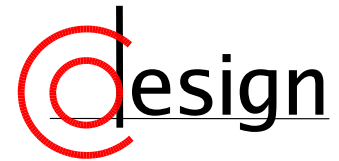


Friedrich-Alexander-Universität
Erlangen-Nürnberg



Syntax and execution behavior of SystemoC

Joachim Falk, Christian Haubelt, Jürgen Teich

Department of Computer Science 12
Hardware-Software-Co-Design
University of Erlangen-Nuremberg
Am Weichselgarten 3
D-91058 Erlangen, Germany

Co-Design-Report 04 - 2005

December 30, 2005

Contents

1	Introduction	3
2	<i>SystemoC</i> - syntax	4
2.1	Network graph	4
2.2	Actor classes	7
2.2.1	Declaration of the actor input and output ports	8
2.2.2	Declaration of the actor functionality	9
2.2.3	Declaration of the communication behavior	12
3	<i>SystemoC</i> simulation environment	18
	References	18

1 Introduction

Actor-based design is based on composing a system of communicating processes called *actors*, which can only communicate with each other via channels. However, *actor-based design* does not constrain the communication behavior of its actors therefore making analyses of the system in general impossible. In a *model-based design* methodology the underlying *Model of Computation* (MoC) is known additionally which is given by a predefined type of communication behavior and a scheduling strategy for the actors. We present a library based on the design language SystemC called *SysteMoC* which provides a simulation environment for model-based designs. The library-based approach unites the advantage of executability with analyzability of many expressive MoCs.

Due to rising design complexity, it is necessary to increase the level of abstraction at which systems are designed. This can be achieved by model-based design which makes extensive use of so-called *Models of Computation* [Lee02] (MoCs). MoCs are comparable to design patterns known from the area of software design [GHJV95]. On the other hand, industrial embedded system design is still based on design languages like C, C++, Java, VHDL, SystemC, and SystemVerilog which allow unstructured communication. Even worse, nearly all design languages are Turing complete making analyses in general impossible. This precludes the automatic identification of communication patterns out of the many forms of interactions, e.g., shared variables and various ways of message passing between processes.

Instead of a monolithic approach for representing an executable specification of an embedded system as done using many design languages, we will use a refinement of *actor-oriented* design. In actor-oriented design, *actors* only communicate with each other via *channels* instead of method calls as known in object-oriented design. *SysteMoC* is a library based on SystemC that allows to describe and simulate communicating actors, which are divided into their *actor functionality* and their *communication behavior* encoded as an explicit finite state machine. In the following, the syntax and semantics of *SysteMoC* designs is discussed.

Using *SysteMoC*, many important models of computation may be described such as SDF [LM87], CSDF [BELP96, EBLP94], Boolean Dataflow, Kahn Process Networks [Kah74], and many process networks, also communicating sequential processes (CSP) [Hoa85], and many others [TSZ⁺99, Lee97, Lee02, LSV98, EJL⁺02, Tei97].

A *SysteMoC* actor contains 3 basic elements:

- *Network graph*: Each application is modeled by a network graph of communicating actors.
- *Actor classes*: Each actor is defined by a class that contains several so-called *actions* that are basic functional blocks implemented as C++ methods that do

computations on tokens on inputs and output ports and the internal state of an actor. The actor itself thus possesses an implementation in the form of a C++ class, and each actor communicates with other actors through a certain number of input ports and a certain number of output ports. The basic entity of data transfer is regulated by the notion of *tokens*. Apart from the existence or absence of tokens, an actor may check and do computations also on values of these tokens. In this paper, we do not make any assumptions on the types of tokens exchanged between actors.

- *Firing finite state machine (FSM)*: The behavior of each actor is ruled by an explicit finite state machine that checks conditions on the input ports, output ports and internal state. If a certain firing rule is satisfied, a state transition will be taken. During the state transition, an *action* is called. Once this action has finished execution, the new state is taken. The complete state of an actor is described by this explicit state of the actor and the state as given by its (local) member variables. Depending on the model of computation, the actor ports may be connected to different types of so-called *channels* such as channels with FIFO semantics, or rendez-vous channels.

2 *SysteMoC* - syntax

A complete application is modeled by a set of actors and their interconnection using channels. The overall model is therefore a network of actors and channels.

2.1 Network graph

The creation of a *SysteMoC* design can be roughly divided into two subtasks: (i) The creation of a *network graph* for the design, e.g., as displayed in Figure 1 for an approximative square root algorithm, and (ii) the creation of all *actor classes* needed by the design, e.g., `SqrLoop` in Figure 2. The network graph is composed of *actor instances* of these actor classes, e.g., $a_1 - a_5$, which are connected via *channels*.

The approximative square root algorithm in Figure 1 is stimulated by an infinite sequence of input token values generated by the `Src` actor a_1 . These input token values are transported via channel c_1 to the `SqrLoop` actor a_2 which implements the error bound checking of the approximation algorithm. If the error bound is not satisfied, the input value will be send to actor a_3 via channel c_2 . This will eventually result in a new better approximated square root value in channel c_5 . This iteration repeats until the error bound is satisfied and the approximation result is forwarded via channel c_6 to the `Sink` actor a_5 .

