# *FunState*—An Internal Design Representation for Codesign

Karsten Strehl, *Member, IEEE,* Lothar Thiele, *Member, IEEE,*
Matthias Gries, *Student Member, IEEE,* Dirk Ziegenbein, *Member, IEEE,*
Rolf Ernst, *Member, IEEE,* and Jürgen Teich, *Member, IEEE*

*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 how properties relevant for scheduling and verification of specification models such as Boolean dataflow, cyclostatic dataflow, synchronous dataflow, marked graphs, and communicating state machines as well as Petri nets can be represented in the *FunState* model of computation. 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. The feasibility of the novel approach is shown with an ATM switch example.**

*Index Terms*— **Formal verification, high-level synthesis, internal specification model, model of computation, symbolic scheduling,**

## I. Introduction

One of the major sources of complexity in the design of embedded systems is related to their heterogeneity. On the one hand, 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 not only concerns the functional units which may be implemented in form of dedicated or programmable hardware, microcontrollers, domain specific or even general purpose processors. In addition, these units communicate with each other via different media, e.g., busses, memories, networks, and by using many different synchronization mechanisms.

This heterogeneity caused a broad range of scheduling policies in hardware and software implementations. Two extreme possibilities are static schedules such as those developed for *synchronous dataflow* (SDF) models [1], and EDF (*earliest deadline first*) schedules developed for dynamically changing task structures. Many intermediate possibilities have been developed over the years.

Recently, a methodology has been designed to deal with the modeling problem of complex embedded systems for the purpose of scheduling [2], [3]. 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.

The present paper is concerned with major refinements of the SPI model in order to allow the explicit modeling of mixed control and data flow within components. This enables the representation of scheduling mechanisms as well as efficient methods for scheduling and verification of system properties. *FunState* has been defined [4] to represent many different well-known models of computation, to support stepwise refinement and hierarchy, and to be suited to internally represent many different synchronization, communication, and scheduling policies. Another application of a mixed representation is the inclusion of third party or legacy system parts where control information is incomplete. In conclusion, *FunState* is a preferred representation wherever the control of a process shall be exposed to a tool or to the user. A good example is a scheduling method demonstrated in this paper.

The role of such an internal model in a multi-language setting is shown in Fig. 1. 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 such as allocation of resources, partitioning the design, scheduling, and verification must be estimated or extracted and mapped to 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. As pointed out already, this is one major stage where the need for a sound model of computation exists. Methods such as scheduling, abstraction, and verification work on these internal representations and eventually refine them by adding components and reducing non-determinism.

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.

K. Strehl is with ETAS GmbH, Research & Development, Borsigstraße 10, 70469 Stuttgart, Germany (e-mail: strehl@computer.org).

L. Thiele and M. Gries are with the Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Gloriastrasse 35, 8092 Zurich, Switzerland (e-mail: thiele@tik.ee.ethz.ch; gries@tik.ee.ethz.ch).

D. Ziegenbein and R. Ernst are with the Institute of Computer and Communication Network Engineering (IDA), Technical University of Braunschweig, Hans-Sommer-Straße 66, 38106 Braunschweig, Germany (e-mail: ziegenbein@ida.ing.tu-bs.de; ernst@ida.ing.tu-bs.de).

J. Teich is with the Computer Engineering Lab (DATE), University of Paderborn, Warburgerstraße 100, 33098 Paderborn, Germany (e-mail: teich@date.uni-paderborn.de).

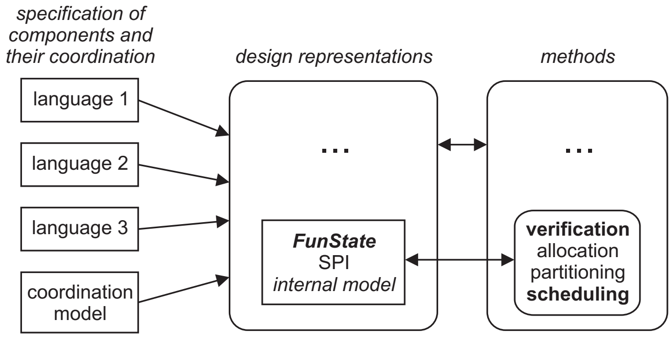



Fig. 1. Role of *FunState* in a design process.

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 timing constraints or timing properties and 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* (RSM) [5]. These dynamic or periodic graphs are theoretically well investigated. The simplicity of the underlying semantics distinguishes the presented representation from other approaches.

Before introducing the basic *FunState* model in Section III and model extensions enabling hierarchical specifications in Section IV, we give an overview of related other approaches to modeling mixed control and data flow (Section II). Then, in Section V the semantics of the *FunState* model is explained using regular state machines, a formal model that uses a periodic graph to describe the state transition behaviour of a *FunState* model. In Section VI, we explain the relationship between *FunState* and other models of computations. Finally, efficient verification and scheduling methods are described in Section VII. The paper concludes with a larger case study, namely specification and scheduling of an ATM switch, in Section VIII.

## II. 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 [6], wireless modems [7], 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 [8].

In the SPI model [2], [3], the control information is communicated using data tokens. Two similar approaches are Huss' codesign model CDM [9] and Eles' conditional process graph [10]. Many other research groups independently proposed models that separate data and control flow. These are, for example, SDL [11], codesign finite state machines (CFSMs) [12], combining synchronous dataflow (SDF) [1] with finite state machines (FSMs) [13], [6] and program state machines [14]. 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 [15] with many variants [16], [17] have been developed. In the implementation of i-Logix Inc., the dataflow aspect of Statemate is covered in a separate domain called activity-chart. But similarly to the Stateflow model employed in the Matlab/Simulink environment, the expressiveness of this part of the model prohibits the development of efficient verification and implementation methods. For example, *FunState* does not allow global variables which makes the model modular. The state machine may only interrogate events that are local to the component. In *FunState*, the dataflow network is also targeted very much to modeling well known dataflow process networks including synchronous dataflow, cyclo-static dataflow, and others used in many design systems for developing and prototyping DSP algorithms. Finally, activities in Statemate have no explicit notion of the consumption of time. This is not natural since activities, once evoked may take longer than one state machine transition. In *FunState*, functions may explicitly consume time before finishing. Therefore, the timing model is much more natural for computation-intensive activities than the single-step synchronous reactive processing in the semantics of Statemate.

In *charts [7], [18], 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., [7], others seek for a consistent semantics for specification of the complete system, for example the COSYMA system [19] and the OLYMPUS system [20].

While in the unified approach, the major problems deal with the challenge of how to extract portions of a design in order to be able to apply efficient analysis and synthesis techniques to portions of the specification, a major problem in the mixed approach lies in finding clean ways to combine diverse models of computation, at various levels of abstraction.

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. (4) It should be possible to incorporate design constraints such as required timing properties.

## III. 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 [21]. In contrast to dataflow models of computation, functions (or actors) are not autonomous.

*Definition III.1:* The basic *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. 2 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.

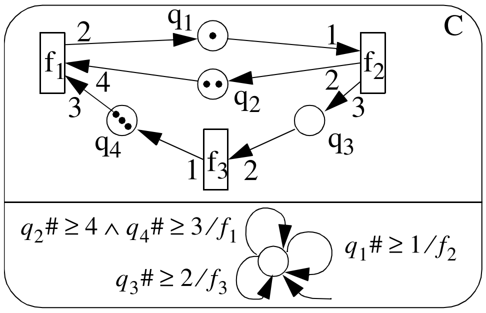



Fig. 2. Example of a simple *FunState* model.

### A. Elements of the Network

#### A.1 Storage Units

For the sake of simplicity, only two sorts of storage elements are introduced here, namely queues and registers. The actual access functionality and the available query functions on storage unit types can be defined individually for each type. Note that only examples are given here.

- **Queues**
  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. The numbers of tokens $q\# \in \mathbf{Z}_{\geq 0}$ in queues $q \in S$ are part of the system state. $q\#0 \in \mathbf{Z}_{\geq 0}$ denotes the initial number of tokens, see Fig. 3. Depending on the abstraction level, we may deal with colored tokens, i.e., tokens with values associated. In this case, $q\$1$, $q\$2$, ... , $q\$k$ with $k = q\#$ denote the values of the first, second, ... , $k$th token in queue $q$, respectively. The assignment of initial values to tokens is not considered here.

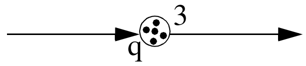



Fig. 3. Example of a queue $q$ with $q\#0 = 3$, i.e., 3 initial tokens. The current number of tokens is $q\# = 5$.

- **Registers**
  Registers are linear arrays of limited length $n$ of pairs $(address, value)$ of addresses and values. 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, see Fig. 4. In comparison with queues, registers do not impose a partial ordering on function evaluations. Registers are used for modeling the flow of information, e.g., in order to estimate the necessary communication bandwidth or impose timing constraints. The assignment of initial values to tokens is not considered here.

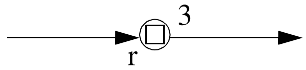



Fig. 4. Example of a register $r$ of size $n = 3$.

#### A.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 $c_i \in \mathbf{Z}_{\geq 0}$ and $p_i \in \mathbf{Z}_{\geq 0}$ 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. Similar functions can be defined which denote the values of the produced tokens.

With each function object there is associated a latency function *lat* which evaluates to a constant or a random process as in the case of the number of consumed and produced tokens.

A function object is in one of the states *idle* or *run*. Initially, the state of a function object is *idle*. In addition, the state of a function object comprises a value $\tau \geq 0$ which denotes the remaining execution time. If a function object receives an event (this process is described later in this paper), it changes state from *idle* to *run*, consumes tokens (reads values) from its input storage units and initializes $\tau = lat$. As described in the operational semantics section, the state of a function object changes from *run* to *idle* and changes are made in the output storage units when the function object finishes execution, i.e., the remaining execution time is $\tau = 0$.

When receiving an event, the function $f$ shown in Fig. 5 consumes $c$ tokens from queue $q_1$ and reads 3 values from register $r_1$. After the latency has expired, $f$ adds to $q_2$ some non-deterministically chosen number of tokens in the interval $[1, 4]$ and replaces $p$ values in $r_2$.

More general models of function objects are possible, e.g., having more states and understanding named events such as *start*, *kill*, *stop*, and *remove*. This way, different kinds of interrupts can easily be specified.

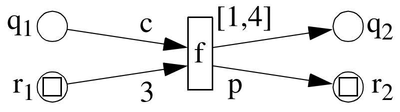



Fig. 5. Example of a function.

## B. State Machine

There are many different possibilities to specify the finite state machine $M$ which controls the activation of embedded components (see hierarchical model) 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 [16] developed at IMAG (Grenoble). It resembles the statecharts formalism by Harel [15], [21] 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 names of function objects. Events are sent to these functions when the transition is taken.

Fig. 6 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 $f_2$ receive an event.

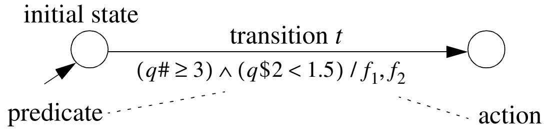



Fig. 6. Part of a simple automaton $M$ of some component.

