SCF - State Machine Controlled Flow Diagrams

TIK-Report Nr. 33
January 1998

Lothar Thiele, Jürgen Teich, Martin Naedele,
Karsten Strehl, Dirk Ziegenbein*

Abstract

In this paper, existing state machine and dataflow models of computation are revisited. A common representation called *SCF* is presented that enables the representation of several dataflow models using a mixture of functional programming and state machines. In particular, models like cyclostatic dataflow [3, 8], Synchronous Data Flow [14, 15], marked graphs [7] and communicating state machines [11, 18, 1] as well Petri nets turn out to be special subclasses of the *SCF* model. The model can be extended with information on the timing of computations and timing constraints. In addition, as abstraction and refinement are defined, the new model supports a hierarchical approach to problems like scheduling, verification and implementation.

*ETH Zurich, Computer Engineering and Networks Laboratory, CH-8092 Zurich, email: {thiele, teich, naedele, strehl}@tik.ee.ethz.ch and Technische Universitaet Braunschweig, Institut fuer Datenverarbeitungsanlagen, D-38106 Braunschweig, email: ziegenbein@ida.ing.tu-bs.de

Contents

1 Introduction

The purpose of the *SCF* model is to represent in an abstract way the interaction between different models of computation. The new unifying model is characterized by the following properties:

- A system consists of interacting components. Information and data between these components is passed via queues and registers.
- Components can contain other components leading to hierarchical composition.
- In each component the state based and computation based parts are separated. But in contrast to other approaches, the computations are represented as functions which are activated under the control of a state machine. They do not have their own dynamic behavior as in a dataflow model of computation.

As a result, the model supports hierarchy via the concept of components, it supports refinement and abstraction as a component can be replaced by an other which either refines it (for the purpose of specification) or makes an abstraction (for the purpose of efficient analysis and scheduling), it supports parallelism and non-determinism in different forms.

The report is concerned with an informal definition of the new model including its relation to other well known models. Examples are given which show the versatility of *SCF* in describing different aspects of a specification or implementation such as non-determinism and scheduling. Finally, some preliminary remarks on a useful abstraction mechanism are provided which may lead to efficient verification, analysis and scheduling methods.

1.1 Motivation

The heterogeneity of today's electronic systems, in particular in the area of embedded real-time systems, makes developers and engineers face new problems when it comes to the problems of how to specify, simulate, design and optimize such complex systems. Due to time and cost constraints, implementations typically comprise as well software (programmable) components, e.g., processors, and hardware (dedicated) components, e.g., ASICs. In order to arrive at optimal systems, different models of computation and different types of optimizations must be brought together.

Whereas some propagate a unified approach that seeks a consistent semantics for specification of the complete system, others favor the systematic combination of different semantics [13, 6].

In the first approach, a major problem consists in extracting properties from portions of a specification in order to be able to apply efficient analysis and synthesis techniques to these portions. A major problem in the second case lies in finding clean ways to mix diverse models of computations, at various levels of abstraction.

Concerning specification models, *abstract graphical models* have a major advantage over language based models such as C, VHDL, etc. because they are independent of a later refinement in hard- or software and because graphical formalisms are more intuitive to the designer as compared to textual models.

Chang et al. [6] classify different models of computation for describing the behaviour of embedded systems into those that have a notion of a global ordering of events (e.g., discrete-event models, cycle-based models, and synchronous language models like Esterel, Lustre, and Signal), and those imposing no or only partial ordering constraints (e.g., most implementations of dataflow models). For example, whereas a discrete-event model might be good at modelling communication networks or hardware circuits on the register-transfer level, specifying software running on a uni-processor would be an overkill due to slow execution.

Dataflow models may be specified as directed graphs of actors that represent computations and arcs connecting them representing streams of data. Design systems such as the Ptolemy system from UC Berkeley [4], COSSAP from Synopsys, or the DSP Station from Mentor Graphics all claim dataflow semantics in their models. They can be shown to be all derivates of *data flow process networks* [13]. In these networks, each process consists of repeated firings of a dataflow actor. A firing is a (often functional) quantum of computation. Processes communicate tokens via FIFO queues. Firing rules determine implicitly, when a computation is enabled to fire and how tokens are consumed and produced on the arcs. Different firing rule sets have lead to different well known dataflow models with different restricted modeling power such as marked graphs [7], synchronous dataflow [14, 15], and cyclostatic dataflow [3, 8]. These models have the major advantage of being implementation independent (no preference to hardware or software implementation is implied; partial order allows for concurrent as well as sequentialized (uni-processor) implementations [2]) and compile-time analyzability (liveness (deadlocks) and safety (boundedness)). However, due to the limited modeling capability, extensions have been proposed to include actors that allow the testing of inputs for data such as boolean controlled dataflow (BDF) [5] and dynamic dataflow models, introducing non-determinism on purpose, but at the price of giving up compile-time analyzability.

Whereas dataflow models are good at modeling concurrent behavior (*transformative systems*), there exist models dedicated to model *reactive control.* The requirement to model control flow separately has been recognized independently by many research groups, e.g., [6, 9]. Examples include mode and parameter control of data flow processing systems, system configuration and initialization, e.g. in packet based transmission systems [9], wireless modems [6], etc. Also in this area, graphical formalisms based on extensions of classical FSMs like hierarchical, concurrent FSMs as introduced by Harel [10] with many variants [17], [19] have been developed. They differ mainly in the semantics of communication between parallel states.

Deriving the expression partly from control theory, Chang et al. [6] call a system *digital hybrid system*, if a discrete-time system modelable using difference equations (hence a dataflow model) interacts with discrete-event controllers. They propose a hierarchical nesting of dataflow graphs with simple FSMs where the depth and the ordering of the nesting is arbitrary. The main idea is that a dataflow actor may have its functionality specified by a FSM. The FSM in turn can have actions associated with either states or transitions, and these actions may be specified by a dataflow graph. Either model can form the top level of a design. In order to avoid side effects of concurrent state machines communicating via hierarchies and using global signals, the requirement of only simple state machines allows for a modular semantics: The question in turn is to determine what a quantum of computation is in the semantic model used to manage the concurrency and communication. In [6], the model could be a dataflow model, a discrete-event model, or a synchronous model. The authors propose to choose a convenient model of computation at each level of hierarchy.

The major weaknesses of the above approach seem to be how to determine the behavior of the complete system without looking down to the lowest level in the hierarchy. When coming to an implementation, the latency of computations must be modelled. Considering a node firing being an atomic action seems to be unrealistic. In the *SCF* model, only the activation of computations that may take non-zero time is controlled. Other open questions that *SCF* is trying to address deal with questions of abstraction and analyzability. Allowing concurrency in the state machines seems to be a gain. The restriction, however, to local signal events with synchronous semantics inside the parallel state machines avoids side effects while maintaining modularity. By allowing state machines only to initiate computations, the computationally demanding part is separated from the control part. We will show that by requiring each elementary constituent of our model, a so-called *component*, to consist as well of a state machine and a (functional) network that existing dataflow models may be easily described, differing mainly in the state machine description that controls the activation (firing) of functional components, and hence describing the firing rules of the chosen model. Scheduling decisions, e.g., cyclo-static solutions, single-processor realizations, may be incorporated into the model by a simple transformation of the FSM of a component.

2 Basic untimed *SCF* model

Definition 2.1 *A basic untimed SCF model consists of a dataflow network N and a finite state machine M.*

Fig. 1 shows an example of a simple *SCF* model. In the basic model, actions are used in state machine transitions to activate functions in the data flow network. Functions symbolise operations on data.

2.1 Dataflow network

Definition 2.2 *The network $N = (F, S, E)$ of a basic SCF model contains a set of storage units S, a set of functions F, and a set of directed edges E. N is a bipartite graph. Data flow is represented by moving tokens $t \in \mathfrak{T}$. The marking m is the distribution of tokens to storage units. The marking m_0 is the initial marking.*

2.1.1 Storage units

Different storage unit classes belong to a common parent class “Storage unit” which defines their general embedding into the network. The actual access functionality and the available query functions on storage units can be defined for each subclass individually. Currently, the *SCF* model contains two storage unit classes: queues and registers. Extensions, for example for stacks, are easily possible.

