functional part is the main source for this capability.

On the other hand, it cannot be expected that efficient analysis, code generation, and scheduling techniques exist in general. As it will be seen, the combination of embedded components, refinement and abstraction mechanisms leads to a new approach to solving these complex problems.

The following comparison may lead to useful application or domain specific restrictions of the *FunState* model. This is one of the major capabilities which leads to efficient methods for this model of computation.

6.1 Communicating Finite State Machines

Basic concepts of statechart-like [16] specifications and synchronous parallel state machines like ARGOS [26] are directly included as the *FunState* model supports AND and XOR substates. As two further examples, the communication mechanisms of the POLIS [2] model for specification and design of embedded systems and those of the communicating finite state machines are described in some detail.

Communicating Finite State Machines In the case of a communicating finite state machine, e.g., as in SDL process networks [31], asynchronously operating finite state machines communicate via FIFO ordered queues. An FSM M_1 can write a value into a queue q during a transition. An FSM M_2 can guard its transitions with predicates on the value of the first element in the queue $q\#1$. If the transition is taken, the element is removed from the queue.

In the *FunState* model, the finite state machines can be embedded into components, e.g., C_1 and C_2 for M_1 and M_2. Writing into and removing from queues can be modeled using functions, e.g., f_1 and f_2, with production and consumption rates of 1. The asynchronous reactions of the finite state machines are implemented using a finite state machine M in the top component with one state and loops for each finite state machine, see Fig. 6.

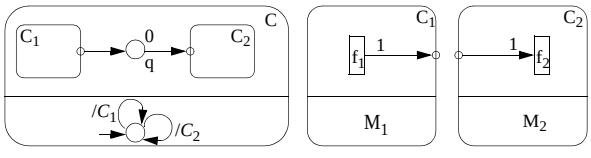

Figure 6: Representation of communicating finite state machines in the *FunState* model.

POLIS The POLIS model [2] has been invented for designing control dominated embedded systems. Here, it will be shown how the communication mechanism can be represented in the *FunState* model. All finite state machines (FSMs) operate asynchronously, here M_1 and M_2. They communicate via single element buffers, e.g., q. If an FSM M_1 writes into this buffer, the old value is replaced by a new one. Reading from the buffer is non-destructive. This communication model can be represented as shown in Fig. 7.

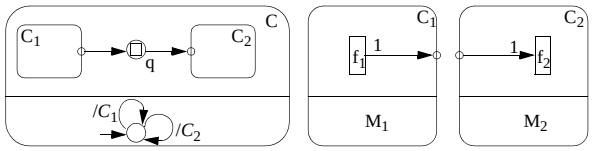

Figure 7: Representation of the POLIS model.

6.2 Marked Graphs and Synchronous Dataflow Graphs

Marked graphs [7] and synchronous dataflow (SDF) graphs [24, 25] are labeled directed graphs with nodes representing the actors of the system and edges denoting the communication and the corresponding queues between the actors. A function m determines for each edge the number of initial tokens in the corresponding FIFO queue. Two functions c and p denote the numbers of tokens removed from the queue if the actor at its target fires and the number of tokens added to the queue if the actor at its source fires, respectively. An actor may fire if in its input queues e there are at least $c(e)$ tokens. For marked graphs, we have $c(e) = p(e) = 1$ for all edges e.

A *FunState* model which behaves like an SDF graph can be constructed as follows. The whole graph is embedded into one component. Each actor is replaced by a function, each edge is replaced by a concatenation of an edge, a queue, and another edge. The initial numbers of tokens in these queues are determined by the initial numbers of tokens on the edges of the SDF graph. The values $c(e)$ and $p(e)$ are written at the corresponding incoming and outgoing edges of the functions. The state machine of the component has one state and one loop transition for each actor. The condition of each loop is the firing condition of the corresponding data flow actor ($e\# \geq c(e)$ for all input queues e), and the action is an activation of the corresponding function. Thus, although the actors are no longer autonomous, no further constraints on the model execution have been added.

Fig. 1 shows a *FunState* model corresponding to the SDF graph shown in Fig. 8. This is constructed as above and is an example of a *global control strategy*. An example of a model with a *local control strategy* is shown in Fig. 8. In the following sections, this strategy will be used to represent cyclo-static, Boolean, and dynamic actors.

