SysteMoC 2.0 Documentation

Joachim Falk

Lehrstuhl für Informatik 12
(Hardware-Software-Co-Design)
Universität Erlangen-Nürnberg
Cauerstraße 11
91058 Erlangen

Co-Design-Report 3-2006

February 12, 2018

Contents

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* [FHT06] 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 2.1 for an approximative square root algorithm, and (ii) the creation of all *actor classes* needed by the design, e.g., `SqrLoop` in Figure 2.2. The network graph is composed of *actor instances* of these actor classes, e.g., a_1 - a_5, which are connected via *channels*.

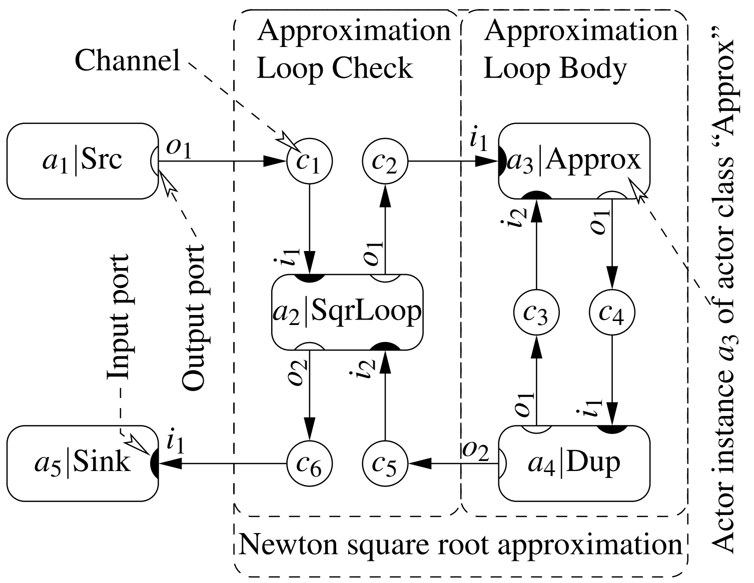


Figure 2.1: The *network graph* displayed above implements Newton's iterative algorithm for calculating the square roots of an infinite input sequence generated by the `Src` actor a_1. The square root values are generated by Newton's iterative algorithm `SqrLoop` actor a_2 for the error bound checking 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.

The approximative square root algorithm in Figure 2.1 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 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 2.1 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 2.1:

```
// 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("a1", 50),
      sqrloop("a2"),
      approx("a3"),
      dup("a4"),
      sink("a5") {
    // 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 2.1, are member variables. They can be parameterized via common C++ syntax in the constructor of the *network graph class*, e.g., `src("a1", 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 $P : C \to \mathbb{N}_\infty \times V^*$ 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 tokens*[1]*, 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

[1] We 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].

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 $A.\mathcal{P} = A.I \cup A.O$.[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.,* $\forall p \in A.\mathcal{P} : |((\{p\} \times C) \cup (C \times \{p\})) \cap E| = 1$ *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
  smoc_port_in<double>  ...
private:
  // Declaration of the actor functionality
  // via member variables and member functions
  ...

  // Declaration of states for the firing FSM
```

[2] We use the '.'-operator, e.g., $a.\mathcal{P}$, for member access, e.g., $\mathcal{P}$, 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., $A.\mathcal{P} = \bigcup_{a \in A} a.\mathcal{P}$, which is also used throughout this document.

```
  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 $q_{\text{func}} \in \mathcal{F}.Q_{\text{func}}$ and the state of the firing FSM $q_{\text{firing}} \in \mathcal{R}.Q_{\text{firing}}$, i.e., $q \in Q = \mathcal{F}.Q_{\text{func}} \times \mathcal{R}.Q_{\text{firing}}$. The three parts of an actor can also be seen in Figure 2.2 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 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 2.1 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:

```
class SqrLoop: public smoc_actor {
public:
  // Declaration of input and output ports
  smoc_port_in<double>  i1, i2;
```

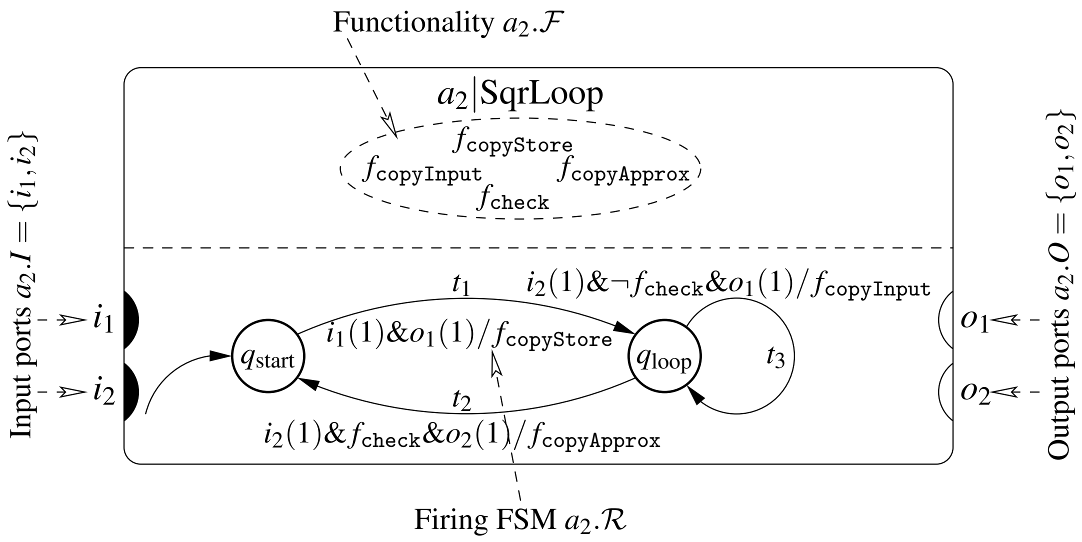

Functionality $a_2.\mathcal{F}$
$a_2|$SqrLoop
$f_{\texttt{copyStore}}$
$f_{\texttt{copyInput}}$
$f_{\texttt{copyApprox}}$
$f_{\texttt{check}}$
Input ports $a_2.I = \{i_1, i_2\}$
i_1
i_2
t_1
$i_2(1)\&\neg f_{\texttt{check}}\&o_1(1)/f_{\texttt{copyInput}}$
$i_1(1)\&o_1(1)/f_{\texttt{copyStore}}$
q_{start}
q_{loop}
t_3
t_2
$i_2(1)\&f_{\texttt{check}}\&o_2(1)/f_{\texttt{copyApprox}}$
o_1
o_2
Output ports $a_2.O = \{o_1, o_2\}$
Firing FSM $a_2.\mathcal{R}$

Figure 2.2: Visual representation of the `SqrLoop` actor a_2 used in the network graph displayed in Figure 2.1. The `SqrLoop` actor is composed of *input ports* and *output ports*, its *functionality*, and the *firing FSM* determining the communication behavior of the actor.

```
  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 *SysteMoC* 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 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 an 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()`, `copy Input()`, 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:

$$
\begin{aligned}
F_{\text{action}} &= \{f_{\texttt{copyStore}}, f_{\texttt{copyInput}}, f_{\texttt{copyApprox}}\} \\
F_{\text{guard}} &= \{f_{\texttt{check}}\} \\
Q_{\text{func}} &= \mathbb{R} \\
q_0\text{func} &= 0
\end{aligned}
$$

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

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

In the above equation, $N_i \in \mathbb{N}_0$ denotes the number of tokens that are consumed from the current token sequence $\mathbf{v}_i$ at the ith actor input port by the firing FSM after the action f_{action} has finished. Similarly, $M_o \in \mathbb{N}_0$ denotes the number of tokens designated for the oth actor output port produced by the firing FSM after the action f_{action} has computed the associated values for these tokens. 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 $\mathbf{v}_{i_1}, \mathbf{v}_{i_2}, \ldots, \mathbf{v}_{i_{|a.I|}} \in V^*$ at the actor input ports $a.I$ and the functionality state to a boolean value:

$$f_{\text{guard}} : V^{N_{i_1}} \times V^{N_{i_2}} \ldots \times V^{N_{i_{|a.I|}}} \times Q_{\text{func}} \to \{\text{false}, \text{true}\}$$

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 Actor 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.

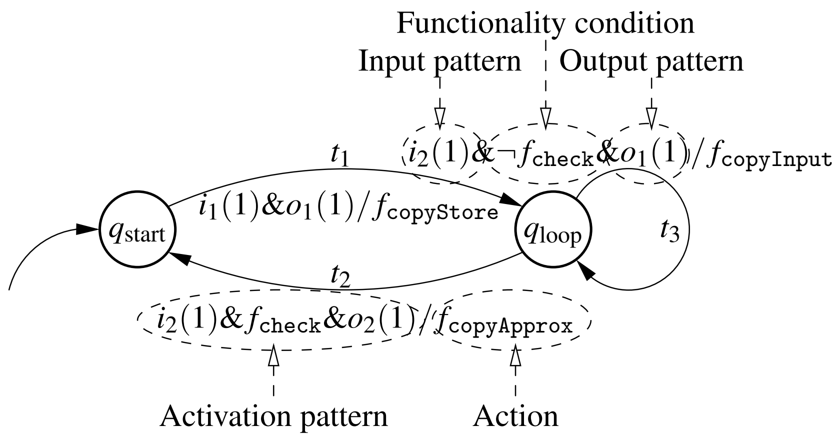
The notion of firing FSM is similar to the concepts introduced in SPI [ZER+99] and an extension of the *finite state machines* in FunState [STZ+01] by allowing requirements for a minimum of space available in output channels before a transition can be taken. The states of the firing FSM are called *firing 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. The activation patterns are partitioned into: (i) Predicates on the number of available tokens on the input ports called *input patterns*, e.g., $i_1(1)$ denotes a predicate that tests the availability of at least one token at the actor input port i_1, (ii) predicates on the number of free places on the output ports called *output patterns*, e.g., $o_1(1)$ checks if at least one free place is available at the output port o_1, and (iii) more general predicates called *functionality conditions* depending on the *functionality state* or the token values on the input ports. More formally, we derive the following two 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 $f_{\texttt{copyStore}}$ 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 $f_{\texttt{check}}$. 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.