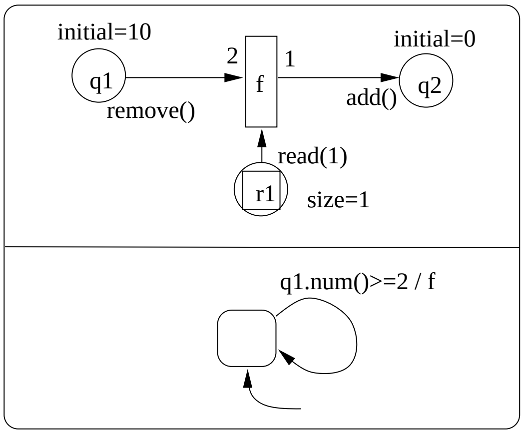

Figure 1: Example of a simple *SCF* model with one function f, two queues (q1, q2) and one register (r1) as storage units and a simple state machine.

Queues Queues have FIFO (first in first out) behavior and unbounded length. Queues have an input function add(arg[1]), which adds tokens to the end of the queue, an output function remove(arg), which removes the indicated number of tokens from the front of the queue, and a query function numberOfTokens() (abbreviated num() in figures and formulas), which returns the number of tokens in the queue. There must always be zero or more tokens in a queue.

Fig. 2 shows possible edge annotations for queues.

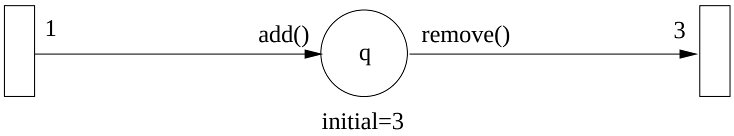

Figure 2: Example of a queue q with 3 initial non-distinguishable tokens. The read access will remove the first three tokens from the queue, if the queue contains more than 2 tokens at that time.

Registers Registers are linear arrays of limited length (size) of pairs (address, $\mathfrak{T} \cup$"$undefined$") of addresses and tokens. Registers have an input function write(arg), which replaces the tokens at the register address(es) determined by arg, an output function read(arg), which creates copies of the tokens at the specified register address(es).

[1] Here and in the following, arg is a possibly empty list of arguments whose interpretation depends on the specific function. These arguments are of type exp. In general, exp can be any expression involving functions and variables, and returning non-negative integer values. For the sake of analyzability, exp is here and in following sections limited to constants and, where this makes sense, intervals with constant limits. In certain circumstances, exp can also be a random process.

2.1.2 Functions

The function objects $f \in F$ of a *SCF* model are uniquely named and operate on tokens (Fig. 3). There exists some interpretation I that maps function objects from F to mathematical functions.

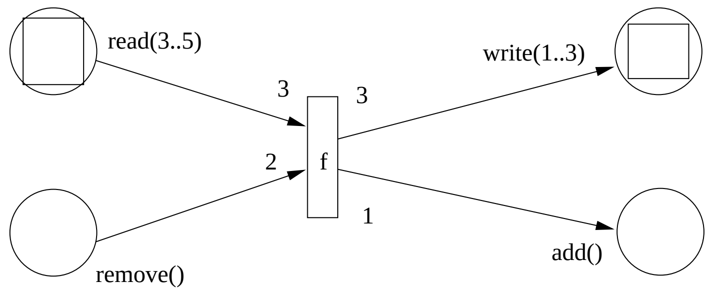

Figure 3: Example of a function with possible edges to storage units. The edges are labeled with weight expressions ($w_i \in W_{inp}$, $w_o \in W_{outp}$ - constants in the example) and storage unit specific access functions ($x \in X_{acc}$ - remove(), read(), add(), and write()) which describe the addressing of storage unit content.

2.1.3 Edges

Definition 2.3 *In the basic model there exist two kinds of edges: $E_{SF} \subseteq (S \times F \times X_{acc} \times W_{inp})$ connect storage units and function objects $E_{FS} \subseteq (F \times S \times X_{acc} \times W_{outp})$ connect function objects and storage units. The total set of edges is $E = E_{SF} \cup E_{FS}$. The set X_{acc} contains the storage unit specific access functions parametrised with expressions of type exp* [2]*. The main purpose of these access functions is to provide ways to address the storage unit content. The sets of function input and output weight expressions (W_{inp}, W_{outp}) of type exp denote the number of tokens consumed respectively emitted by a function* [3]*. The function $l_e : (E_{SF} \cup E_{FS}) \rightarrow L$ associates a label with an edge.*

It is possible to check statically whether access function and input/output weight expressions correspond to each other or not (e.g., a number n of tokens is required to be read, but the width m of the address range in the argument expression of the read function is not equal to n).

2.2 State machine

Definition 2.4 *The finite state machine $M = (\mathcal{S}, \mathcal{T}, s_0)$ (of a basic SCF model consists of a set of named states $\mathcal{S}$ and a set of transitions $\mathcal{T} = \mathcal{S} \times \mathcal{S} \times C \times 2^A$) and an unique initial state $s_0 \in \mathcal{S}$. C is set of possible state transition conditions. In the basic model, the conditions are logical propositions*[4] *over (in)eqalities of the form: $storage\_unit.query\_function$ $relational\_operator$ $constant$. A is set of possible actions. In the basic model, A contains the function names from F.*

[2] In the following text these expressions are omitted where their form is obvious.

[3] In the following text those expressions are assumed to be $const = 1$ unless stated otherwise

[4] In the figures of the paper, the following conventions will be used: & $\equiv$ AND, — $\equiv$ OR, and $\sim$ $\equiv$ NOT.

Fig. 4 shows the example of a simple automaton. The transition t is taken if the automaton is in the initial state, if there are at least two tokens in queue $q1$, and if there are less than three tokens in queue $q2$. At the same instant, actions f and s_1 are broadcast to the data flow elements of the model.

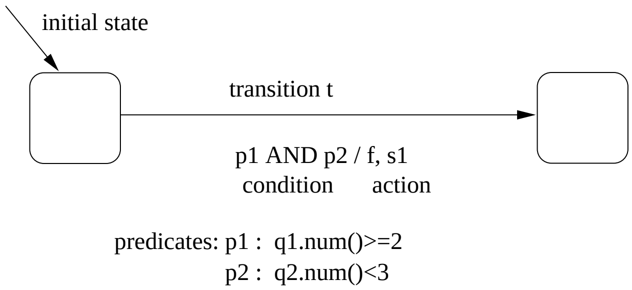

Figure 4: Example of a simple automaton M

2.3 Operational semantics of the non-hierarchical model

A *SCF* model is executed according to the steps described below:

Initialisation The current state s_c of the state machine is set to its initial state s_0. All queues and registers are filled with initial tokens according to the initial marking m_0 of the dataflow network.

Check for possible progress If there are no more enabled state machine transition conditions or active functions, the execution is stopped.

Function finalisation All active functions are finalised. Finalisation means to write/add the tokens produced by the computational process of a function to its output storage units. If two functions add tokens to the same queue at the same instant, the resulting token order is non-deterministic. If two functions write to the same register at the same instant, a non-deterministic decision is made about which write action defines the final state.

Predicate evaluation All predicates over storage units used in conditions of transitions originating from the currently active state of the state machine are evaluated.

State machine reaction The state machine reaction is executed. The automaton makes exactly one reaction for each model execution cycle, but only if at least one transition can be taken. A transition can be taken if it originates from the current state and its condition is *true*. After the reaction, actions from the action set of the taken transition are collected in the set of currently active actions A_c.

Function activation All functions that are named by actions $a \in A_c$ are activated in non-deterministic order. An activated function reads/removes tokens from its input storage units using the access specification on the corresponding arcs. If the input storage unit is a queue and in the same step a different function activation has already caused a token removal from this queue, then the distribution of the tokens from the queue to the removing functions is non-deterministic.

Trying to remove more tokens from a queue than it contains is illegal in the *SCF* model.

Clear action set The set of currently active actions, A_c, is set to $A_c = \emptyset$.

Loop Execution continues at "Check for possible progress".

3 Model extensions

3.1 Extensions of the state machine description

The extensions for the state machine component of a *SCF* model are similar to those of ARGOS [17] developed at IMAG (Grenoble) and the STATECHARTS formalisms by Harel [10].

3.1.1 Parallel states

As in STATECHARTS [10], the parallel composition of two automata is noted as drawing them into a common box, separated by a dashed line. The set of states is the Cartesian product of the sets of states of the composed automata. Its initial state is the pair of the initial states of the composed automata.

Parallel automata communicate using actions from the action set A, which has to be extended for this purpose with a finite vocabulary of signals A_{sig} for communication between automata, thus $A = F \cup A_{sig}$. Transition conditions of parallel state machines can thus contain logical propositions involving storage units and conjunctions as well as disjunctions of action expressions $a \in A$. Action expressions are of the form $a \in A_c$[5].

The operational semantics can be described by the following rules:

- Depending on the conditions at the transitions, either none, or one, or both composed automata participate in a reaction.
- If both react, the common action set is the union of the individual actions of the transitions.