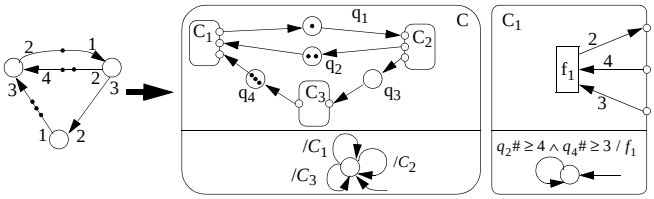

Figure 8: Example of a synchronous dataflow graph and its representation as a *FunState* model with local control. Only embedded component C_1 is shown.

6.3 Cyclo-Static Dataflow Graphs

In cyclo-static dataflow [3, 11], production and consumption rates of actors change periodically. Fig. 9 shows a cyclo-static actor and the corresponding *FunState* component. The different communication behaviors of the cyclo-static actor are represented by separate functions in the *FunState* component. The state machine of the *FunState* component cycles through all possible consumption and production

rates by cyclically activating the corresponding functions. The *FunState* components representing the actors are connected as in Fig. 8.

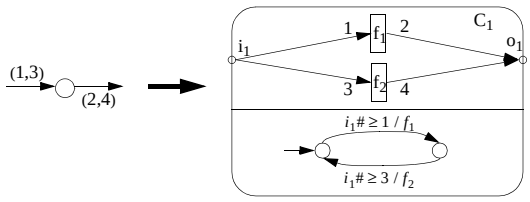

Figure 9: A cyclo-static dataflow node.

6.4 Boolean and Dynamic Dataflow Graphs

Boolean and dynamic dataflow graphs extend the previously described SDF model by introducing data dependent dataflow. In particular, in the BDF model, two additional types of nodes called SELECT and SWITCH are defined, see Fig. 10. SWITCH is enabled if the data input edge i and the control input edge c contain at least one token. Once enabled, the node decides based on the value $c\$1 \in \{true, false\}$ of the first token to which output o_1 or o_2 the first token on the data input edge is transferred. The SELECT node acts similarly, i.e., a token on either input i_1 or input i_2 is transferred to output o if there is a token on c with value $c\$1 = true$ or $c\$1 = false$, respectively.

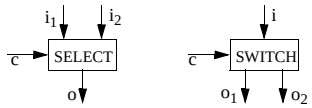

Figure 10: SELECT and SWITCH nodes in Boolean dataflow graphs.

Fig. 11 shows the corresponding *FunState* models. The conditions are defined as

$$
\begin{aligned}
c_1: &\quad i_1\# >= 1 \wedge c\# >= 1 \wedge c\$1 = true \\
c_2: &\quad i_2\# >= 1 \wedge c\# >= 1 \wedge c\$1 = false \\
c_3: &\quad i\# >= 1 \wedge c\# >= 1 \wedge c\$1 = true \\
c_4: &\quad i\# >= 1 \wedge c\# >= 1 \wedge c\$1 = false
\end{aligned}
$$

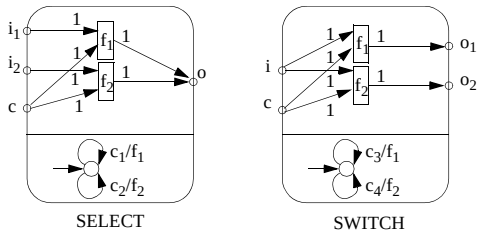

Figure 11: SELECT and SWITCH nodes in the *FunState* model.

As an example of a node type defined in dynamic dataflow graphs, Fig. 12 shows a non-deterministic merge node and its equivalent *FunState* model. A MERGE node is enabled for firing if at least one input edge contains at least one token. The node selects non-deterministically which token is transferred to the output.

6.5 Petri Nets

At a first glance, the *FunState* model seems to be almost equivalent to colored Petri nets (CPN) [18]. But there are several major differences which as well tune the Petri net model to the application domain of the *FunState* model and at the same time generalize it.

The queues can be related to places in Petri nets. But queues in the *FunState* model have a FIFO behavior whereas this is not the

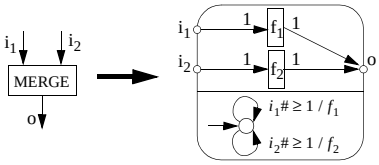

Figure 12: MERGE node in the *FunState* model.

case in CPN. This restriction matches the modeling power necessary for embedded systems and simplifies the operational semantics to a great extent.