## C. Operational Semantics of the Flat Model

Until now, we have not described how the state machine and the network interact. The basic *FunState* component is executed following the steps described below:

1. **Initialization**
The current state $s$ of the state machine is set to its initial state. All function objects are in state *idle* and have the remaining execution time $\tau = 0$. Global time $T$ is set to $T = 0$.
2. **Check for progress**
If from the current state there are no more enabled state machine transition conditions (the corresponding conditions on the states of the storage units have the value *false*) and there are no function objects which are in state *run*, the execution is stopped. If there are enabled state machine transition conditions originating in the current state, the execution continues at "*State machine reaction*" (step 4). If there are no enabled state machine transition conditions, but some function objects are still in state *run*, the execution continues at "*Function object termination*" (step 3).
3. **Function object termination**
Global time $T$ is progressed to the point in time $T'$ when the function objects with the least remaining execution time $\tau$ will finish their processing. The remaining execution times $\tau$ of all function objects in state *run* will be diminished by $T' - T$. All function objects in state *run* and with $\tau = 0$ finish processing at the new time instant $T'$. They write/add the tokens produced by the computational process of a function object to their output storage units and enter state *idle*. If two function objects add tokens to the same queue at the same instant, the resulting token order is non-deterministic, but tokens from one function do not interleave with those from other functions. If two function objects write to the same register at the same instant, a non-deterministic decision is made about which write action defines the final state. Again, the writing of one function is atomic. The execution continues at "*Check for progress*" (step 2).
4. **State machine reaction**
The state machine $M$ makes at most one transition. A transition can be taken if it originates in the current

state and the value of the corresponding condition evaluates to *true*. At the same time, the function objects whose names are in the action of the taken transition receive an event. If they are in state *idle*, they enter state *run*, consume tokens (read values) and initialize $\tau = lat$. Execution continues at "*Check for progress*" (step 2).

## IV. Model Extensions

### A. Extensions of the State Machine Description

Several extensions of the above-described simple state machine lead to a model similar to ARGOS [16]. In particular, we make use of the following concepts.

- **Hierarchy**
  States can be hierarchical, i.e., may contain other automata. In case of an XOR composition, the father state is interpreted as being in one of its child states (comparable to a conventional state machine). If the father is refined using AND composition, it is interpreted as being in all of its child states at the same time (concurrency). In figures, the child states are seperated by dashed lines in the following.
- **Events**
  Events can be part of predicates and action sets. An event has the value *true* if it is in the action set of a taken transition and *false* otherwise. Events are not visible outside a component.

Note that events are used for communication inside components only and reflect the control-oriented aspect of a system's behavior. They do not carry specific data but only do exist or not at a given point in time. Tokens, on the other hand, represent the data-oriented part, may have (almost arbitrary) values and lifetimes greater than zero, and are allowed to cross components' boundaries in either direction.

A state machine involving AND and XOR composition can easily be flattened, i.e., transformed into a simple state machine consisting of states and transitions labeled with conditions and actions.

These straightforward extensions are explained with the example of a simple exception handler in Fig. 7. State $X$ is refined using AND decomposition into three child states. If one of the predicates $p_1$ and $p_2$ in $X$ is *true*, then $X$ is left and one of the states $Y$ and $Z$ is entered to manage the exception. The right-hand child state resolves simultaneous exceptions via some priority rule. The communication uses the events $u$, $v$, $s$, and $t$. In particular, if only $p_1$ or both predicates are *true*, then state $Z$ is entered. If only $p_2$ is *true*, then a transition to state $Y$ occurs. All this happens within the same time instant.

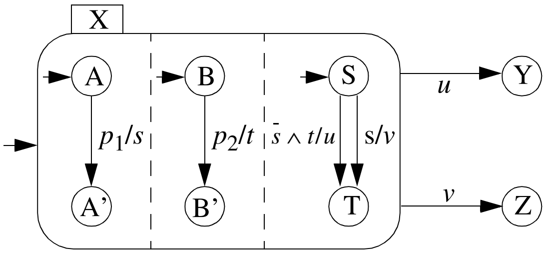



Fig. 7. Example of a state automaton with AND and XOR states.

### B. Hierarchical *FunState* Network

The basic element of the hierarchical network of the *FunState* formalism is the component. Each component contains a network $N$ and a state machine $M$ as defined for the basic model. In addition to this, $N$ can also contain embedded components and ports representing interfaces through which a component exchanges tokens with its father component. A hierarchical *FunState* model has exactly one top-level component. This top-level component has no interfaces.

*Definition IV.1:* The network $N = (\mathbf{F}, \mathbf{S}, \mathbf{C}, \mathbf{I}, \mathbf{O}, \mathbf{E})$ of a hierarchical *FunState* component contains a set of functions $\mathbf{F}$, a set of storage units $\mathbf{S}$, input ports $\mathbf{I}$, output ports $\mathbf{O}$, embedded components $\mathbf{C}$ with input and output ports $\mathbf{I_C}$ and $\mathbf{O_C}$, respectively, and directed edges $\mathbf{E} \subseteq ((\mathbf{F} \cup \mathbf{O_C}) \times (\mathbf{O} \cup \mathbf{S})) \cup ((\mathbf{I} \cup \mathbf{S}) \times (\mathbf{F} \cup \mathbf{I_C}))$. There is at most one edge entering an input port of an embedded component and at most one edge leaving an output port of an embedded component.

This basically means that a *FunState* component can be regarded as a refinement of a function. A simple example of a hierarchical component is shown in Fig. 8.

There are two further details concerning the hierarchical *FunState* model which must be explained.

- The state machine of a component may only access internal storage units or storage units which are directly connected to input ports—via the names of these input ports. For example, the state machine in Fig. 8 may contain a predicate of the form $i_1\# \geq 1$, i.e., the queue connected to input $i_1$ must contain at least one token.
- The state machine can send events to embedded components. If the action set of a taken transition contains the name of a component, then the state machine of this component can make a transition, i.e., it is activated. If it contains the name of an embedded component augmented with a event, this event is *true* within the embedded component. For example, an action set of the state machine in Fig. 8 may look like $C_1, C_2.b$. Then components $C_1$ and $C_2$ are activated, i.e., the state machines may perform a reaction. In addition, $b$ is *true* in $C_2$.

The detailed semantics of the hierarchical model can be described best by constructing an equivalent flat model. Before doing this, some characteristics of the hierarchical model will be summarized.

- The hierarchy can be nested arbitrarily deep.
- Each component contains a state machine as well as a network of functions, embedded components, and storage units.
- Communication between embedded components is performed via the explicit exchange of tokens, i.e., not synchronous.

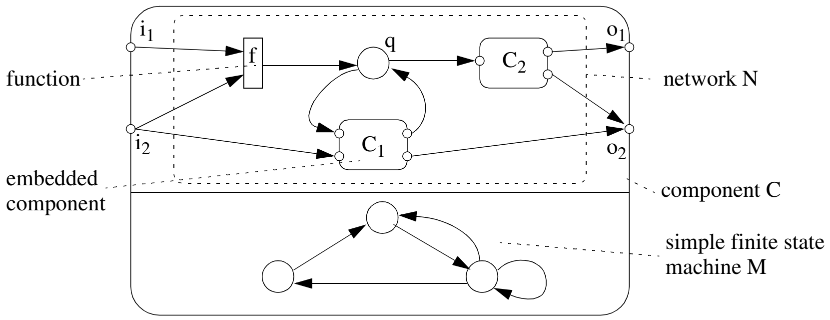



Fig. 8. Example of a hierarchical component.

- The father component explicitly activates its children in a synchronous way, i.e., enables them to make a transition.
- Flattening a hierarchical *FunState* model is quite simple. In particular, hierarchy does not extend the computational model of the basic *FunState* model.

### C. Removing Hierarchy

The operational semantics of *FunState* is defined in terms of the basic model. To this end, the flattening of a hierarchical *FunState* model will be explained. The flattening transformation involves several steps. Let us suppose that component $C$ contains the embedded component $C_1$ which will be flattened.

1. The function objects, embedded components, storage units and the edges of the network $N_1$ are moved from $C_1$ to $C$.
2. The sources of edges starting from an input port of $C_1$ are moved to the storage unit in $C$ having an edge pointing to that input port, and the input port is removed.
3. The targets of edges ending in an output port of $C_1$ are moved to the storage unit in $C$ having an edge coming from that output port, and the output port is removed.
4. Each occurrence of $C_1$ in the actions of any transition in the state machine of $C$ is replaced by a new event name. The condition of each transition in the state machine of $C_1$ is extended with this new event name using conjunction.
5. The state machine of $C_1$ is added to $C$ using AND-composition.
6. The remaining parts of the embedded component are removed.

In the above procedure, a suitable naming scheme must be used in order to avoid multiple occurrences and to uniquely identify elements.

In Fig. 9, part of a hierarchical *FunState* model and its equivalent flat model are shown. If predicate $b$ in Fig. 9 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 also *true*, resulting in a reaction of the state machine of $C_1$ in the same execution cycle if $p$ is satisfied, too.

### D. Timing Constraints

The purpose of timing constraints is to guide the specification or design process of a system by specifying certain deadlines. Checking a timing constraint shows whether all executions of a system satisfy the constraint.

Path constraints as defined by the following rules enable a constructive method to check whether a certain execution trace satisfies a given path constraint, as described in Section VII-D.

A path constraint is defined by a labeled path in the network of a component. A path is of the form $q_1 \xrightarrow{e_1} f_1 \xrightarrow{\hat{e}_1} q_2 ... \xrightarrow{e_{n-1}} f_{n-1} \xrightarrow{\hat{e}_{n-1}} q_n$ involving $n$ storage units, $n-1$ function objects and $2n-2$ edges. The first and last storage units may be input and output ports, respectively. Then they refer to the storage units which are directly connected to them outside the component. In order to specify deadlines properly, we are interested in two complementary properties, namely the first and the last reaction in storage unit $q_n$ as a result of a token arriving in unit $q_1$.

To this end, the path is labeled with two predicates $\lambda(\sigma)$ and $\mu(\sigma)$ which involve a free variable $\sigma$, for example $\lambda(\sigma) = (\sigma \geq 3) \wedge (\sigma \leq 7)$ and $\mu(\sigma) = (\sigma = 5)$. The free variable $\sigma$ denotes the time period which a token takes to travel along the specified path. In particular, the measurement for the time period starts when the token enters the first storage unit $q_1$ and ends when it enters the last one $q_n$. The predicates of a path constraint must be *true* for any token traveling along the specified path and for all possible executions of the system.

- The path constraint with respect to the first predicate $\lambda(\sigma)$ is said to be satisfied if the predicate $\lambda(\sigma)$ is *true* for the traveling time of the *first* token which arrives in $q_n$ and which has been initiated by a token arrival in $q_1$, for all possible execution sequences.
- The path constraint with respect to the second predicate $\mu(\sigma)$ is said to be satisfied if the predicate $\mu(\sigma)$ is *true* for the traveling time of the *last* token which arrives in $q_n$ and which has been initiated by a token arrival in $q_1$, for all possible execution sequences.