In an actor-oriented design [Agh97], a model of computation [Lee02] defines the interaction policy between actors. Actors are objects which execute concurrently. An

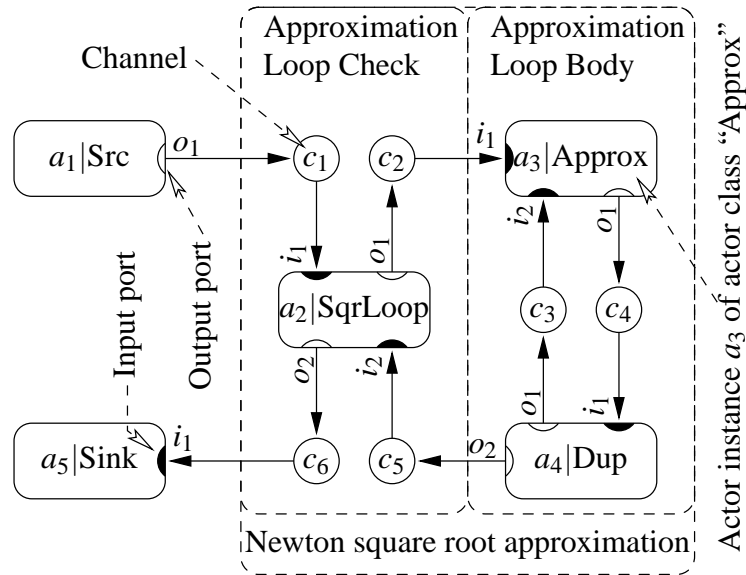


Figure 1: The *network graph* displayed above implements Newton’s iterative algorithm for calculating the square roots of an infinite input sequence generated by the `Src` actor a_1 . The square root values are generated by Newton’s iterative algorithm `SqrLoop` actor a_2 for the error bound checking and a_3 - a_4 to perform an approximation step. After satisfying the error bound, the result is transported to the `Sink` actor a_5 .

actor a can only communicate with other actors through its sets of *actor input and output ports* denoted $a.I$ and $a.O$, respectively. The actor ports are connected with each other via a communication medium called *channel*. The basic entity of data transfer is regulated by the notion of *tokens* which are transmitted via these channels. See Figure 1 for an example of a *network graph*, where $c_1 - c_6$ denote *FIFO* channels.

Actors are connected to other actors via channels. These connections are encoded in the *network graph*. In *SystemoC*, a *network graph* is represented as a C++ class derived from the base class `smoc_graph`, e.g., as seen in the following code for the above square root approximation algorithm example:

Example 2.1 Network graph corresponding to Figure 1:

```
// Declare network graph class SqrRoot
class SqrRoot: public smoc_graph {
protected:
    // Actors are C++ objects
    Src      src;      // Actor a1
    SqrLoop  sqrloop; // Actor a2
    Approx   approx;  // Actor a3
    Dup      dup;     // Actor a4
```

```

    Sink    sink;    // Actor a5
public:
    // Constructor of network graph class assembles network graph
    SqrRoot( sc_module_name name )
        : smoc_graph(name),
          src("src", 50),
          sqrloop("sqrloop"),
          approx("approx"),
          dup("dup"),
          sink("sink") {
    // The network graph is instantiated in the constructor
    // a1.o1 -> a2.i1 using FIFO standard size
    connectNodePorts(src.o1,    sqrloop.i1);
    // a2.o1 -> a3.i1 using FIFO standard size
    connectNodePorts(sqrloop.o1, approx.i1);
    // a3.o1 -> a4.i1 using FIFO size 1
    connectNodePorts(approx.o1, dup.i1,
                     smoc_fifo<double>(1) );
    // a4.o1 -> a3.i2 using FIFO standard size and
    // an initial sequence of 2
    connectNodePorts(dup.o1,    approx.i2,
                     smoc_fifo<double>() << 2 );
    // a4.o2 -> a2.i2 using FIFO standard size
    connectNodePorts(dup.o2,    sqrloop.i2);
    // a2.o2 -> a5.i1 using FIFO standard size
    connectNodePorts(sqrloop.o2, sink.i1);
    }
};

```

The actors of the network graph, e.g., $a_1 - a_5$ in Figure 1, are member variables. They can be parameterized via common C++ syntax in the constructor of the *network graph class*, e.g., `src("src", 50)`. The connections of these actors via FIFO channels are assembled in the constructor of the network graph class, e.g., `connectNodePorts (src.o1, sqrloop.i1)` to connect $a_1.o_1$ to $a_2.i_1$. The FIFO channels are created by the `connectNodePorts(o, i[, param])` function which creates a FIFO channel between output port o and input port i . The optional parameter `param` is used to further parameterize the created FIFO channel, e.g., `smoc_fifo<double>(1) << 2` is used to create a FIFO channel for double tokens of depth one with an initial token of value two. More formally, we can derive the following definition for a *network graph*:

Definition 2.1 (Network graph) A (general) network graph is a directed bipartite graph $g = (A, C, P, E)$ containing a set of actors A , a set of channels C , a channel parameter function $P : C \rightarrow \mathbb{N}_\infty \times V^*$ which associates with each channel $c \in C$ its

buffer size $n \in \mathbb{N}_\infty = \{1, 2, 3, \dots, \infty\}$, and possibly also a non-empty sequence $\mathbf{v} \in V^*$ of initial tokens¹, and finally a set of directed edges $E \subseteq (C \times A.I) \cup (A.O \times C)$, where $A.I = \bigcup_{a \in A} a.I$ and $A.O = \bigcup_{a \in A} a.O$ denote the sets of all actor input ports and all actor output ports in the network graph, respectively.

Each actor $a \in A$ may only communicate with other actors through its dedicated actor input ports $a.I$ and actor output ports $a.O$. Furthermore, the set of all actor input and actor output ports of all actors in the network graph is given by $A.P = A.I \cup A.O$.² However, the preceding definition still allows multiple readers and writers per FIFO channel. Therefore, we define a more restricted form of network graphs called *non-conflicting network graph* which allows only *point-to-point* connections per channel.