Usually, there are no registers defined in CPN. In order to model the usual mechanism of passing values through writing and reading of variables, this capability has been added.

The activation and firing conditions are more general than in CPN as arbitrary predicates on the queues in the preset of a function can be used. Moreover, in the *FunState* model, these predicates can be different from the number of tokens removed while firing.

In a CPN, the transitions are continuously ready for being activated. In the *FunState* model, this can be controlled by the finite state machine. This capability enables the simple consideration of limited resources and scheduling policies.

7 Relationship between *FunState* and SPI

In contrast to *FunState*, SPI does not explicitly separate between control and data flow. Although SPI processes may have internal data and thus an internal state [38], this state is not explicitly represented and thus not visible. Differences in a SPI process external behavior due to state-dependencies are modeled by uncertainty intervals. Even the refinement of process behavior using process modes [39] does not have a notion of state since the execution mode of a process is determined only based on the contents of incoming channels and is "forgotten" at completion of execution. Thus, with the existing set of constructs[2], the state of a SPI model is only composed of the channel contents (amounts of tokens and mode tags). *FunState* refines the SPI model by adding the capability of explicitly modeling state information and control flow separately from data flow.

In the following, it is shown how both models correspond, and translation rules are given and explained by means of simple examples. Since in this paper only an untimed version of *FunState* is presented, timing is ignored for SPI elements as well.

The most important difference between *FunState* and SPI is the control strategy. While SPI processes are autonomous like actors in dataflow models of computation, *FunState* functions and (embedded) components are controlled by a state machine. Due to state machines in *FunState*, it is not generally possible to represent every *FunState* model with SPI[3]. On the other hand, the representation of SPI models in *FunState* is generally possible and equivalent to the representation of dataflow models using a local control strategy (see Fig. 8).

Straight forward correspondences exist for the directly equivalent storage elements in *FunState* and SPI. Also, *FunState* functions and SPI processes without modes and hierarchy directly correspond. In the following, it is shown how a SPI process can be represented by a *FunState* component and vice versa.

A SPI process can be directly represented by a *FunState* component having a state machine with a single state and several loop transitions that all start and end in this state. The actions of these transitions trigger functions in the dataflow network representing the modes of the corresponding SPI process. The condition of each

[2] excluding function variants and configurations as proposed in [30]

[3] It is possible to explicitly model the state machine by a process that controls the execution of each element of the dataflow network. But the synchronous semantics is lost when doing this

transition can be extracted from the activation function of the SPI process by combining the conditions of the rules mapping to the respective mode. A potential uncertainty in the mode selection of a SPI process resulting in a set of possible modes is equivalent to the possible non-determinism in the state machine of a *FunState* component. This analogy is shown for an example in Fig. 13 where M is the set of modes, A is the activation function for process P, and C_P is the *FunState* component representing process P.

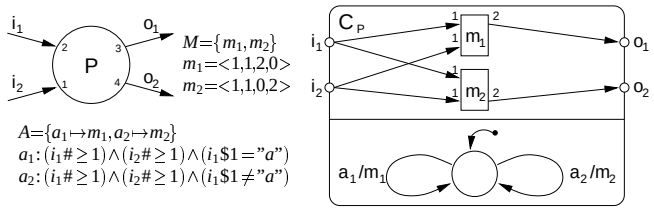

Figure 13: Translation of a SPI process into a *FunState* component.

For the translation of a *FunState* component into a SPI process, there are two different strategies. One approach is to abstract the *FunState* component such that it complies with the above component template that can be easily translated into a SPI process. In the general case, this abstraction of the *FunState* component involves loss of information due to the necessary state reduction in the component's state machine.

The other approach is to model the state-dependent behavior of the *FunState* component in SPI. This can be achieved by using virtual feedback channels for the SPI process that shall represent a *FunState* component. So the SPI process can change the state information as well as use it for adapting its behavior accordingly.

The state of a *FunState* component is composed of the state of its state machine and the contents of its internal storage elements. Due to the unbounded FIFO queues this results in an infinite state space that cannot be visualized using a single feedback channel since there is only a finite mode tag set to encode the state. Thus, one virtual channel is used for encoding the states of the *FunState* component's state machine using mode tags. Additionally, for each internal storage element that is contained in a predicate of the component's state machine, a virtual feedback channel is added to the corresponding SPI process. Then, each transition in the *FunState* component's state machine can be represented by a mode of the SPI process. The behavior and activation rules of this mode can be directly derived from the triggered actions and the predicates, respectively.