In order to clarify the meaning of traveling times, let us suppose that a given system has two different execution traces only, see Fig. 10. In the first one, two tokens named $v_1$ and $v_2$ enter the first queue $q_1$ of a path constraint, in the second one there is only one token $v_3$. The upward

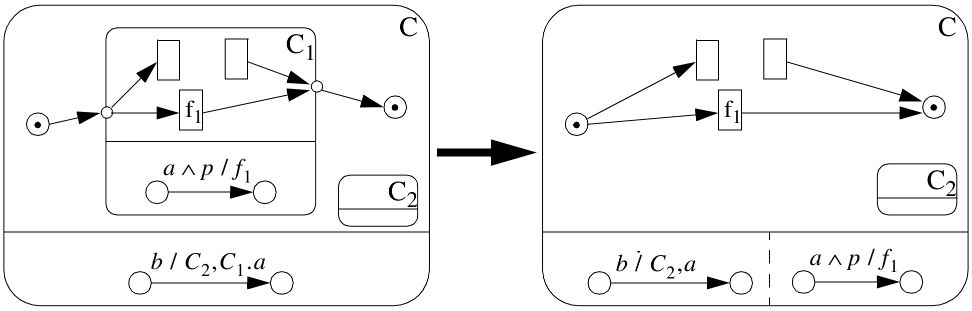



Fig. 9. Part of a hierarchical *FunState* model and its unfolding.

arrows in Fig. 10 represent the corresponding arrival times $t(v_1)$, $t(v_2)$, and $t(v_3)$. The downward arrows represent the causally dependent tokens that arrive in the last queue $q_n$ of the path. Obviously, the first token arrives in queue $q_n$ after time periods $L = \{2, 4\}$ and the last token arrives after time periods $M = \{4, 5, 6\}$. As mentioned above, a method for automatically determining these sets of traveling times $L$ and $M$ is presented in Section VII-D.

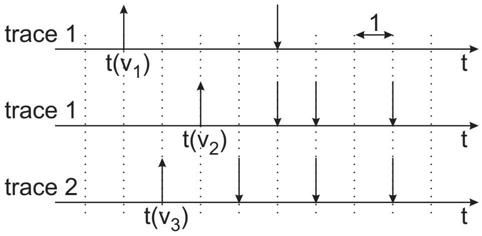



Fig. 10. Lower and upper path constraints.

## V. Regular State Machines

The purpose of this section is to introduce *FunState*'s underlying computational model called regular state machine (RSM) [5]. 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. It can easily be extended to more general settings. In particular, we start from the following class of *FunState* models:

- 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.
- We suppose that the hierarchy of components has been unfolded using the techniques described above. In addition, hierarchical state machines have to be flattened using conventional techniques for unfolding.
- The functions have constant consumption and production rates $c$ and $p$, respectively.
- Timing is neglected.

An example of the relationship between a *FunState* model and its computational model is given in Fig. 11. The numbers of tokens in queues $q_k$ correspond to the respective vector elements $i_k$ as introduced below.

*Definition V.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 with 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 $P : \mathbf{A} \times \mathbf{Z}^m \to \{\mathit{true}, \mathit{false}\}$, 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 V.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 $P(a, I_1) = \mathit{true}$.

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

In Fig. 11, the edges of the static state transition diagram are labeled in the form $p(a, I)/d(a)$. If the predicate is *true* for all index points $I \in \mathbf{Z}^m$ or if $d(a) = 0$, we simply

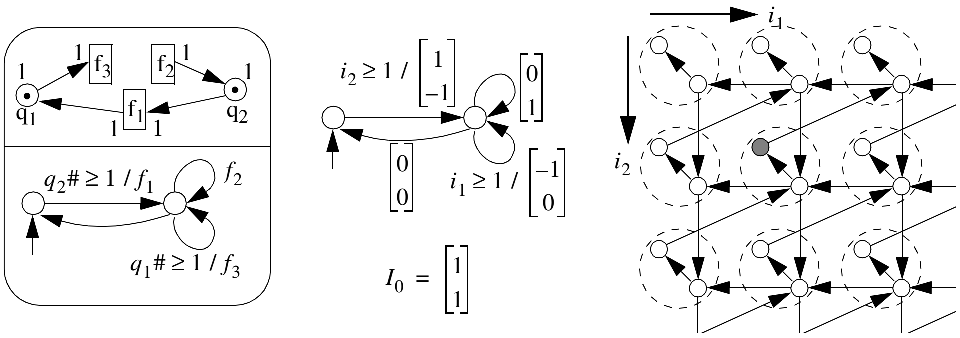


Fig. 11. A basic *FunState* model, its equivalent static state transition graph, and its dynamic state transition diagram.

write the label $d(a)$ or $p(a, I)$, respectively. The dynamic state transition diagram in Fig. 11 only shows a part of the index space. At each index point $I \in \mathbf{Z}^m$, there exist two states corresponding to the two states of the static state transition diagram. The initial state is shaded grey. The index point $I = (0,0)^T$ is shown in the upper left corner. Each transition within the dynamic state transition diagram corresponds to one of the static diagram's transitions. By means of the variables $i_1$ and $i_2$, the dynamic diagram also represents the queue contents of $q_1$ and $q_2$ in addition to the internal finite state of the state machine.

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., [22], [23], [24].

## VI. 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 events) 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.

As it will be seen, the combination of embedded components, refinement and abstraction mechanisms leads to a new approach to solving complex problems such as efficient analysis and scheduling.

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 internal model.

### A. Communicating Finite State Machines

Basic concepts of statechart-like [15] specifications and synchronous parallel state machines like ARGOS [16] are directly included as the *FunState* model supports AND and XOR substates. As a further example, the communication mechanisms of the POLIS [12] model for specification and design of embedded systems is described in some detail.

The POLIS model [12] 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., $q$. When 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. 12. It corresponds to the communication via a POLIS data signal. Other communication mechanisms such as general signals and control signals (involving also events) can be modeled easily as well.

### B. Marked Graphs and Synchronous Dataflow Graphs

Marked graphs [25] and synchronous dataflow (SDF) graphs [26], [1] are labeled directed graphs with nodes representing the actors of the system and edges denoting the communication and the corresponding FIFO queues between the actors. 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 easily. Fig. 2 shows a *FunState* model corresponding to the SDF graph shown in the left-hand part of Fig. 13. This model 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. 13. The terms "local" and "global" here refer to whether the "intelligence" of the scheduling strategy and its control are distributed over the entire system or not.

### C. Cyclo-Static Dataflow Graphs

In cyclo-static dataflow [27], [28], production and consumption rates of actors change periodically. Fig. 14 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

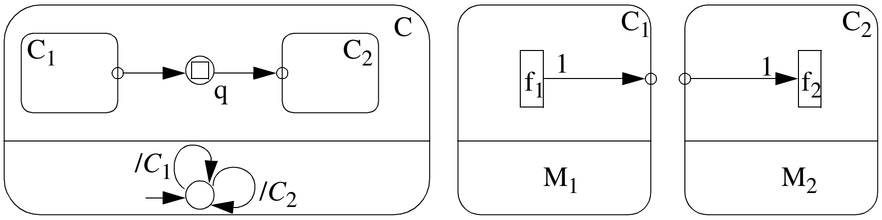



Fig. 12. Representation of the POLIS model.

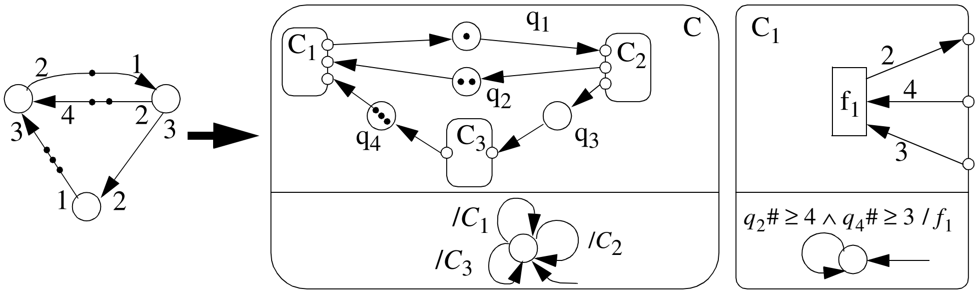



Fig. 13. Example of a synchronous dataflow graph and its representation as a *FunState* model with local control. Only embedded component $C_1$ is shown.

and production rates by cyclically activating the corresponding functions. The *FunState* components representing the actors are connected as in Fig. 13.

### *D. 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. 15. 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 $c\$1 \in \{\mathit{true}, \mathit{false}\}$ 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 $c\$1 = \mathit{true}$ or $c\$1 = \mathit{false}$, respectively.

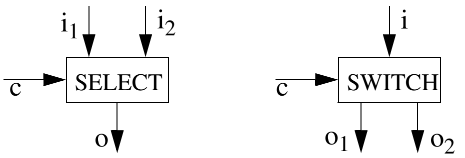



Fig. 15. SELECT and SWITCH nodes in Boolean dataflow graphs.

Fig. 16 shows the corresponding *FunState* models. The conditions are defined as

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

As an example of a node type defined in dynamic dataflow graphs, Fig. 17 shows a non-deterministic merge

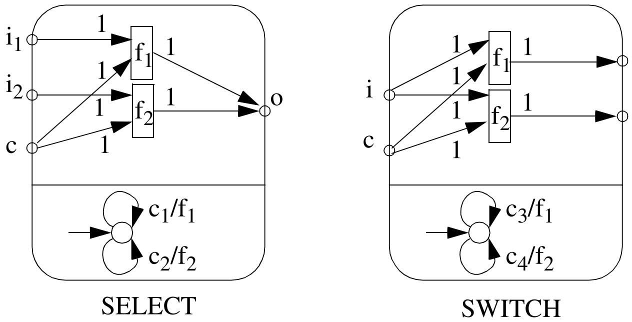



Fig. 16. SELECT and SWITCH nodes in the *FunState* model.

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.

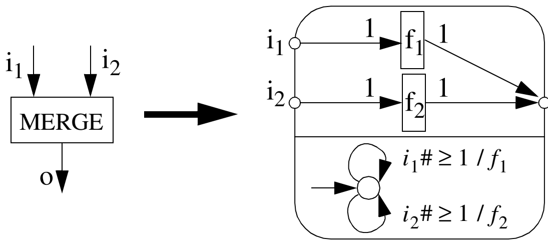



Fig. 17. MERGE node in the *FunState* model.

### *E. Petri Nets*

At a first glance, the *FunState* model seems to be almost equivalent to colored Petri nets (CPN) [29]. But there are several major differences which as well tune the Petri net model to the application domain of the *FunState* model and

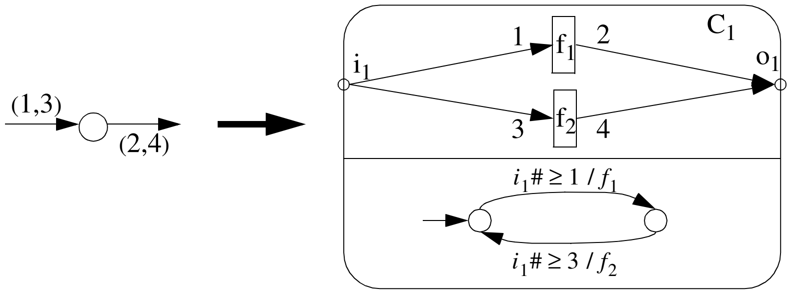



Fig. 14. A cyclo-static dataflow node.

at the same time generalize it. The following differences can be noted.

- 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 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, e.g., it is possible that a function is activated if there are at least 4 tokens in an input queue, but at the time of firing only 2 of them are removed.
- 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.

### *F. System Property Intervals*

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 [2], this state is not explicitly represented and thus not visible. Differences in the external behavior of a SPI process due to state-dependencies are modeled by uncertainty intervals. Even the refinement of process behavior using process modes [3] 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[1], 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 dataflow. In the following, it is shown how both models correspond, and translation rules are given and explained by means of simple examples. Timing is ignored in this context.

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 the top-level state machine in *FunState*, it is not generally possible to represent every *FunState* model with SPI[2]. 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. 13).