For the operational semantics of parallel automata, the important restriction and difference to [10] exists that each parallel state machine may make at most one transition per execution cycle. It is therefore possible to calculate the finite number of possible transition combinations statically. One can then do all analysis on an automaton without parallel state machines whose states are the cartesian product of the parallel states and are connected by transitions that behave equivalent to the possible transition combinations of the original parallel configuration (Fig. 5(b)).

Fig. 5 shows the example of a simple automaton with two parallel state machines. Let the automaton be in its initial state (A,B). If the predicate p

[5]For simplicity, in the following text "a" in a condition will stand for " $a \in A_c$".

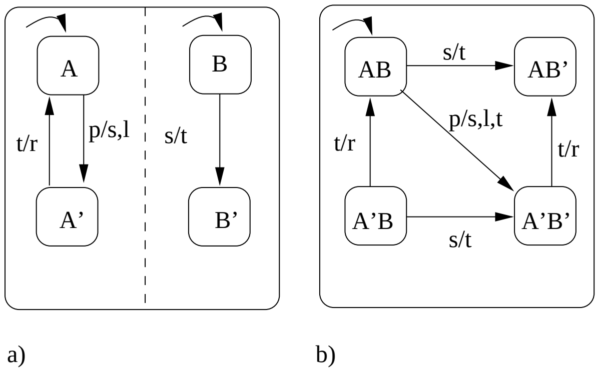

Figure 5: Example of a small automaton with parallel state machines (a) and its equivalent (b) with transitions obtained from all possible transition combinations of (a).

is *true*, the left state machine makes a transition and adds the signals s and l to the set of current actions, A_c. This now enables the transition of the right state machine, which is taken and adds the signal t to A_c. The final state of one execution cycle for this combined state machine is thus (A',B') and $A_c = \{s, l, t\}$ Both state transitions are considered to have happened simultaneously. The left state machine remains in state A' because during this execution cycle it may not make another transition even though there might be one with a *true* condition.

3.1.2 Hierarchical states

The hierarchical decomposition of an automaton consists in considering some of its states to be themselves automata. The different automata communicate via signals as in the case of parallel automata.

Similar to the ARGOS [17] model, the following rules describe the behavior of the hierarchical composition:

- When a hierarchical state is entered, it is in its initial state.
- When the hierarchical state is left, all information about its current state is lost.
- A hierarchical state does not participate in the transition that activates it, i.e., no internal transition may be taken in the same instant as the transition that enters the hierarchical state.
- A hierarchical state participates in the transition that kills it, i.e., the interruption takes place at the end of a possible internal transition, as can be seen from the example in Fig. 6(b)

The following Fig. 6 shows an example. The automaton behaves as follows:

- The initial state is B. Let us suppose that predicate p is *true* initially.
- When the automaton reacts, it enters the hierarchical state $A.X$.

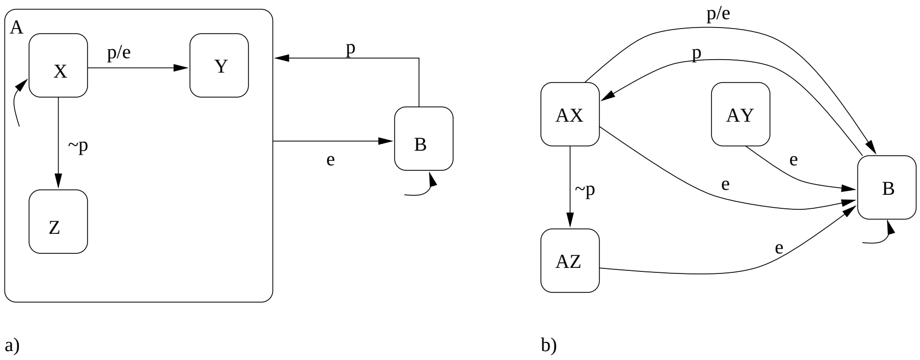

Figure 6: Example of a small automaton with a hierarchical state (a) and its flattened behavioral equivalent (b).

- In the next execution cycle, if p is *true* the automaton makes a transition to $A.Y$, adds action e to A_c and ends in state B. If p is *false*, the automaton makes a transition to $A.Z$.

The hierarchical composition can be used to facilitate the specification of more complex systems, for example process interrupts.

3.2 Hierarchy in the *SCF* data flow network

3.2.1 Components and interfaces

The basic element of the hierarchical data flow network of the *SCF* formalism is the component. Each component contains a data flow network N and a state machine M as defined for the basic model. In addition to this, N can also contain embedded components and interfaces through which the component exchanges data (tokens) with the component into which it is embedded. A hierarchical *SCF* model has exactly one top-level component. This top-level component has no interfaces.

The network $N = (F, S, C, I, O, E)$ of a hierarchical *SCF* component contains a set of storage units S, a set of functions F, a set of embedded components C, a set of input interfaces I, a set of output interfaces O, and a set of directed edges E. We call I_E and O_E the sets of input and output interfaces of the embedded components respectively.

To describe the possible interconnections, it is necessary to introduce several new edge types:

- $E_{IS} \subseteq (I \times S \times X_{acc})$, connects component input interfaces and storage units.
- $E_{II} \subseteq (I \times I_E)$, connects component input interfaces and input interfaces of embeddeded components.
- $E_{FO} \subseteq (F \times O \times W_{outp})$, connects functions and output interfaces.
- $E_{FI} \subseteq (F \times I_E \times W_{outp})$, connects functions and input interfaces of embedded components.

- $E_{OS} \subseteq (O_E \times S \times X_{acc})$, connects output interfaces of embedded components and storage units.
- $E_{OO} \subseteq (O_E \times O)$, connects output interfaces of embeddeded components and component output interfaces.
- $E_{OI} \subseteq (O_E \times I_E)$, connects output interfaces of embeddeded components and input interfaces of embedded components.

This basically means that a *SCF* component can be regarded as a refinement of a storage unit/function pair. This component model was chosen because it allows simple local treatment of predicate evaluations independent of component execution order. However, components that can be used as storage unit or function refinements might turn out to be useful for the modeling.

It is possible for the state machine of a component to communicate with the state machine of an embedded component[6]. This is done by initialising the set of current actions A_c of the state machine of the embedded component with a certain set of signals. Except for the fact that it is a one-way communication only, it works very similar to the communication between parallel state machines.

In Fig. 7, if the predicate p is *true*, the state machine of C makes a transition which results in A_C={t} for this component. It also initialises the set of current actions of $C1$ to A_C={m,n}, which has as consequence that the condition for the transition of the state machine of $C1$ is *true*, resulting in a reaction of the state machine of $C1$ in the same execution cycle.

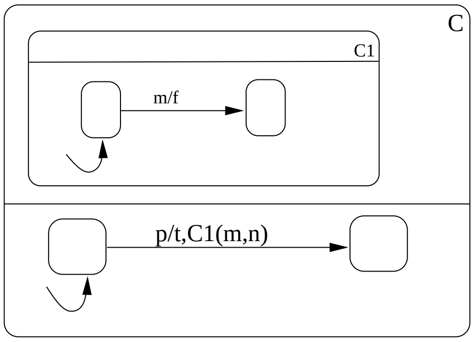

Figure 7: Initialising the set of current actions of a state machine of an embedded component

3.2.2 Operational semantics of the hierarchical model

The operational semantics of *SCF* is defined in terms of the basic model. The hierarchical model needs to be flattened to be able to apply this operational

[6] To provide this means of communication does not really extend the power of the formalism, as the same functionality could be realised by adding a special interface and storage unit to the embedded component, into which the outer component would write its "command" with the help of suitable functions and for which the corresponding predicates would be defined for the transitions of the embedded component. The main difference is that then a signal sent to the embedded component would not be valid before the next execution cycle.

semantics.

In this section first the flattening of a hierarchical *SCF* model will be explained and then a hierarchical interpretation of the operational semantics will be given.

As shown in the following figures, the flattening transformation involves several steps. Let us suppose that component C contains the embedded component C_1 which will be flattened into C.

1. The functions, embedded components, storage units, interfaces, and the edges are moved from C_1 to C. If necessary, elements have to be renamed to avoid multiple occurrences of a name.

2. The sources of edges starting from an input port of C_1 are moved to that/these function(s) and/or component(s) in C having an edge pointing to that input port, and the input port is removed, see Fig. 8.

3. The targets of edges ending in an output port of C_1 are moved to that/these storage unit(s) in C having an edge coming from that output port, and the output port is removed, see Fig. 8.