8 Applicable Methods

The purpose of this section is to show the versatility of the *FunState* model by the use of examples for its application. Again, we would like to emphasize that *FunState* essentially is used as an internal representation model during the design phase, e.g., for HW/SW codesign.

8.1 Formal Verification

There are many different purposes of formal verification of an internal design representation. Instead of dealing directly with the system specification, properties of a representation can be checked which is the basis for design steps like scheduling, binding, and allocation. It is possible to verify certain properties of a partially completed design. For example, one may want to prove that a chosen schedule results in a deadlock-free implementation or necessitates only a bounded amount of memory.

The proposed verification strategy for *FunState* models is based on their representation in form of regular state machines, see Section 5. Of course, during the verification, the state space is not enumerated explicitly.

The efficient verification of process models like *FunState* leads to a number of problems. Here, a new approach based on *interval diagram techniques* is described based on previous results described in [33]. These results are extended from simple process networks towards the more complex *FunState* model containing both finite-state control components and infinite-state dataflow queues. In particular, the verification consists of four steps. (1) Formulation of the verification goal by means of a *computation tree logic* (CTL) formula, e.g., [27]. (2) Representation of the state transition relation in form of an *interval mapping diagram* (IMD). (3) Representation of state sets as *interval decision diagrams* (IDDs). (4) Applying to IDDs Boolean operations, quantification, and state set mapping using an IMD until a fixpoint is reached according to the verification goal.

Interval diagram techniques have shown to be convenient for formal verification of, e.g., process networks [33], Petri nets, and timed automata. This new approach remedies some deficiencies of traditional *symbolic model checking* [27] approaches based on *binary decision diagrams* (BDDs) and provides advantages with regard to computation time and memory resources.

Consider the example *FunState* model of Fig. 1. To show that q_2 may never contain more than 4 tokens, the CTL formula $\mathsf{AG}\,(q_2\# \leq 4)$ can be checked. As this formula evaluates to *true*, it is proven that the memory required for q_2 is bounded by 4. Another simple example is the formula $\mathsf{AG\,EF}\,(q_1\# \geq 1)$ which means that it is always possible to reach a system state which allows f_2 to be executed. Thus, such formulae can be used to prove the absence of deadlocks.

Apart from this, formal verification may assist during the development of scheduling policies. The system model may be extended to describe one or several dynamic or hybrid scheduling policies, too, of which the behavior is verified together with the system model. Thus, common properties such as the correctness of a schedule may be affirmed by proving the boundedness of the required memory and the absence of artificial deadlocks as described above. The verification procedure for *FunState* models has been implemented, and its efficiency in comparison to other state set representations has been shown using several examples [33].

8.2 Representing Schedules

First, we describe the use of *FunState* as a representation model for several classes of scheduling policies. Afterwards, a methodology is sketched how to symbolically determine a partially static schedule of a *FunState* model.

In a hierarchical approach to solve complex scheduling problems it is necessary that the result of partially scheduling components can be represented in the same model. With this information, further scheduling steps can be performed. This stepwise refinement corresponds to the stepwise reduction of the non-determinism in the model. Due to the lack of space, we have to concentrate on one scheduling example. Further mechanisms which may be represented by *FunState* models are shown in [35].

8.2.1 Static Scheduling

As a first example we consider a purely static periodic schedule of the synchronous dataflow graph shown on the left-hand side of Fig. 8 for a uni-processor system. Methods to construct such a schedule are well-known and will not be repeated here.

The chosen schedule executes the functions f_1, f_2, and f_3 iteratively in the following order: $(f_2, f_3, f_1, f_2, f_3, f_3)$. In comparison with Fig. 1, only the state machine of the component C must be changed in order to represent the schedule. Fig. 14 shows two different possibilities, both reflecting the periodic schedule described above. The second possibility takes into account that the subsequence (f_2, f_3) occurs twice in the schedule and uses the AND

composition facility of parallel state machines.