Straightforward 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 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. 18 where $M$ is the set of modes, $A$ is the activation function for process $P$, and $C_P$ is the *FunState* component representing process $P$.

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 component template as in Fig. 18 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 be-

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

[2]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 by this.

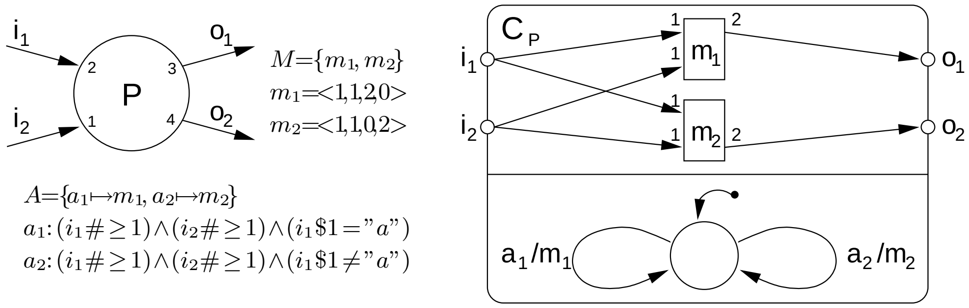



Fig. 18. Translation of a SPI process into a *FunState* component.

havior 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.

## VII. Applicable Methods

The purpose of this section is to show the versatility of the *FunState* model by application examples. 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.

### *A. 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 V. Of course, during the verification the state space is not enumerated explicitly. Instead, *symbolic model checking* [31] techniques are used for efficiency [5].

The verification goal is formulated by means of a *computation tree logic* (CTL) formula. Consider the example *FunState* model of Fig. 2. 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 $\mathsf{AG}\ \mathsf{EF}\ (q_1\# \geq 1)$ 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.

In summary, the above symbolic model checking strategy enables the efficient verification of certain temporal properties of state controlled process networks, where the explicit construction of an entire state transition graph is avoided by implicitly depicting it using symbolic representations. Thus, using *FunState* to internally model a mixed hardware/software system enables its formal verification comprising the whole well-known area of symbolic model checking concerning the detection of errors in specification and implementation.

Apart from this, formal verification may assist during the development of scheduling policies. The system model can be extended to describe a scheduling policy as well, of which the behavior then 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.

Many formal verification methods such as conventional symbolic model checking try to reduce the state explosion problem by implicit construction of the state space. The major limiting factor is the size of the symbolic representation mostly stored in huge *binary decision diagrams* (BDDs) [32]. The traditional BDD-based methods of automated verification suffer from the drawback that a binary representation of the system model and its state is required. As an alternative to BDDs, *interval diagram techniques* have shown to be convenient for formal verification of, e.g., process networks [33], Petri nets [34], and timed automata [35]. This new approach remedies some deficiencies of traditional approaches and often provides advantages with regard to computation time and memory resources [36]. These results can be extended directly from simple process networks towards the more complex

*FunState* model containing both finite-state control components and infinite-state dataflow queues. Even the timed *FunState* model can be verified by combining process networks and timed automata [37]. The verification procedure for *FunState* models has been implemented, and its efficiency in comparison to other state set representations has been shown [33], [34], [35], [37], [38].

### *B. Representing Schedules*

Besides formal verification, *FunState* has been designed to support diverse aspects of scheduling. First we describe the use of *FunState* as a representation model for several classes of scheduling policies. Afterwards, a methodology is sketched how to determine a partially static schedule of a *FunState* model.

In a hierarchical approach to solving complex scheduling problems it is necessary that the results of partially scheduling components can be represented in the same model. On the one hand, this enables the analysis of the entire scheduled model, such as formal verification. On the other hand, with this information further scheduling steps can be performed. This stepwise refinement corresponds to the stepwise reduction of the non-determinism in the model. This section contains some examples of different scheduling mechanisms. Further mechanisms which may be represented by *FunState* models are shown in [39].

#### B.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. 13 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. 2, only the state machine of the component $C$ must be changed in order to represent the schedule. Fig. 19 shows two different possibilities, both reflecting the periodic schedule described above. The second possibility takes into account that the sub-sequence $(f_2, f_3)$ occurs twice in the schedule and uses the AND composition facility of parallel state machines.

#### B.2 Scheduling with Static Priorities

In real-time systems, tasks can usually be suspended for the purpose of scheduling. An example is the theory of rate-monotonic scheduling [40] where mathematical conditions are provided for checking the schedulability of a set of periodic tasks. The rate-monotonic scheduling algorithm simply assigns priorities to the tasks in the order of their rates.

In the example shown in Fig. 20, there are three tasks $C_1, C_2$, and $C_3$. The request for executing a task is signaled by putting a token into the corresponding input queue of the component, i.e., into queues connected to $i_1$, $i_2$, or $i_3$, respectively. This could be done in an enclosing component by some sort of clock generator. A task puts a token into its output queue when it has finished computation. The two state machines are responsible for detecting the end of a task and for priority scheduling, respectively. The following conditions are used: $c_1 : i_1\# \geq 1$, $c_2 : i_2\# \geq 1$, $c_3 : i_3\# \geq 1$, $c_4 : r_1\# = 1$, $c_5 : r_2\# = 1$, and $c_6 : r_3\# = 1$.

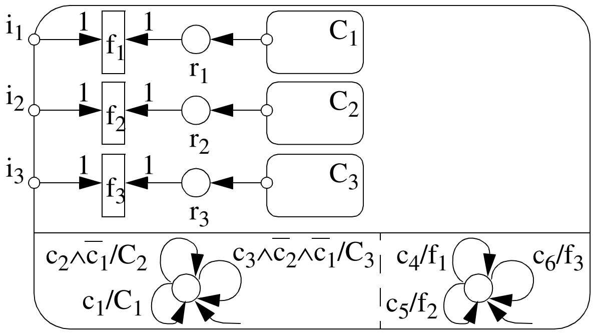



Fig. 20. Example for fixed priority scheduling with preemption.

### *C. Conflict-Dependent Scheduling*

To overcome drawbacks of either purely *static* or *dynamic* scheduling approaches and to combine their advantages, Lee proposed a technique called *quasi-static* scheduling [41]. 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 [42], [43], dynamic dataflow graphs [44], actors with data-dependent execution times [45], free-choice Petri nets [46], and *FunState* models [47]. In the following, the latter approach to *conflict-dependent scheduling* of *FunState* models is sketched.

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., [48].

Conflict-dependent scheduling [47] 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. Constraints imposed by other already implemented components are respected. The scheduling approach avoids the explicit enumeration of execution paths by using symbolic techniques. 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.

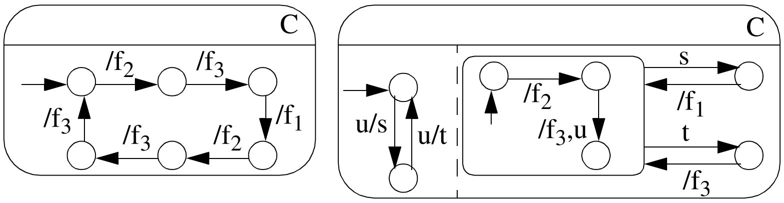



Fig. 19. Two possibilities for static periodic scheduling.

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. Thus, by finding all necessary cycles non-determinism is reduced as far as possible. Hence, design alternatives are removed by taking decisions.
3. The extracted schedule consisting of paths in the dynamic state transition diagram 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.

C.1 Conflicts and Alternatives

The scheduling methodology is introduced intuitively with the following example. Queue $q_1$ in Figure 21 is a multi-reader queue that may contain tokens which only one of the queue's readers $f_2$ and $f_3$ consumes (depending, e.g., on the token data) but the other one does not. The state machine describes a specification of possible schedules for this component (item 1 of the above methodology description).

If the predicates of all state machine transitions leaving a certain state are mutually disjoint, then the state is called *determinate*, otherwise *non-determinate*. We distinguish between two different kinds of non-determinism which leads to a classification of non-determinate states as follows.

- **Conflict**
  Non-determinate states are involved in a conflict concerning its outgoing transitions if the non-determinism 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 are made or when environmental circumstances have to be taken into account. The transitions involved in a conflict are called *conflicting*.
- **Alternative**
  If among several transitions leaving a state any transition can be chosen, this fact represents an alternative. Like this, for instance, different scheduling policies—i.e., different orderings of actor executions—can be modeled. Such decisions do not directly depend on the value of data but describe design alternatives which may even be fixed at compile time. Furthermore, different alternative algorithms can be modeled of which one or some can be selected during the design phase.

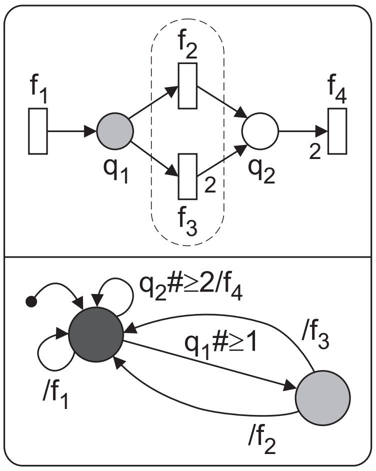



Fig. 21. Example *FunState* model with conflict and schedule specification.

This way, we can identify different sources of non-determinism and use this information for methods such as scheduling or formal verification.

In the following, conflicts are represented by light-shaded *conflict states*, while alternatives are depicted by dark-shaded *alternative states*. In Fig. 21, the data dependency regarding $q_1$ represents a conflict which is modeled using a conflict state. In contrast to this, all transitions starting in the alternative state represent design alternatives which may be chosen during schedule development. In the following, white states denote determinate states which either have only one outgoing transition or of which all transitions have disjoint predicates.

Intuitively, conflict-dependent scheduling as proposed in [47] and sketched in Section VII-C.2 replaces 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. 22 (item 3) which may replace the automaton in Fig. 21 for analysis or synthesis purposes (item 4). It consists of three static cycles and a conflict state switching between them. The predicate $p$ identifies the run-time decision associated with the conflict node.

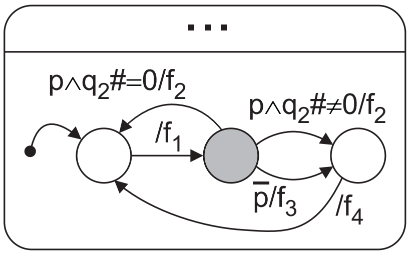



Fig. 22. Resulting controller automaton of example model.

The controller automaton can easily be transformed into program code as shown in Table I as pseudo code. The introduced approach has been applied to perform conflict-dependent scheduling for a molecular dynamics simulation system [47]. In Section VIII, conflict-dependent scheduling is applied to an ATM switch model.

TABLE I

Controller program code of example model.

```
a: f1;
   if p then
      f2;
      if q2# = 0 then goto a;
   else f3;
   f4;
   goto a;
```

C.2 Performing Conflict-Dependent Scheduling

The aim of the scheduling process described here is to sequentialize actor executions specified as concurrent while preserving all given conflict alternatives. The resulting schedule has to be deadlock-free and bounded.

In the following, the scheduling procedure is explained based on Fig. 23. First, the dynamic state transition graph of the corresponding regular state machine is searched for the shortest paths from the initial state to itself or any state already visited during the search. One of these (possibly multiple) shortest paths—representing or at least containing a cycle—is selected as the basis of the following scheduling procedure—$P_1$ in this example.