Definition 2.2 (Non-conflicting network graph) A non-conflicting network graph is a network graph where the edges are further constraint such that exactly one edge is incident to each actor port and the in-degree and out-degree of each channel in the graph is exactly one, i.e., $\forall p \in A.P : |((\{p\} \times C) \cup (C \times \{p\})) \cap E| = 1$ and $\forall c \in C : |(\{c\} \times A.I) \cap E| = 1 \wedge |(A.O \times \{c\}) \cap E| = 1$.

In the following, we assume that we are dealing only with non-conflicting network graphs, thus allowing us to simply omit the term non-conflicting.

2.2 Actor classes

An *actor* can be thought of as an object which maps sequences of token values on its input ports to sequences of token values on its output ports. In *SystemoC*, each actor is represented as an instance of an *actor class* which is derived from the C++ base class `smoc_actor`, e.g., as seen in the following example for the `SqrLoop` actor class.

Example 2.2 Definition of the `SqrLoop` actor class:

```
class SqrLoop
  // All actor classes must be derived
  // from the smoc_actor base class
  : public smoc_actor {
public:
  // Declaration of input and output ports
```

¹We use the $V^* = \bigcup_{n \in \mathbb{N} \cup \{0\}} V^n$ notation to denote the set of all *tuples* of V also called *finite sequences* of V .

Furthermore, we will use $V^{**} = \bigcup_{n \in \mathbb{N}_\infty \cup \{0\}} V^n$ to denote the set of all finite and infinite *sequences* of V [LSV98].

²We use the ‘.’-operator, e.g., $a.P$, for member access, e.g., P , of tuples whose members have been explicitly named in their definition, e.g., $a \in A$ from Definition 2.3. Moreover, this member access operator has a trivial pointwise extension to sets of tuples, e.g., $A.P = \bigcup_{a \in A} a.P$, which is also used throughout this document.

```

    smoc_port_in<double> ...
private:
    // Declaration of the actor functionality
    // via member variables and member functions
    ...

    // Declaration of states for the firing FSM
    smoc_firing_state start;
    ...
public:
    // Constructor responsible for building the
    // firing FSM and initializing the actor
    SqrLoop(sc_module_name name)
        : smoc_actor( name, start ) {
        ...
    }
};

```

Accordingly, each *actor instance*, in the following simply called *actor*, is a C++ object of its corresponding actor class. As can be seen in the above code, each definition of an actor class can be subdivided into three parts: (i) Declaration of the actor *input ports* and *output ports*, (ii) declaration of the actor *functionality*, and (iii) declaration of the actor *communication behavior*, encoded by an explicit *firing FSM*. More formally, we can derive the following definitions:

Definition 2.3 (Actor) *An actor is a tuple $a = (\mathcal{P}, \mathcal{F}, \mathcal{R})$ containing a set of actor ports $\mathcal{P} = I \cup O$ partitioned into actor input ports I and actor output ports O , the actor functionality \mathcal{F} and the firing FSM \mathcal{R} .*

The *actor state* $q = (q_{\text{func}}, q_{\text{firing}})$ is combined from the state stored in the actor functionality $q_{\text{func}} \in \mathcal{F} \cdot \mathcal{Q}_{\text{func}}$ and the state of the firing FSM $q_{\text{firing}} \in \mathcal{R} \cdot \mathcal{Q}_{\text{firing}}$, i.e., $q \in \mathcal{Q} = \mathcal{F} \cdot \mathcal{Q}_{\text{func}} \times \mathcal{R} \cdot \mathcal{Q}_{\text{firing}}$. The three parts of an actor can also be seen in Figure 2 which shows a graphical representation the actor defined in the Examples 2.2 - 2.7. In the following, the steps to construct these three parts will be explained in detail.

2.2.1 Declaration of the actor input and output ports

An actor may only communicate with other actors via tokens passing from output ports to input ports via channels as can be seen in Figure 1 where actor a_2 is connected via its input ports $a_2.I = \{i_1, i_2\}$ and output ports $a_2.O = \{o_1, o_2\}$ to all other actors in the network graph. The port declaration must be located in the *public* part of the actor class to allow to connect all these actors together in a network graph description. An example of a port declaration can be seen in the example below.

Example 2.3 Port declaration for the SqrLoop actor class:

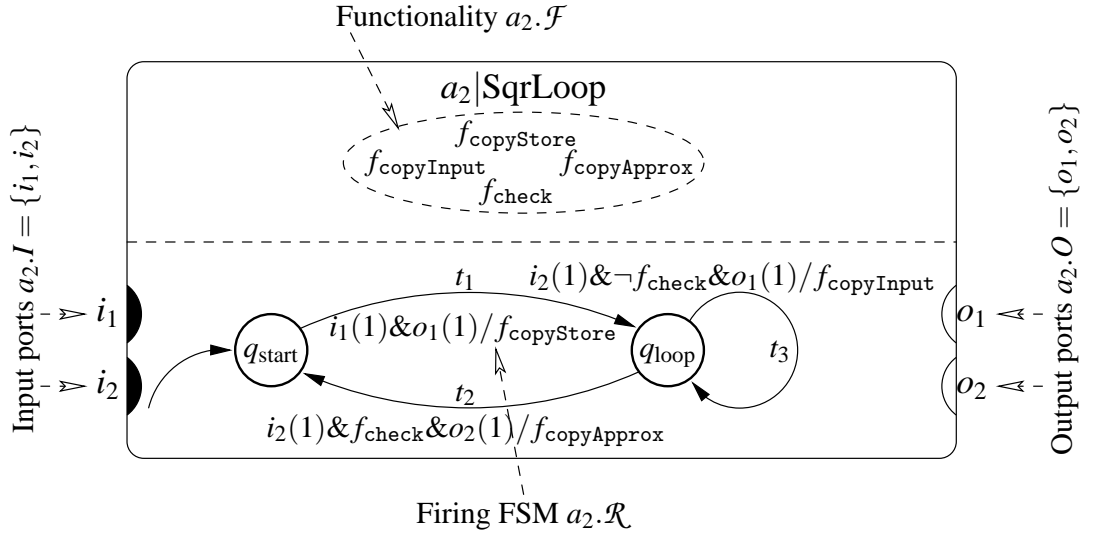


Figure 2: Visual representation of the SqrLoop actor a_2 used in the network graph displayed in Figure 1. The SqrLoop actor is composed of *input ports* and *output ports*, its *functionality*, and the *firing FSM* determining the communication behavior of the actor.

```

class SqrLoop: public smoc_actor {
public:
    // Declaration of input and output ports
    smoc_port_in<double> i1, i2;
    smoc_port_out<double> o1, o2;
private:
    ...
};

```

Please note that the usage of normal `sc_fifo` channels as provided by the SystemC language with `sc_fifo_in` and `sc_fifo_out` ports would not allow to separate actor functionality and communication behavior because these ports allow *destructive reads* and *non-destructive writes*. This would enable the actor functionality to actually consume and produce tokens contradicting the separation of these two aspects. For this reason, the *SystemMoC* library provides its own input and output port declarations `smoc_port_in` and `smoc_port_out`. These use the same concept of template parameters as standard SystemC ports, e.g., `sc_fifo_in`, to specify the type of token communicated.

2.2.2 Declaration of the actor functionality

The actor functionality is represented by member variables and member functions of the actor. These functions manipulate the so-called *functionality state* reflected by the

current values of the set of member variables of the actor. Some member functions of an actor are referenced by the firing FSM to calculate token values to be produced on the output ports. These member functions are called *actor actions* or *actions* for short, e.g., `copyStore` in Example 2.4. They can manipulate the functionality state. Other member functions are referenced by the firing FSM to decide if transitions in the FSM are enabled and can be taken. These member functions are called *actor guards* or *guards* for short, e.g., `check` in Example 2.4, and must not manipulate the functionality state. Therefore, these member functions are required to be declared as *const member functions*.

Example 2.4 Declaration of actor functionality:

```
class SqrLoop: public smoc_actor {
public:
    ...
private:
    // Declaration of the actor functionality
    // via member variables and member functions
    double tmp_i1;

    // action functions triggered by the
    // FSM declared in the constructor
    void copyStore() { o1[0] = tmp_i1 = i1[0]; }
    void copyInput() { o1[0] = tmp_i1; }
    void copyApprox() { o2[0] = i2[0]; }

    // and guards only used by the firing FSM
    bool check() const
        { return fabs(tmp_i1-i2[0]*i2[0]) < BOUND; }
    ...
};
```

The actor has four member functions including three actions (`copyStore()`, `copyInput()`, and `copyApprox()`), one guard (`check()`) as well as one member variable `tmp_i1`.

Definition 2.4 (Actor functionality) *The actor functionality of an actor $a \in A$ is a tuple $a.\mathcal{F} = (F, Q_{\text{func}}, q_{0\text{func}})$ containing a set of functions $F = F_{\text{action}} \cup F_{\text{guard}}$ partitioned into actions and guards, a set of functionality states Q_{func} (possibly infinite), and an initial functionality state $q_{0\text{func}} \in Q_{\text{func}}$.*

Example 2.5 For the example introduced in Example 2.4, we obtain the following

description:

$$\begin{aligned}
F_{\text{action}} &= \{f_{\text{copyStore}}, f_{\text{copyInput}}, f_{\text{copyApprox}}\} \\
F_{\text{guard}} &= \{f_{\text{check}}\} \\
S_{\text{func}} &= \mathbb{R} \\
q_{0\text{func}} &= 0
\end{aligned}$$

The actions $f_{\text{action}} \in F_{\text{action}}$ of the actor functionality map sequences of token values $\mathbf{v}_{i_1}, \mathbf{v}_{i_2}, \dots, \mathbf{v}_{i_{|a.I|}} \in V^*$ at the actor input ports $a.I$ into sequences of token values $\mathbf{v}_{o_1}, \mathbf{v}_{o_2}, \dots, \mathbf{v}_{o_{|a.O|}} \in V^*$ at the actor output ports $a.O$, thereby potentially also modifying the functionality state:

$$f_{\text{action}} : V^{N_{i_1}} \times V^{N_{i_2}} \dots \times V^{N_{i_{|a.I|}}} \times S_{\text{func}} \rightarrow V^{M_{o_1}} \times V^{M_{o_2}} \dots \times V^{M_{o_{|a.O|}}} \times S_{\text{func}}$$

In the above equation, $N_i \in \mathbb{N}_0$ denotes the number of tokens that may be read from the current token sequence \mathbf{v}_i at the i th actor input port. Similarly, $M_o \in \mathbb{N}_0$ denotes the number of tokens designated for the o th actor output port the values of which are computed by the action. Note that in the most general case of an actor, the number of token values read as well as the number of token values computed by an action may be dependent also on the state of the actor. The same holds, of course also for the values computed by the action. Hence, N_i and M_o may be also state-dependent in the most general case. We assume for now that N_i and M_o denote statically computable fixed upper bounds on the length of sequences of tokens the values of which are used/computed by an action. In general, one must be very careful, however, about treating actions and guards as functions in the pure mathematical sense. Finally, please note also that actions and guards are not responsible for determining how many tokens will be consumed and how many tokens will be produced each time an actor processes tokens. This concern is separated from the computation of token values and ruled uniquely by the firing finite state machine that will be described next. Before describing the notion of the firing state machine, we will introduce the difference between actions and guards.