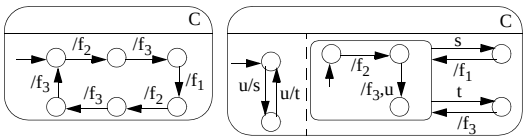

Figure 14: Two possibilities for static periodic scheduling.

8.3 Symbolic Scheduling

In addition to formal verification as described above, symbolic methods may be used not only to analyze but even to develop scheduling policies for *FunState* models. A symbolic scheduling method for heterogeneous embedded systems represented by *FunState* models has been introduced in [34]. Symbolic scheduling methods often outperform both ILP and heuristic methods while yielding exact results.

To overcome drawbacks of either purely *static* or *dynamic* scheduling approaches and to combine their advantages, Lee proposed a technique called *quasi-static* scheduling [21]. Similarly to static scheduling, most of the scheduling decisions are made during the design process, providing only few run-time overhead and partial predictability. Only data-dependent choices—depending on the value of the data or resulting from a reactive, control-oriented behavior—have to be postponed until run time. Techniques related to quasi-static scheduling have been developed using, e.g., constraint graphs [20, 8], dynamic dataflow graphs [5], actors with data-dependent execution times [15], free-choice Petri nets [32], and *FunState* models [34].

8.3.1 Conflict-Dependent Scheduling

Problems which are typical for the design of complex embedded systems are, e.g., different kinds of non-determinism such as partially unknown specification (to be resolved at design time), data-dependent control flow (to be resolved at run time), or unknown scheduling policy (to be resolved at compile time), and dependencies between design decisions for different system components. These properties necessitate new scheduling approaches as the number of execution paths to be considered grows exponentially with increasing degrees of non-determinism. Moreover, the complexity of the models of computation and communication greatly increases the danger of system deadlocks or queue overflows, see, e.g., [22].

The new scheduling method named *conflict-dependent scheduling* [34] is able to deal with mixed data/control flow specifications and takes into account different mechanisms of non-determinism as occurring in the design of embedded systems. It guarantees to find a deadlock-free and bounded schedule if one exists. The generated schedule consists of statically scheduled blocks which are dynamically called at run time.

Applying conflict-dependent scheduling to a *FunState* model may be regarded as an example of a refinement step using *FunState* as an internal design representation. The specification as well as the result of the scheduling procedure are represented as *FunState* models. The scheduling method proceeds as follows. (1) The basis is a *FunState* model which specifies all possible schedules by means of non-determinate transition behavior—representing all design alternatives. (2) By symbolic exploration of the resulting regular state machine, the state space is traversed to search for cycles representing valid schedules. This is motivated by the fact that after having traversed a cycle in the dynamic state transition diagram, an already visited state is reached for which the scheduling behavior is known. (3) The extracted schedule is transformed into a finite state machine which then is compacted using state minimization techniques. (4) Finally, the result is embedded in the original *FunState* model by replacing the schedule specification part. Furthermore, it may be transformed into program code.

8.3.2 Molecular Dynamics Simulation Example

The scheduling methodology is explained with the following example. The introduced approach has been applied to perform conflict-dependent scheduling for a molecular dynamics simulation system [34]. As shown in Fig. 15, the simplified fundamental algorithm has been mapped onto a host workstation (*Host*) linked to a special purpose hardware accelerator serving as a coprocessor (*CoPro*).

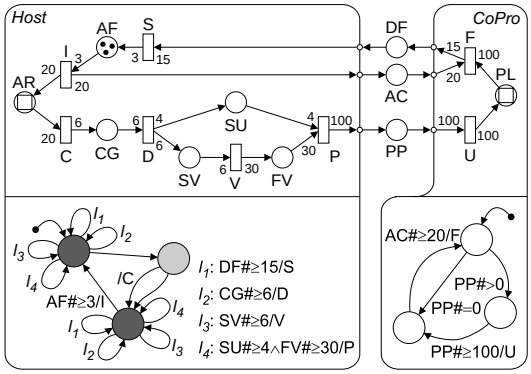

Figure 15: Molecular dynamics model with schedule specification.

The simulation mainly consists of repeated computations in the feedback loop distributed among both processors where atom forces (*AF*) are computed (*F*), added up (*S*), and integrated (*I*) to calculate new atom coordinates (*AC*, *AR*). After a variable number of iterations, the central coordinates of slowly moving sub-molecules called charge groups (*CG*) are updated (*C*). Then, a new list of neighbors called pair list (*PL*) is computed (*D, V, P, U*). The state machine of component *Host* describes a specification of possible schedules for *Host* (item 1 of the above methodology description).