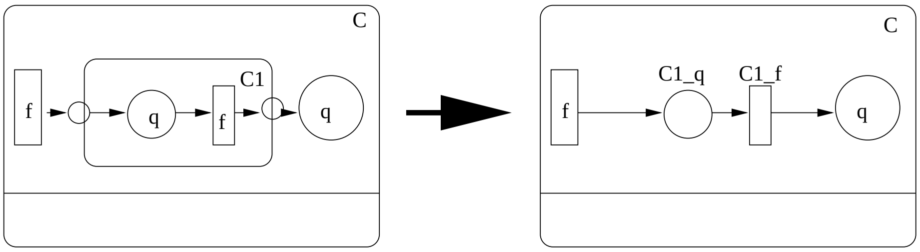

Figure 8: Flattening the network part of an embedded component.

4. The state machine of C_1 is added as parallel state machine to C.

5. The names of actions have to be changed in a suitable way to correspond to the name changes made in step 1 and to emulate the fact that previously the effect of the current action set of C_1 used to be limited to C_1 (Fig. 9).

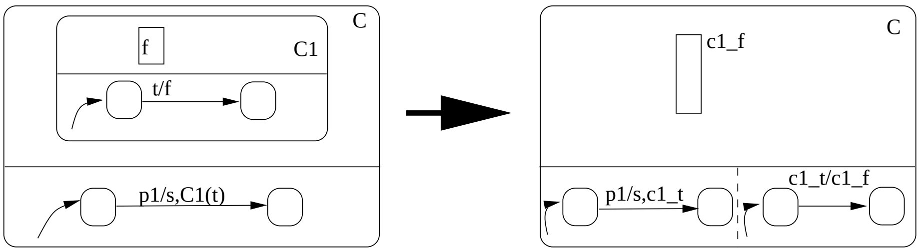

Figure 9: Flattening the state machine part of an embedded component.

6. The remaining parts of the embedded component are removed.

The following paragraphs describe the hierarchical interpretation of the *SCF* operational semantics. It is equivalent, meaning being able to produce the same state changes as result of one execution cycle, to the operational semantics for the basic model, because no additional conflicts are introduced by the concept of hierarchical components and no special functionality is bound to the components as such.

Initialisation The current state s_c of the state machine is set to s_0 for all components respectively. All queues and registers of each component are filled with initial tokens according to the initial markings m_0 of their dataflow networks.

Check for possible progress If there are no more enabled state machine transition conditions or active functions anywhere in the hierarchy, the execution is stopped.

Function finalisation All active functions throughout the hierarchy are finalised. Finalisation means to write/add the tokens produced by the computational process of a function to its output storage units. If two functions add tokens to the same queue at the same instant, the resulting token order is non-deterministic. If two functions write to the same register at the same instant, a non-deterministic decision is made about which write action defines the final state.

Execute top level component The top-level component is executed as described below.

Loop Execution continues at "Check for possible progress".

The execution of a component happens as follows:

Predicate evaluation All predicates over storage units used in conditions of transitions originating from the current state of the state machine of the component are evaluated. The state of the component internal storage units is independent of the order of the executed components, because all write/add operations have already been done before the first component is executed, and because remove operations can only be initiated by functions within the component being executed after its predicates have been evaluated.

State machine reaction The state machine reacts on predicates that are *true* and actions in A_C of this component. Actions from the action set of the taken transition are added to the set of currently active actions A_C for this compoent.

Function activation All functions within this component that are named by actions $a \in A_c$ are activated in non-deterministic order. An activated function reads/removes tokens from its input storage units using the access specification on the corresponding arcs. If the input storage unit is a queue and in the same step a different function activation has already caused a token removal from this queue, then the distribution of the tokens from the queue to the removing functions is non-deterministic.

Trying to remove more tokens from a queue than it contains is illegal in the *SCF* model.

Embedded component execution All embedded components of this component are executed in non-deterministic order, after first having initialised their respective A_C, if necessary.

Clear action set The set of currently active actions for this component, A_C, is set to $A_C = \oslash$.

3.3 Timed *SCF* model

In this section, the *SCF* model will be extended with informations concerning the timing of its execution. Two classes of timing information can be distinguished: Timing properties and timing constraints.

- Timing properties specify certain execution durations. A typical example is the latency of a function.
- The purpose of timing constraints is to check whether any execution of a system satisfies them. They do not restrict the behavior of the system. Timing constraints can be used to guide the specification or design process of a system by specifying certain deadlines.

It would be possible to define timing constraints and timing properties in an equivalent way. Attached to some timing annotation would be a flag which distinguishes between properties and constraints. But it turns out that this generality comes with a high notational overhead. Therefore, we restrict the class of possible timing annotations to a useful subset, namely *latency* in the case of properties and *paths* in the case of constraints. Depending on the particular application domain it may be necessary to include more general annotations.

3.3.1 Latency

The definition of the latency of a function enables a timed execution of a model, i.e. with each state change can be associated a physical time.

Definition 3.1 *The function $\tau : F \rightarrow \mathbf{R}_0^+$ associates a latency with each function of the dataflow network N of a SCF component. The function $T : F \rightarrow \mathbf{R}_0^+$ associates a remaining processing duration with each active function.*

The operational semantics of the basic *SCF* model has to be extended to incorporate function latency. The respective steps are changed as follows:

Initialisation The current state s_c of the state machine is set to its initial state s_0. All queues and registers are filled with initial tokens according to the initial marking m_0 of the dataflow network. Global time t is set to $t = 0$.

Check for possible progress If there are no more enabled state machine transition conditions or active functions, the execution is stopped. If there are no more enabled state machine transition conditions, but some functions are still active, global time t is progressed to the point in time t' when the function(s) with the least remaining processing duration T_i will finish their processing. The remaining processing duration T_j of all other functions will be diminished accordingly. If there are still enabled state machine transition conditions at the current time instant t, the global time will not be advanced.

Function finalisation All functions f_i that finish processing at the new time instant t' are finalised. Finalisation means to write/add the tokens produced by the computational process of a function to its output storage units. If two functions add tokens to the same queue at the same instant, the resulting token order is non-deterministic. If two functions write to the same register at the same instant, a non-deterministic decision is made about which write action defines the final state.

Predicate evaluation As before.

State machine reaction As before.

Function activation All functions that are named by actions $a \in A_c$ are activated in non-deterministic order. It is illegal to reactivate a function that is already active at that instant. The remaining processing duration timers of the newly activated functions are set to $T_i = \tau_i$. An activated function reads/removes tokens from its input storage units using the access specification on the corresponding arcs. If the input storage unit is a queue and in the same step a different function activation has already caused a token removal from this queue, then the distribution of the tokens from the queue to the removing functions is non-deterministic.

Trying to remove more tokens from a queue than it contains is illegal in the *SCF* model.

Clear action set As before.

Loop As before.

The hierarchical interpretation of the operational semantics has to be extended accordingly.

In the case of $\tau_i = 0 \; \forall i | f_i \in F$, the operational semantics for the *SCF* model with function latencies becomes equal to the operational semantics of the basic untimed model.

3.3.2 Path constraints

As an example of timing constraints, path constraints will be considered. They can be used to specify certain deadlines[7].

The following rules define path constraints and give a constructive method to check whether a certain execution trace satisfies a given path constraint.

- A path constraint is a labeled path in the network N of a *SCF* model. The path is of the form $(s_1 \xrightarrow{e_1} f_1 \xrightarrow{\hat{e}_1} s_2 ... \xrightarrow{e_{n-1}} f_{n-1} \xrightarrow{\hat{e}_{n-1}} s_n)$ involving n storage units, $n-1$ functions and $2n-2$ edges. The path is labeled with a predicate, i.e. a conjunction of inequalities which involve a free variable σ, for example $(\sigma \geq 3) \wedge (\sigma \leq 7)$. The free variable σ denotes time intervals which tokens need to travel along the specified path. The predicate of a path constraint must be *true* for any token during any possible execution of the system.

[7] Note again, that it may also be possible to define path properties in a similar way. Then, the system is build in a way that only executions satisfying these path properties are possible

- To be more specific, let us define sets Σ_v of time values for each token v in s_1 during the execution of the system. The predicate of the path constraint must be *true* for all time values $\sigma \in \Sigma_v$ in all sets Σ_v. The set Σ_v for a specific token v in s_1 can be computed as follows:

  1. Let us suppose that the token v is added/written to s_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 an instance of function f_i for some $1 \leq i \leq n-1$. Then the tokens which are added/written to s_{i+1} by that instance are marked too.
  3. If a marked token enters storage unit s_n at some time t, then the time difference $t - t(v)$ is added to Σ_v and the mark is removed from the token.
  4. The sets Σ_v must be determined independently for all tokens in s_1.