A guard $f_{\text{guard}} \in F_{\text{guard}}$ of the actor functionality maps sequences of token values $\mathbf{v}_{i_1}, \mathbf{v}_{i_2}, \dots, \mathbf{v}_{i_{|a.I|}} \in V^*$ at the actor input ports $a.I$ and the functionality state to a boolean value:

$$f_{\text{guard}} : V^{N_{i_1}} \times V^{N_{i_2}} \dots \times V^{N_{i_{|a.I|}}} \times S_{\text{func}} \rightarrow \{\text{false}, \text{true}\}$$

In the above equation, $N_i \in \mathbb{N}_0$ denotes the number of tokens on the input port i needed for computing the boolean guard decision. Note that similar to actions, the sequence lengths N_i may also be dependent on the current functionality state of the actor in the most general case and must hence be considered to be upper bounds in the above equation.

In summary, there are, however, two fundamental differences of actions and guards: (i) A guard function just returns a boolean value instead of computing values of tokens for output ports, and (ii), a guard must be side-effect free in the sense that it must not be able to change the functionality state. In our implementation, we guarantee this second property by requiring that guards need to be declared as *const member functions*.

The input values for the actor functionality are provided by the firing FSM which retrieves input values from the tokens in the FIFO channels connected to the actor input ports. Output values from actions f_{action} are used by the firing FSM to generate tokens for the FIFO channels connected to the actor output ports.

2.2.3 Declaration of the communication behavior

The consumption and production of tokens is locally triggered by transitions of an explicit *firing FSM* required in each actor. The purpose and advantage of this clear separation of that part that does computation on token values (in actions and guards) from the control of the behavior of an actor in particular to our *SysteMoC* approach and inspired by the following advantages:

- *recognizability*: recognize important data-flow models of computation such as SDF, and CSDF just from the complexity of the firing FSM.
- *analyzability*: As a consequence of being able to detect important well-known models of computation within *SysteMoC* actors and actor network graphs, many important and well-known analysis algorithms such as boundedness of memory, liveness and periodicity properties may be applied immediately.
- *optimizability*: As an immediate consequence, buffer minimization and scheduling algorithms may be applied on individual or subgraphs of actors.
- *simulatability*: Finally, even most complex actor networks may be handled for which no formal analysis techniques are known by simply simulating the network of actors. As *SysteMoC* is built on top of SystemC, an event-driven simulation of the exact timing and concurrency among actors is immediately possible.
- *refinement*: We expect to show another important feature of *SysteMoC* in the future, namely a transformative refinement of actor code: We intend to apply important refinement transformations towards a final target implementation by providing transformations on the specification, and finally, also automatic platform-based automatic code synthesis is envisioned. This will be, however, a topic of future work.

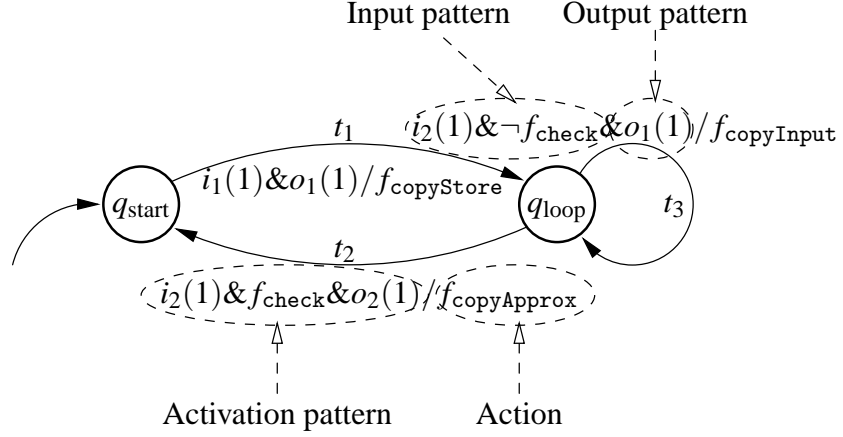


Figure 3: Visual representation of the *firing FSM* of the `SqrLoop` actor a_2 shown in Figure 1. The `SqrLoop` actor controls the number of iterations performed by Newton’s square root algorithm.

The notion of firing FSM is similar to the concepts introduced in SPI [ZER⁺99] and an extension of the *finite state machines* in FunState [STZ⁺01] by allowing requirements for a minimum of space available in output channels before a transition can be taken. The states of the firing FSM are called *firing states*, directed edges between these firing states are called *firing transitions* or *transitions* for short. Each transition is annotated with an *activation pattern*, a boolean expression which decides if the transition can be taken, and an *action* from the *actor functionality* which is executed if the transition is taken. An *activation pattern* may consist of an *input pattern* and an *output pattern*. Input (output) patterns are responsible for checking conditions on the actor input (output) ports, respectively. More formally, we derive the following two formal definitions:

Definition 2.5 (Firing FSM) *The firing FSM of an actor $a \in A$ is a tuple $a.\mathcal{R} = (T, Q_{\text{firing}}, q_{0\text{firing}})$ containing a finite set of firing transitions T , a finite set of firing states Q_{firing} and an initial firing state $q_{0\text{firing}} \in Q_{\text{firing}}$.*