All conflict states of the selected path $P_1$ need further investigation as no conflict decision may be taken during schedule design. Hence, beginning with the successor states of the conflict state marked with $C_1$ again the dynamic state transition graph is searched until reaching any state visited already—resulting in the paths $P_2$ and $P_3$. Additional conflict states traversed during this search are also treated as described above. Thus, the conflict marked with $C_2$ causes the path $P_4$. Finally, $C_3$ causes $P_5$.

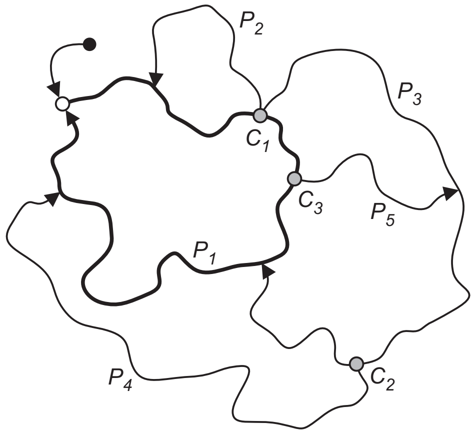



Fig. 23. Paths in dynamic state transition graph describing schedule.

The schedule is complete when each successor state of each visited conflict state has been considered. Thus, it is guaranteed that any conflict alternative during run time may be treated by providing a determinate schedule until the next conflict to be resolved.

If no schedule has been found while traversing one of the conflict paths, another shortest path is selected to repeat the scheduling procedure. If all shortest paths have been checked without finding a complete schedule, longer paths are selected. By introducing a bounding box on the state space, consisting of bounds on state variable values, the search space may be restricted if necessary. Thus, the termination of the algorithm is guaranteed. Furthermore, if a deadlock-free and bounded schedule exists, the above procedure will find it.

The length of the paths as the optimization objective is a heuristic criterion to minimize the number of actor executions and run-time decisions. This objective can be combined or replaced by criteria such as the least number of conflicts involved and their extent, or the shortest execution time along a path. In addition, the size of the bounding box on the state space is closely related to the amount of memory needed to store data of the scheduled *FunState* model. Therefore, restrictions on the run time as well as on the required memory can be included in the scheduling procedure.

Unfortunately, graph traversal tasks such as the mentioned search for paths in the dynamic state transition graph often suffer from the "state explosion" problem for real-world applications. This means that the possibly exponential blow-up of the number of states to be considered severely restricts the feasibility of such techniques. To avoid this, a symbolic approach to the scheduling problem [47], [49] has been introduced which uses symbolic model checking principles in order to avoid the explicit enumeration of execution paths.

In order to perform conflict-dependent scheduling, the dynamic state transition graph is traversed symbolically without constructing it explicitly. This way, shortest paths are determined by symbolic breadth-first searches. To achieve this, sets of states reachable from another set of states are considered—and computed in one single operation—instead of traversing the state transition graph path after path, state by state. For more details on conflict-dependent scheduling, the reader is referred to [47], [49], [37], [5].

Hence, in addition to formal verification as described above symbolic methods may be used not only for analyzing but even for developing scheduling policies for *FunState* models. Due to similar transition behaviors, the above advantages of symbolic approaches based on interval diagram techniques as an alternative to BDDs may be transferred to the area of symbolic scheduling. Symbolic scheduling methods turned out to often outperform both *integer linear programming* (ILP) and heuristic methods while yielding exact results. There exist some BDD-based symbolic approaches to control/data path scheduling in high-level synthesis. BDDs are used for describing scheduling constraints and solution sets either directly [50] or encapsulated in finite state machine descriptions [51], [52], [53]. Control/data path scheduling mostly is performed on the *register-transfer level* (RTL) which is located below the abstraction levels *FunState* is dedicated to. BDDs are well suited to represent RTL-based models.

### D. Performance Analysis

In Section IV-D, path constraints have been introduced as a means of specifying timing constraints on a model. In the following, a constructive, path constraint based method for checking the satisfaction of timing and performance properties is presented.

For a given path constraint, let us define sets $L$ (and $M$) which contain as elements all time periods of possible first (and last) causal dependencies, respectively. In other words, the predicates $\lambda(\sigma)$ and $\mu(\sigma)$ as defined in Section IV-D must be *true* for all time periods $\sigma \in L$ and $\sigma \in M$, i.e., for all tokens entering $q_1$ and for all possible execution traces of the system.

Let us now construct these sets $L$ and $M$ in principle. As shown in Section IV-D, we have to consider all possible execution traces of the system and for each trace all tokens which are written/added to $q_1$, the first storage unit in the path. For each of these execution-token pairs we do the following steps:

1. Let us suppose that a token $v$ is added/written to $q_1$ at some time $t(v)$. Then, the token is marked.
2. Let us suppose that a marked token is either read through an edge $e_i$ or removed via $e_i$ by function object $f_i$ for some $1 \le i \le n-1$. Then the tokens which are added/written to $q_{i+1}$ by that instance of function unit $f_i$ are also marked.
3. If a marked token enters storage unit $q_n$ (the last storage unit in the path) at some time $t$, then the mark is removed from the token. If the token was the first marked one which enters $q_n$, then add the time period $t - t(v)$ to $L$; if it was the last one, add the time period $t - t(v)$ to $M$. In the case that marked tokens are coming for an infinitely long time period, add $\infty$ to $M$.

All path constraints in the system must be checked independently.

The meaning of the path constraints should be obvious by the above construction. The second rule takes into account that we are interested in causality chains only, i.e., the value of the token in $q_n$ may depend on the value of the marked token in $q_1$. Only the dependency via the path is taken into account.

It is possible to extend the above definitions to the case where a path constraint involves an embedded component. Let us suppose that the path enters an embedded component $C_1$ at input port $i$ and leaves it at output port $o$. Corresponding to our hierarchical approach, the path need not be specified within $C_1$. Then, all possible paths in $C_1$ from $i$ to $o$ are considered. In particular, the second rule holds for all queues, functions, and edges in the embedded component $C_1$.

As an example, consider the component shown in Fig. 24. It models an iterative algorithm (function $f_2$) with latency $lat_2 = 1$. The algorithm needs four coefficients which can be put into the system via input port $i$. In the example, this happens in any execution trace at times 1, 2, 3, and 4. Function object $f_1$ with latency $lat_1 = 2$ transmits the coefficients to register $q_1$. The path constraint is drawn as a dotted arrow from the input port (queue connected to $i$) to the output port (queue connected to $o$).

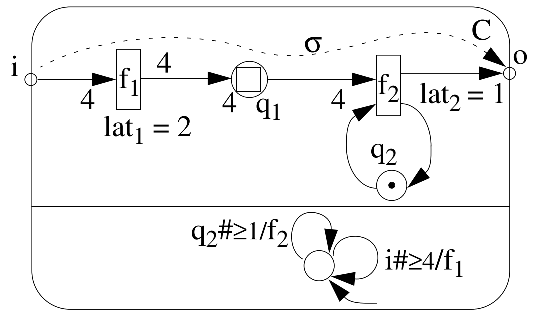



Fig. 24. Example of a path constraint.

Obviously, we have $L = \{3, 4, 5, 6\}$ and $M = \{\infty\}$. For example, the predicate $\lambda(\sigma) = (\sigma \le 6)$ is *true* as the first causally dependent token arrives in the queue connected to $o$ no later than 6 time units after a coefficient arrives in the queue connected to $i$.

## VIII. ATM Switch Example

This section treats a model of a shared memory ATM (*asynchronous transfer mode*) switch mapping ATM connections arriving at four input ports onto connections leaving on four output ports. Several *FunState* features and applications are explained with this model in the following.

### A. The ATM Switch Model

Fig. 25 shows the structure of the ATM switch model considered. The model is composed of several components working in parallel. The model imitates the tasks in the ATM user plane of the ATM layer [54] which are necessary for the switching of ATM cells. The interface to the physical layer is modeled according to the UTOPIA [55] specification using the octet-level handshake mode.

All input and output buffers are realized in a single memory block which is shared by all ports. This block is subdivided into segments which are large enough to store the information of an ATM cell. Usually, payload data and scheduling information for the output ports are stored separately. The 48 bytes data field can be stored at arbitrary addresses in the shared memory since the address of this field is stored in a FIFO organized queue of the corresponding connection together with scheduling information and the cell header. Another FIFO queue keeps track of addresses of free memory segments for storing payload fields. For each output port, there is a scheduler that must decide which connection will be allowed next to transfer a cell if a cell slot becomes available on the output port. Recently, variants of the *weighted fair queueing* (WFQ) [56] scheduling algorithm have been used for this task. A survey of scheduling algorithms including WFQ for packet and cell switching networks can be found in [57]. The basic idea of WFQ is to calculate a priority tag for each incoming ATM cell according to the reserved and currently used bandwidth of the corresponding connection. Then, the cell with the highest priority tag is chosen by the scheduler for transmission. In order to achieve a fair service of connections with a schedule close to the order in which the cells would have finished their transmission if the scheduler had served multiple connections simultaneously in proportion of their reserved rates, WFQ assumes an ideal fluid server for the calculation of the priority tags.

This functionality is modeled as follows. Incoming ATM cells are processed byte by byte by the UTOPIA interface. For each byte, the interface generates an event (*UtopInEv*) and produces a token carrying one byte of information (*UtopInBuf*). The first five bytes of the cell contain the cell header. When the beginning byte of a cell is signaled by the input processing component (*IP*), the switch starts several tasks. In order to find a free memory segment for the storage of the cell payload, the address queue *PayloadAddrQueue* is read out by the storage management component (*SM*). If no memory segment is available, the whole ATM cell and all associated temporary information will be dropped. Furthermore, a priority tag *Vi* must be calculated in the fair service calculation component (*FSC*) with the help of a virtual service measure *VtotAi* at the arrival time of the cell. The priority tag is used later on to schedule cells of different connections sharing the same output port. After the whole header has been transferred from the interface to component *IP* and stored at *HeaderStruct*, connection information can be extracted from the header by component *IP* and transferred via *VPIVCI* to the switching information component (*SI*) where it is used to map the cell onto an output port according to connection mapping data stored in an internal lookup table. The lookup additionally reveals connection context information which is needed by component *FSC* and passed through *Context*. The header can now be stored in one of the output port queues *HeaderQueues* of which each is sorted by rising priority. The sorting is performed in component *SM*. The payload is stored independently in the FIFO organized queue of the corresponding connection *PayloadQueues* at the address determined before (*PayloadAddr*). For this task, component *IP* passes the payload field via *Payload* to the storage management. However, if the lookup in component *SI* determines an impermissible cell, e.g., if the cell belongs to a non-existing connection, the cell and all its associated temporary information will be dropped.

The readout of cells via the output ports is triggered by the corresponding output UTOPIA interface, again byte by byte. It generates events (*UtopOutEv*) for each free byte in the output buffer of the interface. If sufficient free byte slots are available, the cell content is reconstructed in the output processing component (*OP*) by concatenating header and payload and then transferred to the corresponding output port buffer (*UtopOutBuf*). However, if there is no cell stored in the switch for transmission, the event tokens generated by *UtopOutEv* will be dropped. Finally, each cell transmission may trigger an update of the virtual service measure through *ConContext*. This measure is used within component *FSC* for priority tag calculations. Note that the priority tag calculation used in WFQ does not depend on this trigger since updates of the virtual service measure are initiated by an emulated ideal fluid server within component *FSC*. However, most approximations of WFQ use cell transmissions as a trigger for an update or a recalibration of internal variables.