All path constraints in the system must be checked independently. It is illegal to violate a path contraint during model execution. The meaning of a path constraint should be obvious by the above construction. The second rule takes into account that we are interested only in causality chains, i.e. the value of the token in q_n may depend on the value of the marked token in s_1. Only the dependency via the path is taken into account. The last rule avoids some notational difficulties as otherwise, dependency paths of several tokens in s_1 may interfere.

It would be 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 is executed for all queues, functions and edges in the embedded component C_1.

As an example, consider the component shown in Fig. 10. It models an iterative algorithm (function f_2) with latency $\tau_2 = 1$. The algorithm needs 4 coefficients which can be put into the system via queue $q1$. In the example, this happens at times $1, 2, 3$ and 4. Function f_1 transmits the coefficients to a register, i.e. r_1, with a latency of $\tau_1 = 2$. The path constraint is drawn as a dotted arrow from the queue connected to input port i to the output port o.

Two possible execution traces are shown in Fig. 11. At row i, the time instances where coefficients are put into the input queue are shown. In row σ the arrows indicate the extreme intervals of values which σ can take for these two execution traces. Obviously, the variable σ of the path constraint can take values $\sigma \in \{3, 4, 5, 6, 7\}$. Therefore, predicates of the form $(\sigma \leq 6)$ or $(\sigma \geq 4)$ would be violated.

4 Relation to other basic models of computation

Of course, the *SCF* model is a fairly general model. As it will be seen, it contains many well-known computation paradigms. The modeling power does neither come from the concept of hierarchical or parallel automata (as they

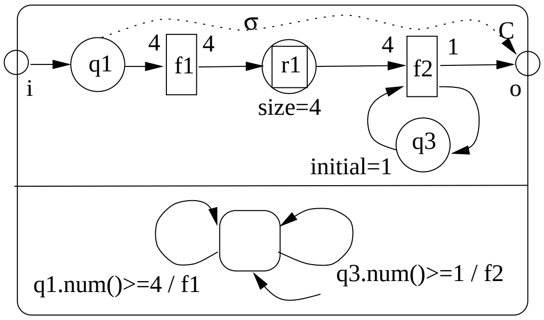

Figure 10: Example of a path constraint.

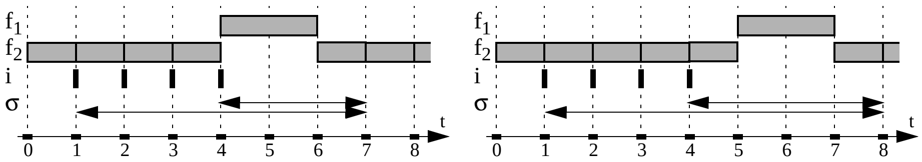

Figure 11: Possible execution traces of the component in Fig. 10.

can be transformed into simple automata without signals) nor from the concept of embedded components (as they can be flattened). Instead, the partition into a purely reactive part (state machine) without computations and a passive functional part is the main source for this capability.

On the other hand, it can not be expected that efficient analysis, code generation, and scheduling techniques exist in general. This subject will be tackled in Section 6. As it will be seen, the combination of embedded components, refinement, and abstraction leads to a new approach to solve these complex problems.

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

4.1 Communicating finite state machines

STATECHART-like [10] specifications and synchronous parallel state machines like ARGOS [17] are directly contained as the *SCF* model supports and- and or-substates. As two further examples, the communication mechanisms of the POLIS [1] model for the specification and design of embedded systems and that of the Communicating Finite State Machines are described in some detail.

Communicating Finite State Machines In the case of a Communicating Finite State Machine (CFSM), e.g., as in SDL process networks [18], asynchronously operating finite state machines communicate via FIFO ordered queues. A FSM, e.g., M_1, can write a value from a finite alphabet into a queue, e.g., q, during a transition. Furthermore, a FSM, e.g., M_2, can guard its transitions with predicates on the value of the first element in the queue, e.g., p. If the transition is taken, the element is removed from the queue. There are finitely many queues in the system.

In the *SCF* model, the finite state machines can be embedded into components, e.g., C_1 and C_2 for M_1 and M_2. The writing into and removing from queues can be modeled using functions, e.g., f_1 and f_2, with production and consumption rates of 1. The asynchronous reactions of the finite state machine are implemented using a finite state machine M in the top component with one state and loops for each finite state machine, see Fig. 12. M_1 and M_2 control the function activations. In this and the following examples, a signal named *step* is used to illustrate the triggering of subcomponents. The name *step* is not a reserved symbol.

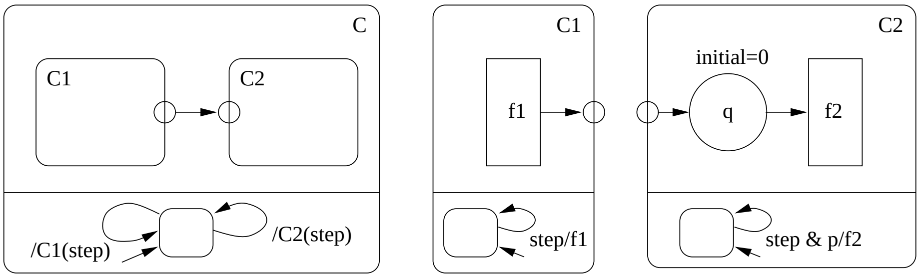

Figure 12: Representation of Communicating Finite State Machines in the *SCF* model.

POLIS The POLIS model [1] has been invented for designing control dominated embedded systems. Here, it will be shown how the communication mechanism can be represented in the *SCF* model. All finite state machines (FSMs), M_1 and M_2 here, operate asynchronously. They communicate via single element buffers represented by buffers, e.g., q. If 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 simply be represented as shown in Fig. 13. The essential difference between this and the CFSM model is that the storage units are registers instead of queues.

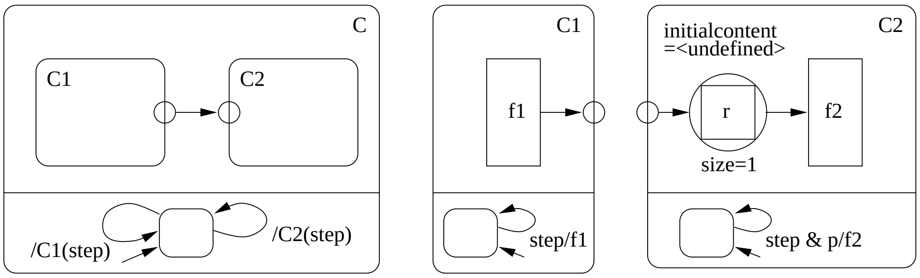

Figure 13: Representation of the POLIS model.

4.2 Marked graphs and synchronous dataflow graphs

Marked graphs [7] and synchronous dataflow (SDF) graphs [14, 15] can be represented as labeled directed graphs with nodes $v \in V$ which represent the operations of the system, edges $e \in E$ denoting the communication and the corresponding queues between the operations, a function $m : E \to \mathbf{Z}_{\geq 0}$ determining for each edge the actual number of tokens in the corresponding FIFO queue, a function $c : E \to \mathbf{Z}_{\geq 0}$ denoting the number of tokens removed from the queue if the node at its target fires, and a function $p : E \to \mathbf{Z}_{\geq 0}$ denoting the number of tokens added to the queue if the node at its source fires. A computation node 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 $e \in E$.

The *SCF* model behaving like a SDF graph can be constructed as follows. The whole graph is embedded into one component. Each node v is replaced by a function, each edge e is replaced by a concatenation of an edge, a queue, and another edge. The initial number of tokens in these queues are determined by the initial number of tokens on the edges of the SDF graph. The values $c(e)$ and $p(e)$ are written at the corresponding incoming and outgoing edges of the functions. The state machine of the component has one state with one self-loop transition for each node v. Each condition is a conjunction of predicates on the number of tokens in all input edges of v, e.g., $q.num() \geq c(e)$, and the action consists in an activation of the function corresponding to v, see Fig. 14.

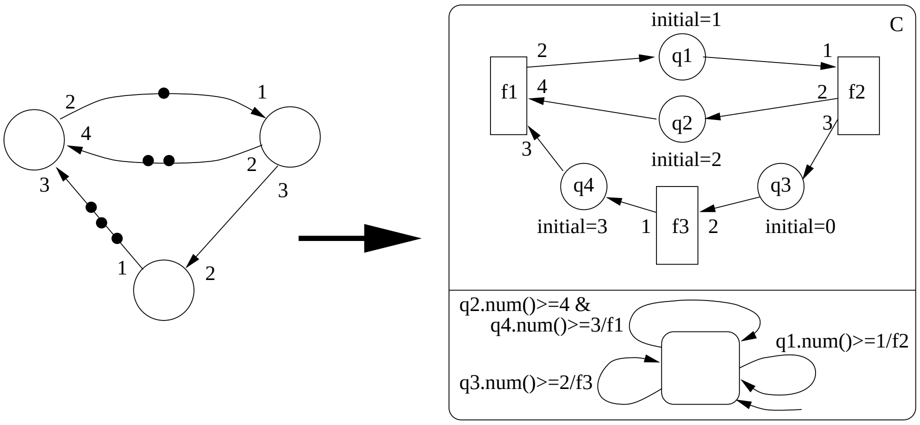

Figure 14: Example of a synchronous dataflow graph and its representation as *SCF* model with global control.