Definition 2.6 (Transition) *A firing transition is a tuple $t = (q_{\text{firing}}, k, f_{\text{action}}, q'_{\text{firing}}) \in T$ containing the current firing state $q_{\text{firing}} \in Q_{\text{firing}}$, an activation pattern k , the associated action $f_{\text{action}} \in a.\mathcal{F}$, and the next firing state $q'_{\text{firing}} \in Q_{\text{firing}}$. The activation pattern k is a boolean function which decides if transition t can be taken (true) or not (false).*

Example 2.6 In the above figure, the firing FSM of the `SqrLoop` is shown. From the q_{start} state only the transition t_1 can be taken, which blocks until at least one token is available on input port i_1 and one token can be written on output port o_1 . The first token at input port i_1 represents the input value for the square root algorithm. It is

stored and forwarded by action $f_{\text{copyStore}}$ via port o_1 to the approximation loop body $a_3 - a_4$ of Newton's algorithm.

From state q_{loop} , either the transition t_2 can be taken if the approximation satisfies the error bound BOUND , or the transition t_3 if another approximation step is necessary. This termination criteria is determined by the guard f_{check} . Furthermore, both transitions t_2 and t_3 can only be taken if at least one approximation value (token) is available via port i_2 . Finally, for transition t_2 and t_3 to be ready to be taken, at least space to write one token on output port o_2 and output port o_1 must be available, respectively. Whereas the transition t_2 forwards the square root approximation to the Sink actor a_5 and transition t_3 forwards the input value for the square root algorithm again to the approximation loop body to calculate a refined approximation.

As said before, the activation pattern k may consist of patterns specifying the availability of tokens on the input ports and the availability of free space in the output ports of an actor. In this paper, we will not formally define activation patterns. Instead, we introduce them throughout our running example.

Note that in case at a certain instant of time, more than one transition should be ready to be taken, we assume that one of these transitions is chosen non-deterministically.

In the following Example 2.7, the *SystemoC* representation of the firing FSM of the SqrLoop actor a_2 , also visually represented in Figure 3, is given.

Example 2.7 Declaration of the firing FSM:

```
class SqrLoop: public smoc_actor {
    ...
    // Declaration of states for the firing FSM
    smoc_firing_state start, loop;
public:
    // Constructor responsible for declaring the
    // firing FSM and initializing the actor
    SqrLoop(sc_module_name name)
        : smoc_actor( name, start /* start state of firing FSM */ ) {
        // Declaration of start state consisting
        // of one outgoing transition t1
        start =
            // transition t1
            // with input pattern requiring at least one token
            // in the FIFO connected to input port i1
            i1(1) >>
            // with output pattern requiring at least space for one token
            // in the FIFO connected to output port o1
            o1(1) >>
            // has action SqrLoop::copyStore and next state loop
    }
```

```

        CALL(SqrLoop::copyStore)          >> loop
    ;
// Declaration of loop state consisting of two transitions t2 and t3
loop =
    // transition t2
    // with input pattern requiring at least one token
    // in the FIFO connected to input port i2 and
    // that guard SqrLoop::check be true
    (i2(1) && GUARD(SqrLoop::check)) >>
    // with output pattern requiring at least space for one token
    // in the FIFO connected to output port o2
    o2(1) >>
    // has action SqrLoop::copyApprox and next state start
    CALL(SqrLoop::copyApprox)          >> start
// transition t3
    // with input pattern requiring at least one token
    // in the FIFO connected to input port i2 and
    // that guard SqrLoop::check be false
    | (i2(1) && !GUARD(SqrLoop::check)) >>
    // with output pattern requiring at least space for one token
    // in the FIFO connected to output port o1
    o1(1) >>
    // has action SqrLoop::copyInput and next state loop
    CALL(SqrLoop::copyInput)          >> loop
    ;
}
};

```

In *SystemoC* firing states are represented as instances of the class `smoc_firing_state` and declared as member variables of the corresponding actor class. One firing state is selected as the start state by passing it to the `smoc_actor` base class of the actor class. The firing FSM is specified in the constructor of the actor class by assigning each firing state its set of outgoing transitions. The syntax used in the above example to declare the firing FSM is given in the following as extended *Backus-Naur form* (BNF):

Syntax 2.1 Extended Backus-Naur form for firing FSM declarations:

```

StateDefinition ::= FiringState '=' OutgoingTransitions ';'

OutgoingTransitions ::= Transition '|' OutgoingTransitions |
                    Transition ;

Transition ::= ActivationPattern '>>' Action '>>' FiringState |
            Action '>>' FiringState |

```

```

ActivationPattern      '>>' FiringState ;

ActivationPattern ::= InputPattern '>>' OutputPattern |
                    InputPattern      |
                    OutputPattern ;

Action               ::= 'CALL' '(' <actor action> ')' ;

InputPattern         ::= '(' InputPattern ')' |
                    InputPattern '&&' InputPattern |
                    InputExpression |
                    Expr ;

InputExpression      ::= InputPort '.getAvailableTokens()' '>=' Expr |
                    InputPort '.getAvailableTokens()' '>' Expr |
                    InputPort '(' Integer ')' ;

InputPort            ::= <actor input port>

OutputPattern        ::= '(' OutputPattern ')' |
                    OutputPattern '&&' OutputPattern |
                    OutputExpression ;

OutputExpression     ::= OutputPort '.getAvailableSpace()' '>=' Expr |
                    OutputPort '.getAvailableSpace()' '>' Expr |
                    OutputPort '(' Constant ')' ;

OutputPort           ::= <actor output port>

```

FiringStates are defined by assigning them their set of OutgoingTransitions. This transition set is created by combining their member Transitions via the | operator. The transitions themselves are created by combining an InputPattern, an OutputPattern, an Action, and the next FiringState via the »-operator. Whereas input pattern, output pattern, and action are optional, but at least one of them must be present. If an action is used the included <actor action> is declared in the actor functionality, as defined previously in Subsection 2.2.2.