The moment when to start the pair list computation is unknown until run time. This fact represents a *conflict* which is modeled using a light-shaded conflict state. Conflicts can be resolved only at run time, hence, no design decision is possible. Conflicts occur, for instance, when decisions depending on the value of data or environmental circumstances have to be taken. In contrast to this, all transitions starting in a dark-shaded state represent design *alternatives* which may be chosen during schedule development. White states in the *FunState* model are states which either have only one outgoing transition or of which all transitions have disjoint predicates. Thus, the transition behavior of these states is *determinate*. Note that in component *CoPro* the state with two outgoing transitions is determinate for this reason.

Intuitively, the symbolic techniques for conflict-dependent scheduling as proposed in [34] replace dark-shaded states by white states—taking decisions and thus removing design alternatives (item 2). The result is the *schedule controller automaton* shown in Fig. 16 (item 3) which may replace the automaton of component *Host* of Fig. 15 for analysis or synthesis purposes (item 4). It consists of two static cycles and a conflict state switching between them. The schedule is respecting the specification of *CoPro*. Note that even the schedule of *CoPro* is not static as it depends on the content of queue *PP*. For implementational efficiency, the original comparison $DF\# \geq 15$ has been automatically replaced by the non-zero test $DF\# > 0$.

The controller automaton can easily be transformed into program code as shown in Table 1 as pseudo code. The predicate *p* identifies the run-time decision associated to the conflict node.

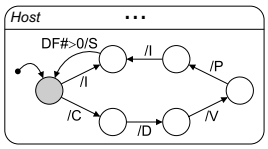

Figure 16: Resulting controller automaton.

```
a: if p then
       C; D; V; P;
   I;
   while DF# = 0 nop;
   S;
   goto a;
```

Table 1: Controller program code.

9 Summary and Conclusion

As has been explained in this paper, the *FunState* model enables the internal representation of complex system behavior. To cope with the design complexity, a hierarchical step-by-step approach is advertised and supported by the *FunState* model. The approach can be interpreted as a stepwise reduction of the non-determinism in a system specification.

In the present paper, only the basic untimed semantics of *FunState* is described. Extensions towards timed functions, the representation of timing constraints and timing properties can be found in [35].

References

[1] W. Backes, U. Schwiegelshohn, and L. Thiele. Analysis of free schedule in periodic graphs. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 333–342, San Diego, CA, USA, June 1992.

[2] F. Balarin, A. Jurecska, and H. Hsieh et al. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, Boston, 1997.

[3] G. Bilsen, P. Wauters, M. Engels, R. Lauwereins, and J. Peperstraete. Development of a static load balancing tool. In *Proc. of the fourth Workshop on Parallel and Distr. Processing*, pages 179–194, Sofia, Bulgaria, 1993.

[4] W. Boßung, S. A. Huss, and S. Klaus. High-level embedded system specifications based on process activation conditions. *Accepted for publication in the Journal of VLSI Signal Processing, Special Issue on System Design, Kluwer Academic Publishers*, 1999.

[5] J. T. Buck. Scheduling dynamic dataflow graphs with bounded memory using the Token Flow Model. Technical Report UCB/ERL 93/69, Ph.D dissertation, Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A., 1993.

[6] W.-T. Chang, A. Kalavade, and E. A. Lee. Effective heterogeneous design and cosimulation. In *G. De Micheli and M. Sami (eds.), Proc. of the NATO/ASI Workshop on Hardware/Software Co-design*, pages 187–212, Tremezzo, Italy, 1995. Kluwer Academic Publihers, 1995.

[7] F. Commoner and A.W. Holt. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[8] M. Cornero, F. Thoen, G. Goossens, and F. Curatelli. Software synthesis for real-time information processing systems. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 260–279. Kluwer Academic Publishers, 1995.

[9] G. De Micheli, D. Ku, F. Mailhot, and T. Truong. The olympus synthesis system. In *IEEE Design and Test of Computers*, 1990.

[10] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE98)*, pages 132–138, 1998.

[11] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete. Cyclo-Static Data Flow: Model and implementation. In *Proc. 28th Asilomar Conf. on Signals, Systems, and Computers*, pages 503–507, Pacific Grove, CA, 1994.