The state machine of the FunState model in Fig. 25 schedules the components in a parallel manner. This is suited to a parallel implementation, e.g., by means of separate functional blocks in hardware. As an example, we pick one of the components to give a more detailed insight into the ATM switch. Component *FSC* in Fig. 26 performing fair service calculation shows an implementation which uses a WFQ approximation as cell scheduling algorithm. *Self-clocked fair queueing* (SCFQ) [58] simplifies the calculation of priority tags by estimating the virtual service measure by the priority tag of the cell currently in service. SCFQ's fair service calculations component thus only needs temporary registers for storing the current virtual time measure per output port *Vtot* and the priority tag of the preceding cell per connection *Vi1*. The functions within this component are scheduled sequentially by the state machine which reflects a software implementation of the component. As mentioned above, function *CalcVi* must not be executed on impermissible cells in queue *Context*. This is ensured by using the predicate *permissible* in the respective state machine transition.

### *B. Taking Advantage of FunState*

The above implementation of SCFQ requires only little computing resources. However, SCFQ is not able to guarantee as sharp delay bounds for cell transmissions as WFQ

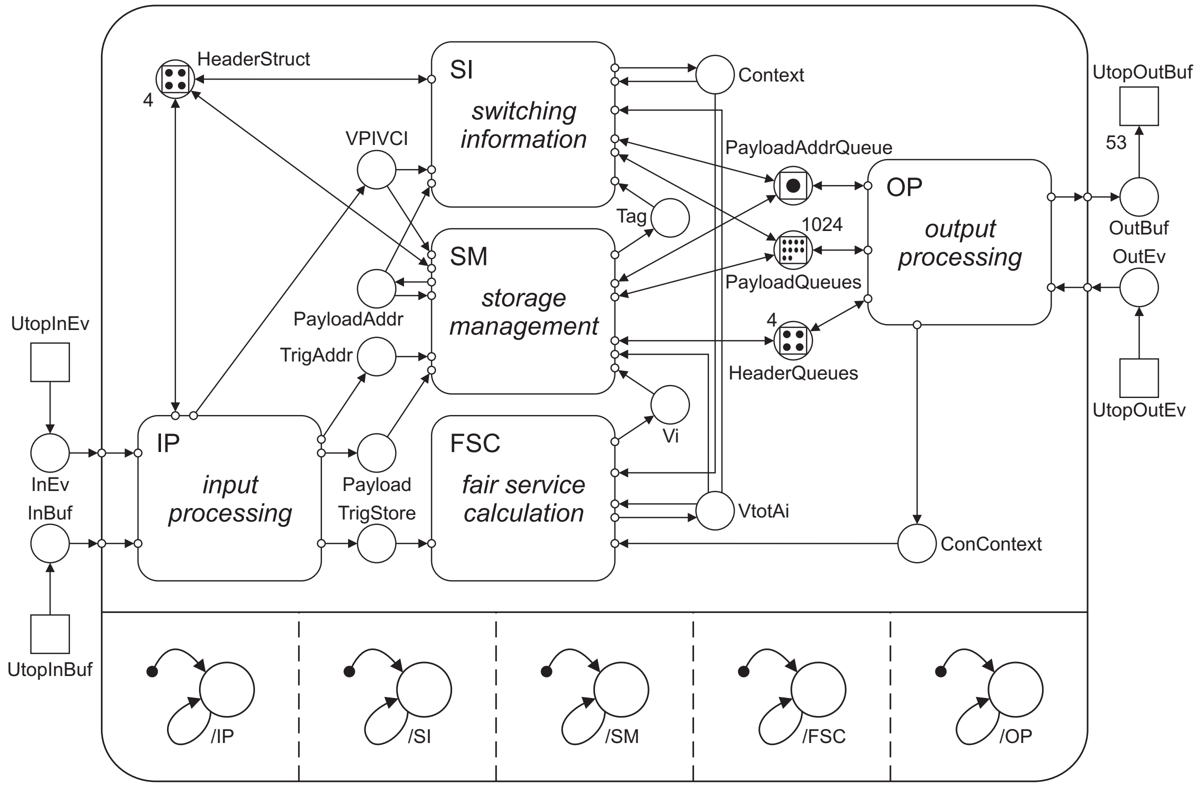



Fig. 25. Hierarchical ATM switch model.

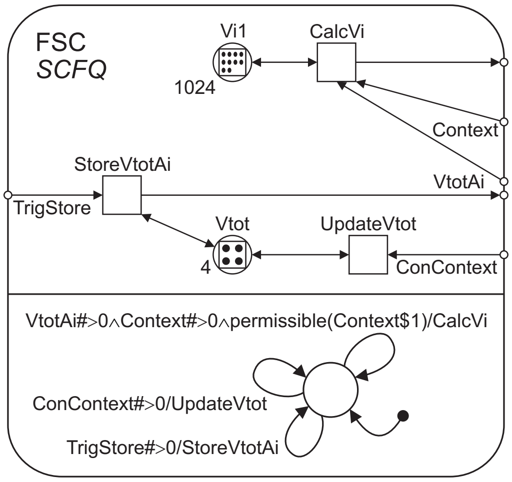



Fig. 26. Self-clocked fair queueing component performing fair service calculation.

does. Therefore, one may be interested in modeling a fair service calculations component for WFQ. Within the *FunState* model in Fig. 25, the SCFQ component in Fig. 26 can be replaced by the WFQ component shown in Fig. 27. This is achieved by simply plugging another component into the model in Fig. 25. Obviously, hierarchy and modularity are well supported by *FunState*.

The additional temporary registers are needed in order to keep track of the state of the emulated ideal fluid server such as the set of backlogged connections and the level of the emulated header queues. In addition to that, the WFQ component needs a timer component in order to model the appearance of cell transmissions in the fluid system which trigger updates of the virtual service measure. Moreover, the timer must be interruptable at the arrival of new cells since a cell arrival may initiate updates of all temporary registers in the WFQ component. Analogously to Fig. 26, the functions in Fig. 27 are scheduled sequentially. The transition labels indicated by "$\{l\}$" and "..." are omitted for clarity. In parallel to function execution, the timer is scheduled. This component *Timer* is shown in Fig. 28.

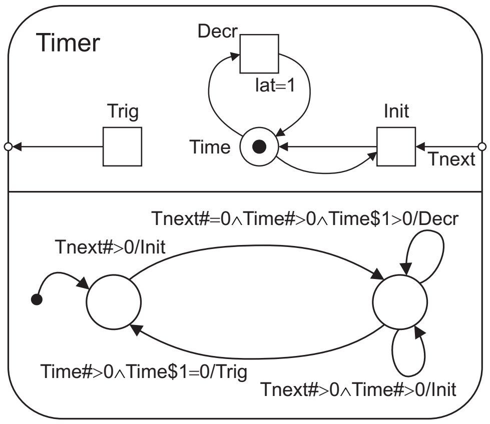



Fig. 28. Timer component within weighted fair queueing scheduling.

Function *Init* initializes the timer delay with the value of

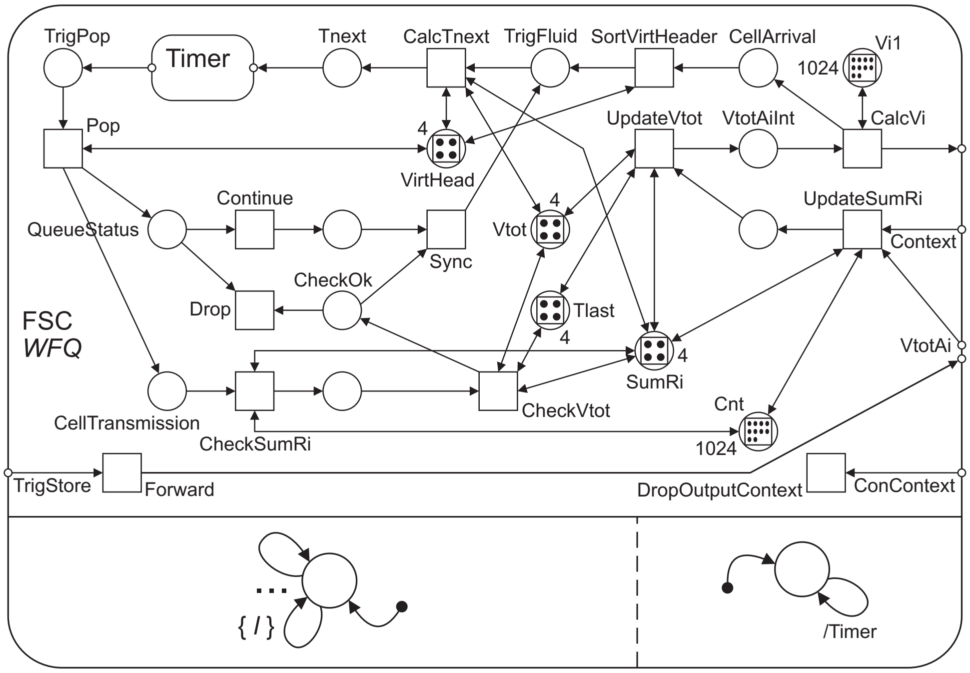



Fig. 27. Weighted fair queueing component emulating ideal fluid server within fair service calculation.

an incoming token. The current remaining delay is stored in queue *Time*. Function *Decr* with a latency of 1 time unit repeatedly decreases the value of the token in *Time* until it has reached 0. Then function *Trig* outputs a token triggering the superordinate component. The peculiarity of this timer is that if another token arrives at its input port during timer execution, the timer has to restart immediately without producing any output. This behavior can easily be modeled using *FunState* as shown in Fig. 28.

### C. Conflict-Dependent Scheduling

In this section, conflict-dependent scheduling is applied to the ATM switch model. Scheduling is performed after removing the hierarchy of the SCFQ model using the techniques described in Section IV. This results in the model shown in Fig. 29 which has been extended by a schedule specification. Obviously, the data dependency mentioned above regarding impermissible cells in queue *Context* has been abstracted by a conflict since it can only be resolved at run time. Software scheduling for a uni-processor is performed, hence the implementation is sequential in contrast to the parallel model used above. The partial state machines *InSequ* and *OutSequ* ensure that the correct order of incoming and outgoing ATM cell fields is guaranteed.

Table II shows the transition labels corresponding to the transitions abbreviated by "$\{l\}$" and "..." in Fig. 29, each starting and ending in the dark-shaded state.

In addition to the explicit event *Read*, function calls in the FSM part are also used as events for communication between concurrent state machines. For the sake of clearness, two state variables *ContextConfl* $\in$ $\{free, Imperm\}$ and *AddrConfl* $\in$ $\{free, DropIt\}$—both initialized with *free*—have been introduced which could be replaced by further concurrent state machines, each with two states. An in-state operator $M$ in $s$ has been introduced which is *true* iff state machine $M$ is in its state $s$.

The queues involved in conflicts are marked by shading in Fig. 29. All three conflicts have in common that usually one of their conflicting functions (*CalcVi*, *Store*, *OutHeader*) is executed and the other one (*Impermit*, *Drop*, *EmptyQueue*) only in case of an irregular operation which depends on the value of the token in the respective queue.

The *NewHeader* conflict differs from both the *Context* and the *Addr* conflict in that the transition predicates of both conflicting functions *OutHeader* and *EmptyQueue* are equal. Hence, the conflict model represents the fact that if one of the transitions is enabled, the other one is enabled as well. Then, based on the value of the token in *NewHeader* the conflict can be resolved directly by executing either *OutHeader* or *EmptyQueue*.