This is an example of a *global control strategy.* The finite state machine has access to all queues in the system. An example of a model with *local control strategy* is shown in Fig. 15.

The numbers of tokens in the queues correspond to those on the edges of the SDF graph if the latency of the functions (time difference between consumption and production) is zero.

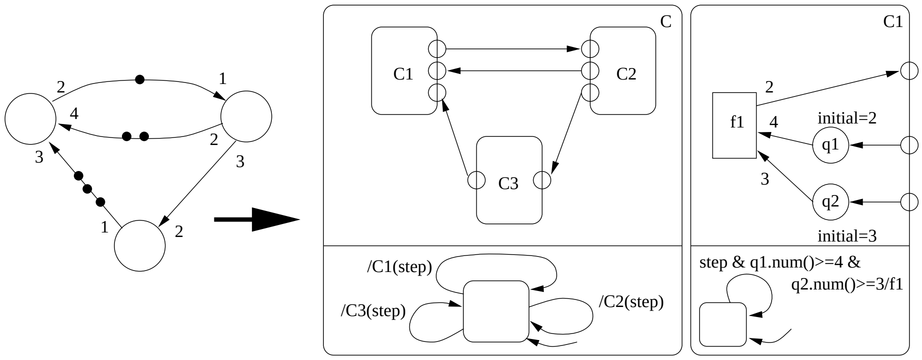

Figure 15: Example of a synchronous dataflow graph and its representation as *SCF* model with local control. Only embedded component C_1 is shown.

4.3 Cyclostatic dataflow graphs

In cyclostatic dataflow [3, 8], production and consumption rates of nodes $v \in V$ change periodically. Figure 16 shows the translation of this model of computation. To each node v, there corresponds now an embedded component. The state machine cycles through all possible consumption and production rates of the corresponding node v. The components representing dataflow nodes are connected as in Fig. 15.

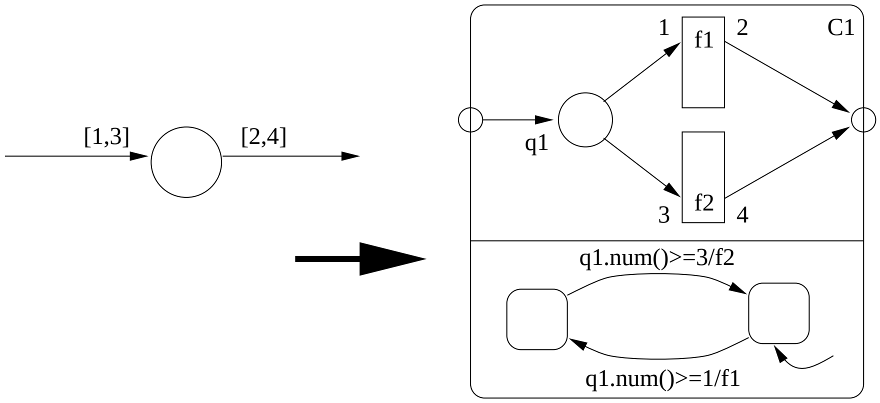

Figure 16: A cyclostatic SDF node.

4.4 Boolean and dynamic dataflow graphs

Boolean and dynamic dataflow graphs extend the previously described SDF model by introducing data dependent dataflow. In particular, in the BDF model, two additional types of nodes called SWITCH and SELECT are defined, see Fig. 17.

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 on the value

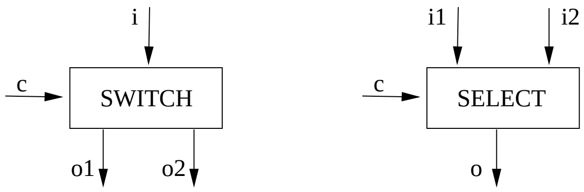

Figure 17: SWITCH and SELECT nodes in Boolean dataflow graphs.

$val(c) \in \{true, false\}$ of the first token to which output o_1 or o_2 the first token on the data input edge is transferred. The SELECT node acts similarly, i.e., a token on input i_1 or i_2 is transferred to output o if there is a token on c with value $val(c) = true$ or $val(c) = false$, respectively. Fig. 18 shows the corresponding *SCF* models.

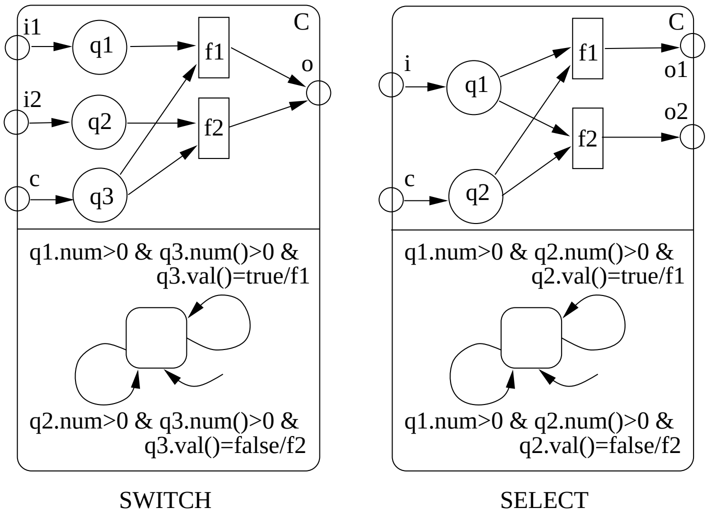

Figure 18: SWITCH and SELECT nodes in the *SCF* model.

As an example for a node type defined in Dynamic Data Flow Graphs, Fig. 19 shows a non-deterministic merge node and its equivalent *SCF* 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.

4.5 Petri nets

At a first glance, the *SCF* model seems to be almost equivalent to Colored Petri Nets (CPN) [12]. But there are several major differences which as well tune the Petri net model to the application domain of the *SCF* model and, at the same time, generalize it. The following differences can be noted.

- The storage units can be related to places in Petri nets. However, unlike

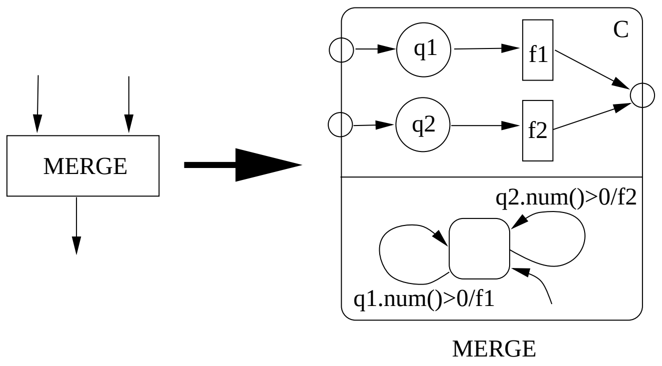

Figure 19: MERGE node in the *SCF* model.

the *SCF* queues, CPN places do not have FIFO behavior and there is no direct equivalent for *SCF* registers.

- 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 *SCF* model, these predicates can be different from the number of token removed while firing, e.g., it could be possible that a function is activated if there are at least 4 token 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 *SCF* model, this can be controlled by the finite state machine. This capability enables the simple consideration of limited resources.

5 Modeling examples

The purpose of this section is to show the versatility of the *SCF* model by the use of additional examples.

5.1 Interrupt

The first example describes a possible interrupt mechanism. It is supposed that the normal execution is carried out in the embedded component C_1. The interrupt request is passed to the system via adding a token to the *req* queue. The embedded component C_2 models the interrupt service routine. As shown in Fig. 20, a queue *rdy* is used to inform the state machine that the interrupt service routine has finished its computations.

Of course, the example only gives a possible basic mechanism. Many extensions can be thought of, for example, several interrupt service routines and vectorized interrupts.