As can be seen in the Example 2.7 input and output patterns are assembled via the &&-operator³ from InputExpressions and OutputExpressions respectively. Whereas input expressions check available tokens on input ports, e.g., i1(1) to check that at least one token is available on port i1, and output expressions check available space on output ports, e.g., o1(1) to check that at least one token can be written on port o1. The input expression i1(n) and output expression o1(m)

³C++ logical AND

are used as shortcuts for the longer forms `i1.getAvailableTokens() >= n` and `o1.getAvailableSpace() >= m` respectively.

As can be seen in the previous syntax definition input patterns can have additional *transition expressions* (Expr) checking, e.g., token values on input ports or the functionality state of the actor. The syntax rules for legal compositions of these transition expressions is given below.

Syntax 2.2 Extended Backus-Naur form of transition expressions:

```
Expr ::= Expr '+' Expr | Expr '-' Expr |
      Expr '*' Expr | Expr '/' Expr |
      Expr '==' Expr | Expr '!=' Expr |
      Expr '<' Expr | Expr '<=' Expr |
      Expr '>' Expr | Expr '>=' Expr |
      Expr '^' Expr | Expr '&' Expr |
      Expr '|' Expr | Expr '&&' Expr |
      Expr '||' Expr | '(' Expr ')' |
      '!' Expr | '~' Expr |
Terminal ;

Terminal ::= Constant | Variable | Guard | Token ;

Constant ::= <C++ integer expression> ;

Variable ::= 'VAR' '(' <actor member variable> ')' ;

Guard ::= 'GUARD' '(' <actor guard member function> ')' ;

Token ::= InputPort '.getValueAt' '(' Constant ')' ;
```

The transition expressions can be combined via C++ operators listed in the preceding syntax definition, e.g., + for arithmetic addition. The precedence of these operators is exactly their C++ precedence. A transition expression can be combined from the following terminals: (i) the `Constant`-terminal represents a C++ expression which evaluates at instantiation time of the transition expression to a C++ integer, e.g., `sqr(a+5)`. A later value change of a variable included in this C++ expression, e.g., `a`, will not change the value of instantiated `<Integer>`-terminals containing this variable. (ii) the `Variable`-terminal represents a actor member variable. A later value change of this variable will change the value of the `<Variable>`-terminal. (iii) the `Guard`-terminal represents an actor guard, as defined previously in Subsection 2.2.2. The value of the `Guard`-terminal is returned by its guard function and may depend on token values and the functionality state of the actor. (iv) the `Token`-terminal, e.g., `i.getValueAt(n)`, represents the `n`th token on an actor input port `i`. Where `n` is starting from zero, e.g., `i.getValueAt(0)` is the first token in the channel connected to input port `i`.

3 *SysteMoC* simulation environment

The execution of *SysteMoC* models can be divided into three phases: (i) checking for enabled transitions for each actor, (ii) selecting and executing one enabled transition per actor, and (iii) consuming and producing tokens needed by the transition. Note that step (iii) might enable new transitions. The activation patterns discussed above decide if a transition is enabled. Moreover, our activation patterns encode both step (i) and step (iii) of the execution phases, because each transition communicates the shortest possible prefix sequence on each input and output port still satisfying the activation pattern.

In order to implement a simulation environment for our *SysteMoC* library, we have several choices: More traditional approaches to encode these conditions would depend on callback functions for parts of the condition for which compile time code generation should be performed and use dynamic assembly of parts of the condition which should be available at runtime, e.g., by operator overloading to build an AST of the expression at runtime to express a sensitivity list. In the first case, a standard C++ compiler is not sufficient to extract the AST of the callback function from the source code and provide the simulation kernel with the information. In the second case, the simulation kernel is provided with the AST of the expression, but a costly interpretation phase is necessary to evaluate it.

To overcome these drawbacks, we model these conditions with *expression templates* [Vel95]. Using expression templates allows us to use both compile time code transformation and to derive at C++ compile time the *abstract syntax tree* (AST) for our activation patterns enabling: (i) extraction of the FIFO channels used in an activation pattern to generate sensitivity lists, (ii) compile time code generation for parts of an activation pattern only dependent on the actor state, e.g., as seen in Figure 4, or (iii) generation of an XML representation of the firing FSM, e.g., as seen in Figure 5, for later usage in the design flow.

As an example, we use the activation pattern on transition t_2 of the `SqrLoop` actor a_2 , as shown in Figure 4. The constructed expression template for an activation pattern is a tree of nested template types which corresponds to the *abstract syntax tree* of the activation pattern.

The actor state-dependent AST part is only evaluated after its corresponding sensitivity AST evaluates to `true`. Note that in general, arbitrarily complex parts dependent on the actor state of an activation pattern can be identified. For these actor state dependent AST parts, dedicated code is generated at C++ compile time for their evaluation.

References

- [Agh97] G. Agha. Abstracting interaction patterns: A programming paradigm for open distribute systems, 1997.