The latter is not the case for the *Context* conflict (and the *Addr* conflict as well). If *CalcVi* is enabled, then *Impermit* is also enabled and can be executed, but not vice versa. If the execution predicate of *CalcVi* is satisfied, the conflict is resolved based on the token in *Context*: Either *CalcVi* is executed directly or *Impermit* has to be executed—which possibly cannot be done due to its unsatisfied predicate. To avoid the necessity of repeatedly evaluating the conflict decision or of delaying the execution of *CalcVi*, the result of the conflict resolution is stored by changing the internal state of the conflict model and thus binding the conflict to *Impermit* by setting *ContextConfl* = *Imperm*. Later on, only the remaining execution predicates of *Impermit* have to be checked which results in less dynamic decision overhead than otherwise. The value *free* represents an unresolved conflict, while the

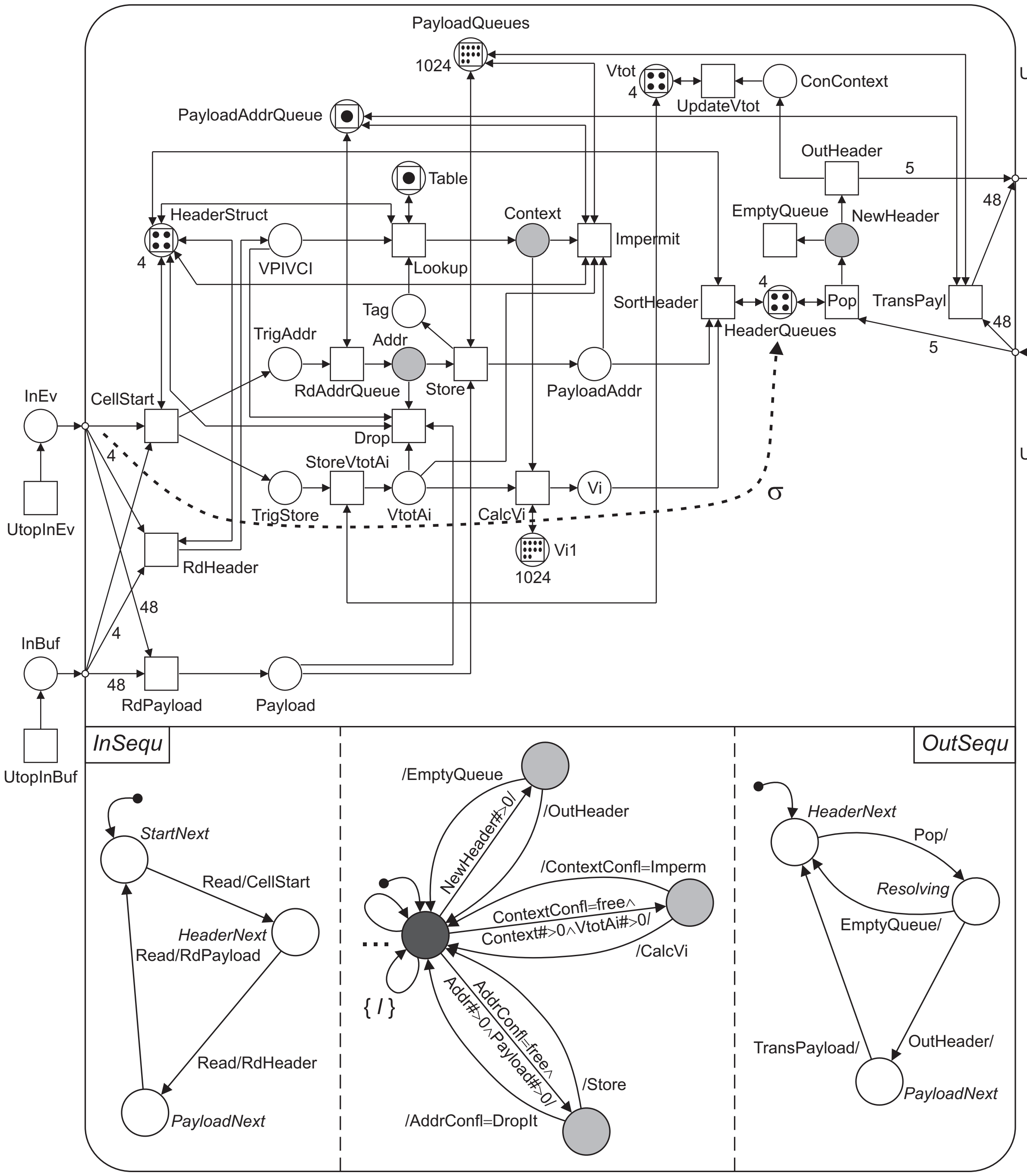



Fig. 29. ATM switch model with schedule specification.

values *Imperm* and *DropIt* denote that a conflict is bound to the respective transition.

For the sake of clearness, transitions resulting from the asynchronous behavior of the interfaces to the environment are disregarded here. Techniques for the automatic generation of a *FunState* model representing the conflict behavior from a given Petri net model as shown in Fig. 29 have been described in [49], [37].

Based on the *FunState* model in Fig. 29, the conflict-dependent scheduling procedure introduced above has been applied. The dynamic state transition graph of the schedule has been transformed into the controller automaton shown in Fig. 30. The result is a scheduling policy which may be implemented, e.g., as a software controller on a uni-processor.

Conflict decisions remaining in the resulting schedule again are represented by conflict states in Fig. 30. Besides conflict decisions—which cannot be resolved during compile time—, only three decisions had to be postponed until run time. Hence, the overhead by such dynamic decisions has been drastically reduced in comparison to the original *FunState* model. The scheduling process has been

TABLE II

Transition labels abbreviated by "$\{l\}$" in Fig. 29.

| | | |
|---|---|---|
| $\mathit{OutSequ}$ in $\mathit{HeaderNext} \wedge \mathit{OutEv}\# \geq 5$ | / | $\mathit{Pop}$ |
| $\mathit{OutSequ}$ in $\mathit{PayloadNext} \wedge \mathit{OutEv}\# \geq 48$ | / | $\mathit{TransPayl}$ |
| $\mathit{InSequ}$ in $\mathit{StartNext} \wedge \mathit{InEv}\# > 0 \wedge \mathit{InBuf}\# > 0$ | / | $\mathit{Read}$ |
| $\mathit{InSequ}$ in $\mathit{HeaderNext} \wedge \mathit{InEv}\# \geq 4 \wedge \mathit{InBuf}\# \geq 4$ | / | $\mathit{Read}$ |
| $\mathit{InSequ}$ in $\mathit{PayloadNext} \wedge \mathit{InEv}\# \geq 48 \wedge \mathit{InBuf}\# \geq 48$ | / | $\mathit{Read}$ |
| $\mathit{ContextConfl} = \mathit{Imperm} \wedge \mathit{PayloadAddr}\# > 0$ | / | $\mathit{Impermit}, \mathit{ContextConfl} = \mathit{free}$ |
| $\mathit{AddrConfl} = \mathit{DropIt} \wedge \mathit{VPIVCI}\# > 0 \wedge \mathit{VtotAi}\# > 0$ | / | $\mathit{Drop}, \mathit{AddrConfl} = \mathit{free}$ |
| $\mathit{TrigAddr}\# > 0$ | / | $\mathit{RdAddrQueue}$ |
| $\mathit{TrigStore}\# > 0$ | / | $\mathit{StoreVtotAi}$ |
| $\mathit{VPIVCI}\# > 0 \wedge \mathit{Tag}\# > 0$ | / | $\mathit{Lookup}$ |
| $\mathit{ConContext}\# > 0$ | / | $\mathit{UpdateVtot}$ |
| $\mathit{PayloadAddr}\# > 0 \wedge \mathit{Vi}\# > 0$ | / | $\mathit{SortHeader}$ |

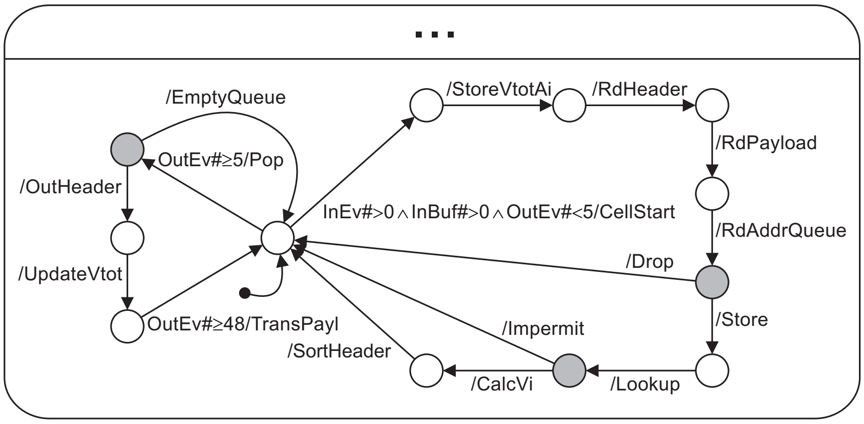



Fig. 30. Resulting controller automaton of ATM switch model.

performed mainly using the symbolic model checking tool SMV of Carnegie Mellon University. The computation took 11.1 s on a Sun Ultra 60 with 360 MHz.

For an implementation, the state machine representing the schedule may be transformed easily into program code as shown in Table III as pseudo code. The predicates $p_i$ identify run-time decisions associated with the respective conflicts and depending on token values. For instance, predicate $p_{Context}$ is equivalent to $\neg \mathit{permissible}(\mathit{Context}\$1)$.

In the preceding example, several ATM cells are allowed to be in a preprocessing state within the ATM switch, i.e., waiting for lookups or tag calculations. Thus, an outgoing link may be idle although there are cells in the system for this link. We can introduce a timing constraint so as to bound the preprocessing time, as sketched in Fig. 29. In the worst case, demanding that the switch should work at wire speed, we could constrain the preprocessing delay by $\lambda(\sigma) = \mu(\sigma) = (\sigma \leq \mathit{SizeOfATMCell}/\mathit{LinkBandwidth})$. In this way, only a single cell is allowed to be in a preprocessing state—virtually bounding the corresponding FIFO queues to the length one.

TABLE III

Program code representing schedule of ATM switch model.

```
a: if OutEv# ≥ 5 then
       if p_NewHeader then
           EmptyQueue; goto a;
       OutHeader; UpdateVtot;
       while OutEv# < 48 nop;
       TransPayl;
   else if InEv# > 0 ∧ InBuf# > 0 then
       CellStart; StoreVtotAi; RdHeader;
       RdPayload; RdAddrQueue;
       if p_Addr then
           Drop; goto a;
       Store; Lookup;
       if p_Context then
           Impermit; goto a;
       CalcVi; SortHeader;
   goto a;
```

## IX. Concluding Remarks

As it has been explained in this paper, the *FunState* model enables the internal representation of complex system behavior. In order to cope with the design complexity, the following hierarchical step-by-step approach is advertised and supported by the *FunState* model:

1. Restriction in some portions of the system, i.e., components, to well-known and simple models of computation. Within these subsystems, specialized and adapted methods can be applied.

2. Making use of the hierarchical composition to design hierarchical methods. To this end, it should be possible to restrict the scope of, e.g., scheduling, code generation, or verification to one component. Consequently, as well its environment as its embedded components must be simplified without sacrificing the quality or correctness of the overall result. This simplification should take into account the results of, e.g., previous scheduling or verification results for parts of the environment or embedded components.