5.2 Mode change

In many embedded systems, there is a special mechanism present to switch between different modes of computation, for example, 'emergency mode', 'reset',

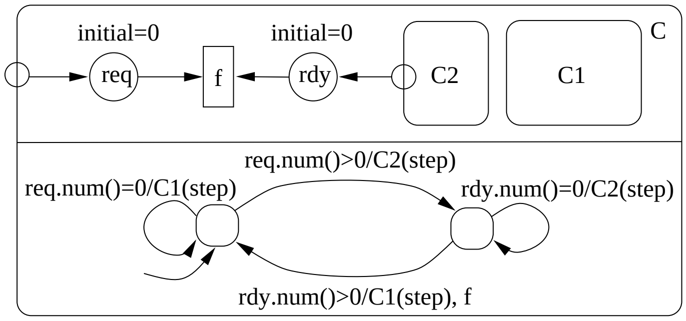

Figure 20: An interrupt mechanism.

'data input', or 'error correction'. The change of a mode is similar to an external interrupt and can be characterized as follows:

- The mode change can be initiated by either internal or external signalization. This situation is similar to an exception or interrupt (see Fig. 20).
- Usually, the change of the computation is not performed immediately as in the case of an interrupt. The current operation mode is driven to a safe state before it is left.
- The different operation modes work with the same state variables, at least to a certain extent. This is usually not the case for an interrupt service routine.

Fig. 21 shows a possible modeling of this behavior. There are two possible modes represented by the embedded components C_1 and C_2. They partially operate on a common state, exchanged via edges. The register *mode* contains a token whose value $val(mode) \in \{1, 2\}$ denotes the current mode. The queue *rdy* contains a token if one of the embedded components has finished and is ready for a mode change.

5.3 Scheduling

In a hierarchical approach to solve complex scheduling problems, it is necessary that the result of a partial scheduling can be represented in the same model. With this information, further scheduling steps can be performed. This stepwise refinement corresponds to the stepwise reduction of the non-determinism in the model. This section contains some examples of different scheduling mechanisms.

5.3.1 Static scheduling

As a first example, let us consider a purely static periodic schedule of the Synchronous Data Flow Graph shown in Fig. 14 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 to Fig. 14, only the state machine of the component C must be changed in order to take into account

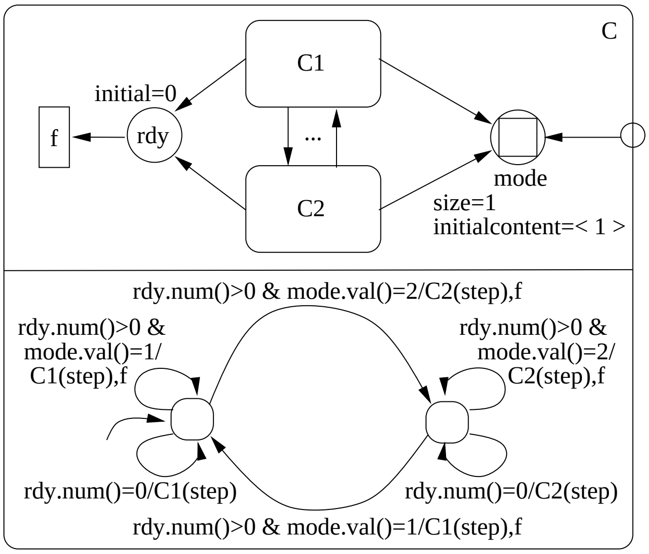

Figure 21: Mode change in the *SCF* model. The content of the mode register is initially set to 1.

the schedule. Fig. 22 shows two different possibilities, both reflecting the above described periodic schedule. The second possibility considers that the subsequence (f_2, f_3) occurs twice in the schedule.

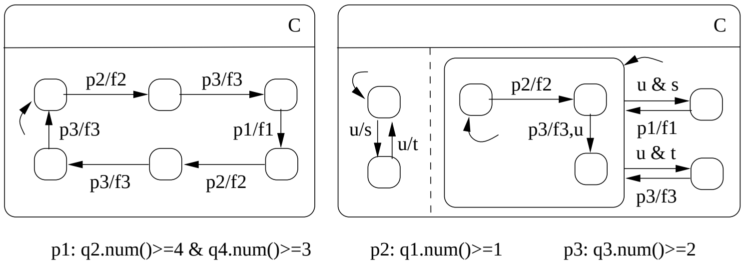

Figure 22: Two ways of static periodic scheduling for Fig. 14.

5.3.2 Quasi-static scheduling

In the above periodic static scheduling, the sequence of function invocations is constant during the computations. In a quasi-static scheduling, this sequence is composed of constant building blocks. The ordering of these building blocks depends on run-time decisions. In comparison to a completely dynamic scheduling, the amount of run-time decisions and the corresponding overhead and nondeterminism is greatly reduced. In comparison to a static schedule, the available resources can be utilized better.

As an example, we choose a *greatly* simplified algorithm for molecular dynamics simulation. The simulation algorithm contains 3 major tasks. In the

τ_F	τ_I	τ_P	$size_{AF}$	$size_{AC}$	$size_{CG}$	$size_{PL}$
15	1	5	0.3	2	0.6	10

Table 1: Typical parameters for molecular dynamic simulation. Latencies τ are in seconds, data sizes *size* are in MByte.

force calculation function F, the forces on the atoms AF are calculated based on the atom coordinates AC. In the integration function I, the forces on the atoms AF are used to calculate the new atom coordinates AC and the new centers of sub-molecules, called charge groups CG. These coordinates of charge groups are used in the function P to calculate the neighborhood list of each charge group, called pair list PL. Forces in F are calculated only between the atoms of neighbored charge groups. Because of the slowly moving atoms, the pairlist needs not to be updated every integration step.

Table 1 shows typical data sizes and computation times on a sequential processor.

The following representation in form of a *SCF* model supposes that the algorithm is executed on a sequential host workstation (integration function I) and a sequential co-processor[8] which calculates the forces (function F) and the pairlist (function P). The whole component C consists of two embedded components *Host* and *CoPro* operating concurrently. They exchange data via three queues, namely AC for the atom co-ordinates, AF for the atom forces and CG for the charge group co-ordinates. This situation is shown in Fig. 23.

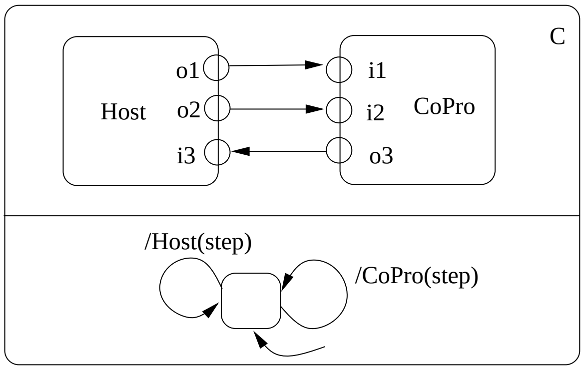

Figure 23: System for molecular dynamic simulation.

The embedded host component is shown in Fig. 24. The integration function I takes the atom forces from AF, produces the atom co-ordinates via o_1 and updates the charge group co-ordinates in queue CGQ. If a pairlist update via a neighborhood search is necessary, function T transmits these co-ordinates via o_2. The scheduling is represented in the state machine of the *Host*. The two hierarchical states correspond to two modes of operation. The left state machine initiates the updates of co-ordinates without a pairlist computation whereas the right one initiates two updates including a pairlist computation. The schedule

[8]Extensions to a parallel co-processor are easily possible.

takes into account that an overlapping computation of the pairlist and the forces could be possible in a parallel coprocessor. The parallel state machine models the decision whether a pairlist update is necessary (signal p) or not (signal f).

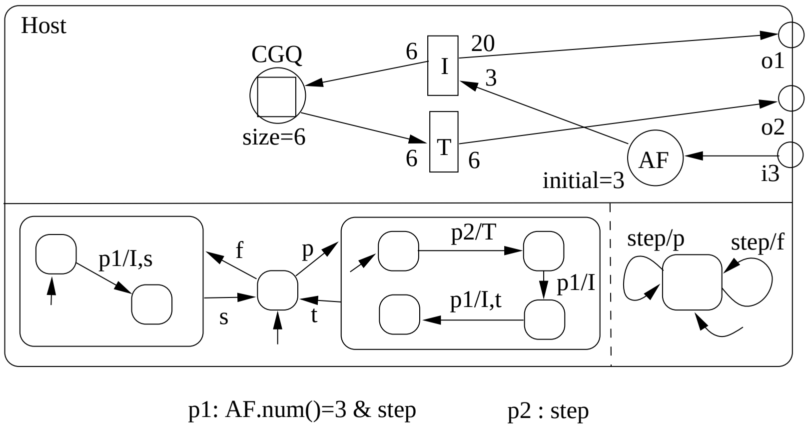

Figure 24: Host component.