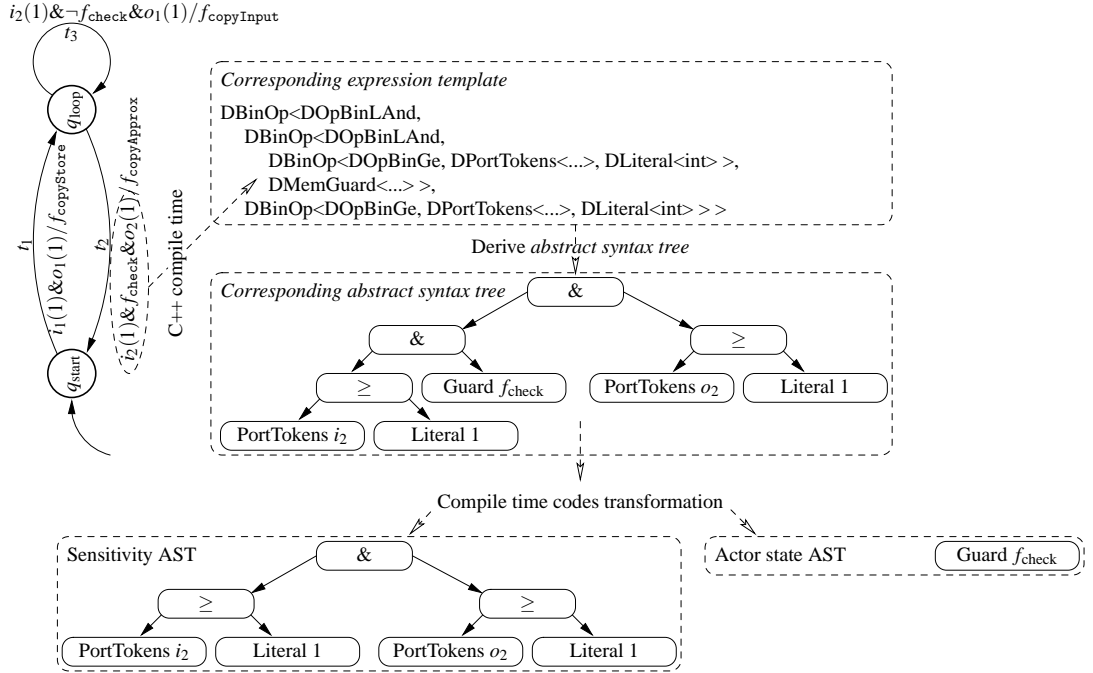


Figure 4: Compile time code transformation of an activation pattern into a sensitivity list used for scheduling and a functionality state-dependent part used to check transition readiness after the scheduling step.

- [BELP96] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-Static Dataflow. *IEEE Transaction on Signal Processing*, 44(2):397–408, February 1996.
- [EBLP94] Marc Engels, Greet Bilsen, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow: Model and implementation. In *Proc. 28th Asilomar Conf. on Signals, Systems, and Computers*, pages 503–507, Pacific Grove (U.S.A.), November 1994.
- [EJL⁺02] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jee Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yhong Xiong. Taming heterogeneity - the Ptolemy approach. In *Proceedings of the IEEE*, 2002.
- [GHJV95] R. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, April 1985.
- [Kah74] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.
- [Lee97] Edward A. Lee. A denotational semantics for dataflow with firing. Technical report, EECS, University of California, Berkeley, CA, USA 94720, 1997.
- [Lee02] Edward A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press London, London, September 2002.
- [LM87] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, volume 75(9), pages 1235–1245, September 1987.

```

<transition nextstate="id9" action="SqrLoop::copyApprox">
  <ASTNodeBinOp valueType="b" opType="DOpBinLAnd">
    <lhs><ASTNodeBinOp valueType="b" opType="DOpBinLAnd">
      <lhs><ASTNodeBinOp valueType="b" opType="DOpBinGe">
        <lhs><PortTokens valueType="j" portid="id6"/></lhs>
        <rhs><Literal valueType="j" value="1"/></rhs>
      </ASTNodeBinOp></lhs>
      <rhs><MemGuard valueType="b" name="SqrLoop::check"></rhs>
    </ASTNodeBinOp></lhs>
    <rhs><ASTNodeBinOp valueType="b" opType="DOpBinGe">
      <lhs><PortTokens valueType="j" portid="id8"/></lhs>
      <rhs><Literal valueType="j" value="1"/></rhs>
    </ASTNodeBinOp></rhs>
  </ASTNodeBinOp>
</transition>

```

Figure 5: XML representation of the transition t_2 of the SqrLoop actor a_2 including the *abstract syntax tree* derived from the activation pattern used in the transition.

- [LSV98] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.
- [STZ⁺01] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, J. Teich, and M. Gries. Symbolic scheduling based on the internal design representation FunState. *IEEE Trans. on VLSI Systems*, 9(4):522–544, 2001.
- [Tei97] Jürgen Teich. *Digitale Hardware/Software-Systeme*. Springer, Berlin Heidelberg, 1997. ISBN 3-540-62433-3.
- [TSZ⁺99] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich. FunState - An Internal Design Representation for Codesign. In *Proc. ICCAD'99, the IEEE/ACM Int. Conf. on Computer-Aided Design*, pages 558–565, San Jose, U.S.A., November 1999.
- [Vel95] Todd Veldhuizen. Expression Templates. In *C++ Report*, Vol. 7 No. 5, pages 26–31. SIGS Publications, New York, June 1995.
- [ZER⁺99] D. Ziegenbein, R. Ernst, K. Richter, L. Thiele, and J. Teich. SPI - an internal representation for heterogeneously specified embedded systems. In *Proc. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 160–169, Braunschweig, Germany, February 1999.