The whole approach can be interpreted as a stepwise reduction of the non-determinism in a system specification. It has been shown that the *FunState* model supports the first item as it can represent different important elementary models of computation. The major property required for the second item is abstraction. It has been shown in the paper that the *FunState* model can represent the result of a (partial) schedule.

An approach to symbolic scheduling of mixed hardware/software systems has been presented. It is based on a *FunState* model of the system and the scheduling constraints. The result is a scheduling policy which may be implemented, e.g., as a software controller on a uni-processor.

## Acknowledgements

We wish to thank the anonymous reviewers for their constructive comments and helpful suggestions.

## References

[1] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[2] 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)*, Seattle, Washington, Mar. 1998, pp. 9–13.

[3] 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.

[4] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich, "*FunState*—an internal design representation for codesign," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-99)*, San Jose, California, November 7–11, 1999.

[5] L. Thiele, J. Teich, and K. Strehl, "Regular state machines," *Journal of Parallel Algorithms and Applications*, vol. 15, pp. 265–300, 2000, Special Issue on Advanced Regular Array Design.

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

[7] W.-T. Chang, A. Kalavade, and E. A. Lee, "Effective heterogeneous design and co-simulation," in *Proceedings of the NATO/ASI Workshop on Hardware/Software Co-design*, 1995, pp. 187–212.

[8] 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*, vol. 17, no. 12, pp. 1217–1229, 1998.

[9] W. Boßung, S. A. Huss, and S. Klaus, "High-level embedded system specifications based on process activation conditions," *Journal of VLSI Signal Processing*, vol. 21, no. 3, pp. 277–291, July 1999.

[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)*, 1998, pp. 132–138.

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

[12] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press, 1997.

[13] M. Pankert, O. Mauss, S. Ritz, and H. Meyr, "Dynamic data flow and control flow in high level DSP code synthesis," in *Proceedings of the 1994 IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 1994, vol. 2, pp. 449–452.

[14] F. Vahid, S. Narayan, and D. D. Gajski, "SpecCharts: a VHDL frontend for embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 6, pp. 694–706, 1995.

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

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

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

[18] A. Girault, B. Lee, and E. A. Lee, "A preliminary study of hierarchical finite state machines with multiple concurrency models," Tech. Rep. UCB/ERL M97/57, Electronics Research Laboratory, College of Engineering, University of California at Berkeley, 1997.

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

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

[21] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," *ACM Transactions on Software Engineerings Methods*, vol. 5, no. 4, Oct. 1996.

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

[23] 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*, 1988, pp. 398–406.

[24] J. Orlin, "Some problems in dynamic and periodic graphs," in *Progress in Combinatorial Optimization*, W. R. Pulleyblank, Ed. 1984, pp. 215–225, Academic Press, Orlando, Florida.

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

[26] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, 1987.

[27] G. Bilsen, P. Wauters, M. Engels, R. Lauwereins, and J. Peperstraete, "Development of a static load balancing tool," in *Proceedings of the 4th Workshop on Parallel and Distributed Processing*, Sofia, Bulgaria, 1993, pp. 179–194.

[28] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete, "Cyclo-Static Data Flow: Model and implementation," in *Proceedings of the 28th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, 1994, pp. 503–507.

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

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

[32] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[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)*, San Jose, California, November 8–12, 1998, pp. 686–692.

[34] K. Strehl and L. Thiele, "Interval diagram techniques for symbolic model checking of Petri nets," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE99)*, Munich, Germany, March 9–12, 1999, pp. 756–757.

[35] K. Strehl, "Interval diagrams: Increasing efficiency of symbolic real-time verification," in *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*, Hong Kong, December 13–15, 1999, pp. 488–491.

[36] K. Strehl and L. Thiele, "Interval diagram techniques and their applications," in *Proceedings of the 8th International Workshop on Post-Binary ULSI Systems*, Freiburg im Breisgau, Germany, May 19, 1999, pp. 23–24, invited paper.

[37] K. Strehl, *Symbolic Methods Applied to Formal Verification and Synthesis in Embedded Systems Design*, Ph.D. thesis, Swiss Federal Institute of Technology (ETH) Zurich, Shaker Verlag, Aachen, Germany, ISBN 3-8265-7193-2, March 2000.

[38] K. Strehl and L. Thiele, "Interval diagrams for efficient symbolic verification of process networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 8, pp. 939–956, Aug. 2000.

[39] L. Thiele, J. Teich, 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, Jan. 1998.

[40] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

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

[42] 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*, vol. 11, no. 6, pp. 696–718, June 1992.

[43] M. Cornero, F. Thoen, G. Goossens, and F. Curatelli, "Software synthesis for real-time information processing systems," in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds., pp. 260–279. Kluwer Academic Publishers, 1995.

[44] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Ph.D. thesis, Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A., 1993.

[45] S. Ha and E. A. Lee, "Compile-time scheduling of dynamic constructs in dataflow program graphs," *IEEE Transactions on Computers*, vol. 46, no. 7, pp. 768–778, July 1997.

[46] 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.

[47] 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 3–5, 1999, pp. 173–177.

[48] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–799, 1995.

[49] K. Strehl, L. Thiele, D. Ziegenbein, and R. Ernst, "Scheduling hardware/software systems using symbolic techniques," Technical Report TIK-67, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, Jan. 1999.

[50] I. Radivojević and F. Brewer, "Ensemble representation and techniques for exact control-dependent scheduling," in *Proceedings of the 7th International Symposium on High-Level Synthesis*, 1994, pp. 60–65.

[51] C. N. Coelho Jr. and G. De Micheli, "Dynamic scheduling and synchronization synthesis of concurrent digital systems under system-level constraints," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-94)*, 1994, pp. 175–181.

[52] S. Haynal and F. Brewer, "Efficient encoding for exact symbolic automata-based scheduling," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, 1998.

[53] S. Haynal and F. Brewer, "A model for scheduling protocol-constrained components and environments," in *Proceedings of the 36th Design Automation Conference (DAC '99)*, 1999.

[54] The ATM Forum Technical Committee, *ATM User-Network Interface Specification, Version 3.1*, ftp://ftp.atmforum.com/pub/approved-specs/af-uni-0010.002.pdf.tar.Z, Sept. 1994.

[55] The ATM Forum Technical Committee, *UTOPIA, An ATM-PHY Interface Specification, Level 2, Version 1.0*, ftp://ftp.atmforum.com/pub/approved-specs/af-phy-0039.000.pdf, June 1995.

[56] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 344–357, June 1993.

[57] H. Zhang, "Service disciplines for guaranteed performance service in packet-switching networks," *Proceedings of the IEEE*, vol. 83, no. 10, pp. 1374–1396, Oct. 1995.

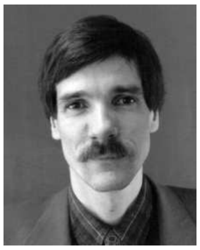
[58] S. J. Golestani, "A self-clocked fair queueing scheme for broadband applications," in *Proceedings of IEEE INFOCOM '94*, June 1994, vol. 2, pp. 636–646.

**Karsten Strehl** holds the Diploma degree (with distinction) in electrical engineering from the University of Karlsruhe, Germany, since 1997. He received the Doctor of Technical Sciences degree from the Swiss Federal Institute of Technology (ETH) Zurich in early 2000.

In 1997, Dr. Strehl joined Lothar Thiele's research group at the Computer Engineering and Networks Lab (TIK) at ETH Zurich. Since June 2000, he is with ETAS GmbH, Stuttgart, Germany, an embedded control engineering tool vendor in the automotive sector.

His research interests include design automation methods and tools for specification, analysis, and synthesis of embedded hardware/software systems, in particular high-level system design approaches, formal methods, and automated production code generation.

Dr. Strehl received the 1997 Award of the Faculty of Electrical Engineering at Karlsruhe University, the 1997 Siemens Information and Communication Award, and the ETH Medal for his Ph.D. thesis.



**Lothar Thiele** received the Dipl.-Ing. and Dr.-Ing. degrees in electrical engineering from the Technical University of Munich, Germany, in 1981 and 1985, respectively. After 1981, he has been a research associate with Rudolf Saal at the Institute of Network Theory and Circuit Design of the Technical University of Munich. After finishing his habilitation thesis, he joined the group of Thomas Kailath at the Information Systems Laboratory, Stanford University, in 1987.

In 1988, Dr. Thiele has taken up the Chair of Microelectronics at the Faculty of Engineering, Saarland University, Saarbrücken, Germany. He joined ETH Zurich, Switzerland, as a full professor in computer engineering at the end of 1994.

His research interests include models, methods, and software tools for the design of embedded systems.

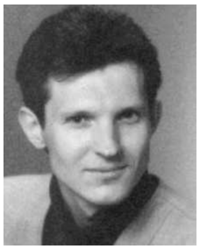
In 1986, Dr. Thiele received the Award of the Technical University of Munich for his Ph.D. thesis. He received the 1987 Outstanding Young Author Award of the IEEE Circuits and Systems Society. In 1988, he was the recipient of the 1988 Browder J. Thompson Memorial Prize of the IEEE.

**Matthias Gries** received the Dipl.-Ing. degree in electrical engineering from the Technical University of Hamburg-Harburg, Germany, in 1996.
Since 1997, he is with the Computer Engineering and Networks Lab (TIK) at the Swiss Federal Institute of Technology (ETH) Zurich, Switzerland, working towards the Ph.D. degree.
His research project deals with algorithm-architecture design trade-offs of network processors with the aim of preserving the Quality of Service (QoS) by feasible packet scheduling and policing. His interests also include queue management for access network devices as well as memory controller design.

**Dirk Ziegenbein** received the M.S. degree in electrical engineering from Virginia Tech in 1996.
Since 1997, he has been a member of Rolf Ernst's research group at the Institute of Computer and Communication Network Engineering (IDA) at Technical University of Braunschweig, Germany, where he is working on the development of the SPI Workbench, an approach to multi-language embedded system design.

His research interests include modeling, analysis, and optimization of complex embedded systems, in particular systems specified using several languages or models of computation.

**Rolf Ernst** got his diploma in Computer Science in 1981 and his Ph.D. in electrical engineering in 1988, both from the University of Erlangen-Nuremberg, Germany. From 1988 to 1989, he was a member of the technical staff at Bell Labs in Allentown, PA.
Since 1990, he is a full professor at the Technical University of Braunschweig, Germany, where he heads the Institute of Computer and Communication Network Engineering (IDA).
He was a main author of one of the first hardware/software co-synthesis systems, COSYMA. His main research interests are in embedded system design and embedded system design automation.



**Jürgen Teich** received his Dipl.-Ing. degree in 1989 from the University of Kaiserslautern, Germany (with honours). From 1989 to 1993, he was a Ph.D. student at Saarland University from where he received his Ph.D. degree (summa cum laude).
In 1994, Dr. Teich joined the DSP design group of Edward A. Lee and David G. Messerschmitt at the University of California at Berkeley, where he was working in the Ptolemy project. From 1995 to 1998, he held a position at TIK at ETH Zurich, finishing his habilitation thesis in 1996.

Since 1998, he is full professor in the Electrical Engineering and Information Technology Department of the University of Paderborn, Germany, holding a chair in computer engineering.

Dr. Teich is member of the IEEE and author of a textbook on hardware/software codesign edited by Springer in 1997.

His special interests are massive parallelism, embedded systems, hardware/software codesign, and computer architecture.