The embedded coprocessor component is built in a similar way, see Fig. 25. The calculation of forces in F takes atom co-ordinates from AF and the pairlist from PLQ. The computation of the pairlist is done via a neighbourhood search in function P. It takes the charge group co-ordinates from CG. The two hierarchical states for the state machine again reflect the two modes of operation, i.e., update with and without pairlist computation. In a parallel co-processor, the pairlist computation (via the edge with label $p2/P$) could be executed in parallel with the first force calculation (via the edge with label $p1/F$) in the right hierarchical state. In this case, a slightly different state machine would reflect the new schedule.

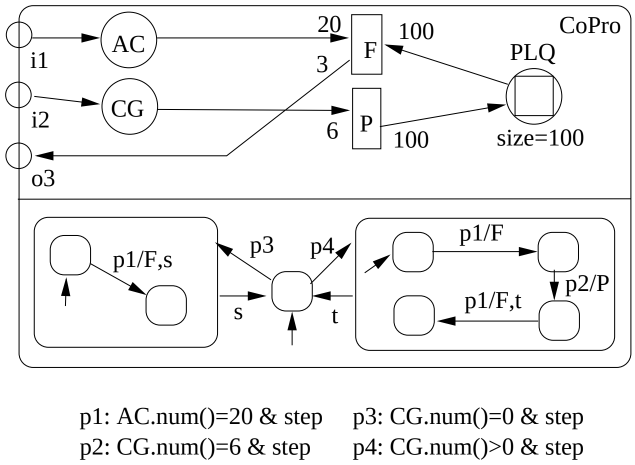

Figure 25: Coprocessor component.

5.3.3 Constant rate scheduling

As another example, we consider the schedule of a set of tasks where each task j has a fixed rate. This rate is the inverse of the period $P_j \in \mathbf{Z}_{>0}$ of the task j.

To this end, it is convenient to suppose a common period P which is the greatest common divisor of the task periods, i.e. $P = gcd\{P_j : \text{for all tasks } j\}$. This common period defines static time intervals of the form $t \in [kP, (k+1)P]$ for all $k \geq 0$. Within these intervals, the tasks are executed such that their periods are respected. In particular, a task j is executed in all intervals $[kP, (k+1)P]$ where k is a multiple of P/P_j. Depending on additional constraints such as dependencies and deadlines, the detailed schedule of all tasks within such an interval can be constraint further.

To illustrate the modeling of a constant rate schedule, the following simple example is considered. There are two tasks modeled by the embedded components C_1 and C_2 with the periods $P_1 = 2$ and $P_2 = 4$. Therefore, we have the common period $P = 2$. The schedule is driven by a clock generated by a function T with latency $\tau_T = 2$. The sub-sampling is modeled using two parallel state machines, see Fig. 26.

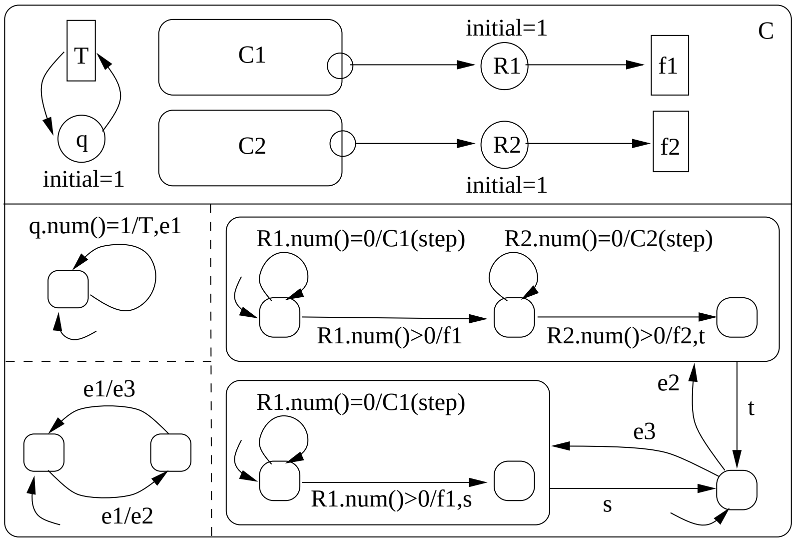

Figure 26: Example for constant rate scheduling.

The schedule of the tasks is modeled using a third parallel state machine. The two hierarchical states correspond to the two different schedules in the intervals. The queues $R1$ and $R2$ enable the tasks to signal that they finished computation. If there is a token in R_1 or R_2, the computations of the corresponding task finished.

Of course, there are many alternative models possible. Moreover, additional deadlines could be considered easily.

5.3.4 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 [16], where mathematical conditions are provided for assessing 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. 27, there are three tasks C_1, C_2, and C_3. The request for executing a task is signaled by putting a token in the corresponding input queue of the component. This could be done in an enclosing component by some sort of clock generator, see also Fig. 26. A task puts a token into its output queue if it finishes computation. The two state machines are responsible for detecting the end of a task and the priority scheduling, respectively.

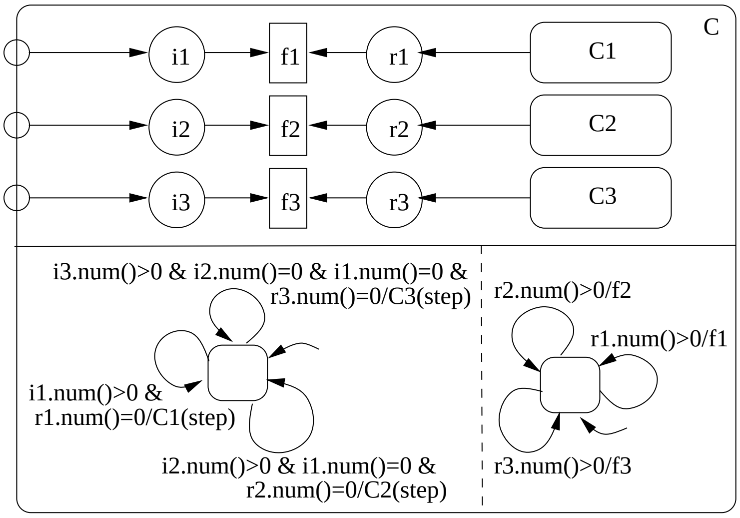

Figure 27: Example for rate monotonic scheduling.

6 Concluding remarks

As has been explained above, the *SCF* model enables the modeling of complex behavior which includes data-flow systems and state machines in particular. Consequently, it is hard to develop efficient methods for scheduling, code generation and analysis/verification.

To cope with this complexity, the following hierarchical step-by-step approach is advertised and supported by the *SCF* 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 *SCF* 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 previous chapters that the *SCF* model can represent the result of a (partial) schedule.

Further work should address the following problems:

Clustering Clustering is the inverse process of flattening. Methods should be developed which create embedded components containing a subset of the component's network. There are several possible application areas. During system design, one embedded component could be implemented as a module or block. Consequently, clustering adds structural information to a specification. During scheduling, one could adopt an hierarchical approach while first scheduling smaller subsystems, i.e. embedded components.

Compression Compression makes the network of an embedded component simpler without changing the external behavior. The purpose of the compression is to reduce the complexity of methods applied to the model without loosing correctness or quality.

Abstraction Abstraction means simplification of an embedded component. In contrary to the compression, the behavior of the embedded component is changed in a certain way. The particular kind of abstraction very much depends on the application area. It must be chosen such that methods applied to the system still give meaningful results.

A useful definition for many purposes is the following: A component B abstracts A if for any execution trace in A there exist an equivalent one in B. Equivalent traces produce the same token on output queues if given the same token on input queues. Therefore, abstraction increases the non-determinism in the model. Work to be done is to formally define abstraction, to define abstraction methods and to design methods which work with abstract embedded components.

Until now, the report contains no results on the analysis, verification and/or scheduling procedures. Several preliminary results are available in these areas, in particular for deadlock analysis and verification using symbolic model checking.

Another interesting aspect is the implementation of a *SCF* model in hardware or software or both. In particular, the state machines which model the scheduling can be transformed directly into the control automaton of an implementation.

References

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

[2] S. S. Bhattacharyya and E. A. Lee. Scheduling synchronous data flow graphs for efficient looping. *Journal of VLSI Signal Processing*, 6:271–288, 1993.

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

[4] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal on Computer Simulation*, 4:155–182, 1991.

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

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

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

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

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

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

[11] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ, 1985.

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

[13] E. A. Lee and T. M. Parks. Dataflow Process Networks. Technical Report UCB/ERL 94/53, Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A., 1994.

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

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

[16] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM*, 20(1):46–61, 1973.

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

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

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