[12] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, pages 64–75, December 1993.

[13] A. Girault, B. Lee, and E. A. Lee. A preliminary study of hierarchical finite state machines with multiple concurrency models. Technical Report UCB/ERL M97/57, Electronics Research Laboratory, College of Engineering, Univ. of Cal. at Berkeley, 1997.

[14] T. Grötker, R. Schoenen, and H. Meyr. Pcc: A modeling technique for mixed control/data flow systems. In *Proc. of the European Design and Test Conference (ED&TC 97)*, 1997.

[15] S. Ha and E.A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers*, 46(7):768–778, July 1997.

[16] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[17] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM Trans. Soft. Eng. Method*, 5(4), October 1996.

[18] K. Jensen. Colored Petri Nets: A high level language for system design and analysis. In *Advances in Petri Nets 1990, G. Rozenberg (ed), Lecture Notes in Computer Science, Springer, LNCS 483*, 1990.

[19] S. R. Kosaraju and G. F. Sullivan. Detecting cycles in dynamic graphs in polynomial time (preliminary version). In *20th Annual ACM Symposium on Theory of Computing*, pages 398–406, 1988.

[20] D. C. Ku and G. De Micheli. Relative scheduling under timing constraints: algorithms for high-level synthesis of digital circuits. *IEEE Transactions on Computer-Aided Design*, 11(6):696–718, June 1992.

[21] E. A. Lee. Recurrences, iteration, and conditionals in statically scheduled block diagram languages. In R. W. Brodersen and H. S. Moscovitz, editors, *VLSI Signal Processing III*, pages 330–340. IEEE Press, New York, 1988.

[22] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, 1995.

[23] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12):1217–1229, 1998.

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

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

[26] F. Maraninchi. Argonaute: graphical description, semantics, and verification of reactive systems by using a process algebra. In *Proc. Int. Workshop on Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science, Springer, LNCS 407, 1989.

[27] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[28] J. Orlin. Some problems in dynamic and periodic graphs. In W.R. Pulleyblank, editor, *Progress in Combinatorial Optimization*, pages 215–225. Academic Press, Orlando, Florida, 1984.

[29] M. Pankert, O. Mauss, S.Ritz, and H. Meyr. Dynamic data flow and control flow in high level dsp code synthesis. In *Proc. 1994 IEEE Int. Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 449–452, April 1994.

[30] K. Richter, D. Ziegenbein, R. Ernst, J. Teich, and L. Thiele. Representation of function variants for embedded system optimization and synthesis. In *Proceedings of the 36th Design Automation Conference (DAC '99)*, New Orleans, June 1999.

[31] R. Saracco, J. R. W. Smith, and R. Reed. *Telecommunications systems engineering using SDL*. North-Holland, Elsevier, Amsterdam, 1989.

[32] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Quasi-static scheduling of embedded software using free-choice Petri nets. In *Proceedings of the Workshop on Hardware Design and Petri Nets (HPWN '98)*, 1998.

[33] K. Strehl and L. Thiele. Symbolic model checking of process networks using interval diagram techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, pages 686–692, San Jose, California, November 8–12, 1998.

[34] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and J. Teich. Scheduling hardware/software systems using symbolic techniques. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES'99)*, Rome, Italy, May 1999.

[35] L. Thiele, J., M. Naedele, K. Strehl, and D. Ziegenbein. *FunState*—functions driven by state machines. Technical Report TIK-33, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, January 1998.

[36] F. Vahid, S. Narayan, and D.D. Gajski. SpecCharts: a VHDL frontend for embedded systems. *IEEE Transactions on CAD for Integrated Systems*, 14(6):694–706, 1995.

[37] M. von der Beeck. A comparison of statecharts variants. In *Proc. Formal Techniques in Real Time and Fault Tolerant Systems*, pages 128–148, Lecture Notes in Computer Science, Springer, LNCS 863, 1994.

[38] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Combining multiple models of computation for scheduling and allocation. In *Proceedings of the 6th International Workshop on Hardware/Software Codesign (Codes/CASHE '98)*, pages 9–13, Seattle, Washington, March 1998.

[39] D. Ziegenbein, K. Richter, R. Ernst, J. Teich, and L. Thiele. Representation of process mode correlation for scheduling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, San Jose, California, November 8–12, 1998.