Functionality condition
Input pattern
Output pattern
t_1
$i_2(1)\&\neg f_{\texttt{check}}\&o_1(1)/f_{\texttt{copyInput}}$
$i_1(1)\&o_1(1)/f_{\texttt{copyStore}}$
q_{start}
q_{loop}
t_3
t_2
$i_2(1)\&f_{\texttt{check}}\&o_2(1)/f_{\texttt{copyApprox}}$
Activation pattern
Action

Figure 2.3: Visual representation of the *firing FSM* of the `SqrLoop` actor a_2 shown in Figure 2.1. The `SqrLoop` actor controls the number of iterations performed by Newton's square root algorithm.

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 2.3, 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 '&&' FunCond '>>' OutputPattern |
                                           FunCond '>>' OutputPattern |
                         InputPattern              '>>' OutputPattern |
                         InputPattern '&&' FunCond                    |
                                                        OutputPattern |
                                           FunCond                    |
                         InputPattern                                 ;

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

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

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

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

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

InputPort            ::= <actor input port>

OutputPort           ::= <actor output port>

FunCond              ::= Expr ;
```

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 `&&`-operator[3] 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)` 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 activation patterns can have additional *functionality conditions* (`FunCond`) checking, e.g., token values on input ports or the functionality state of the actor. A functionality condition is simply an expression *Expr* which is constraint to have a boolean type. The syntax rules for legal compositions of these expressions is given below.

Syntax 2.2 Extended Backus-Naur form of expressions used in activation patterns:

```
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 ’)’ ;
```

[3]C++ logical AND

The 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 expression can be combined from the following terminals: (i) the `Constant`-terminal represents a C++ expression which evaluates at instantiation time of the 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`

2.2.4 Actor parameters

As mentioned in Section 2.1 actors can be parameterized using constructor parameters, e.g., as seen in Example 2.1 where the `Src` actor a_1, also displayed below, is parameterized `src("a1", 50)` to produce ascending values beginning from fifty.

Example 2.8 Definition of the `Src` actor class:

```
class Src: public smoc_actor {
public:
  smoc_port_out<double> out;
private:
  int i;

  void src() { out[0] = i++; }

  smoc_firing_state start;
public:
  Src(sc_module_name name, SMOC_ACTOR_CPARAM(int,from))
    : smoc_actor(name, start), i(from) {
    start =
        (VAR(i) <= 100)           >>
        out(1)                    >>
        CALL(Src::src)            >> start
      ;
  }
};
```

To enable the *SysteMoC* synthesis framework to extract the value of these constructor parameters they must be wrapped with the `SMOC_ACTOR_CPARAM(type, name)` macro. The extraction of these parameters via code parsing of the executable *SysteMoC* specification is in general not possible because these parameters may depend on data not included in the *SysteMoC* code, e.g., they may be derived from a configuration file. Therefore, it is necessary that these values are extracted at run time after the SystemC elaboration phase has finished. To facilitate this extraction the constructor parameters are wrapped by the `SMOC_ACTOR_CPARAM(type, name)` which enables the retention of the values of these parameters, till they can be inserted into an XML representation of the network graph which can be generated after the SystemC elaboration phase has finished. Note that the macro must not be applied to the `sc_module_name` parameter.

3. *SysteMoC* - XML formats

SysteMoC enables an XML representation of the network graph as defined in Section 2.1 to be automatically extracted, e.g., a skeleton of the XML representation of the network graph shown in Figure 2.1 can be seen in the following example.

Example 3.1 Skeleton of a network graph XML representation:

```
<?xml version="1.0"?>
<!DOCTYPE networkgraph SYSTEM "networkgraph.dtd">
<networkgraph name="sqrroot">
  <!-- All actors and channels of a network graph are given below.
       Where both actors and channels are represented via the XML
       process tag.
    -->
  <!-- Process tag describing SqrRoot actor a1 -->
  <process name="sqrroot.a1" type="actor" id="id1">
    <port name="sqrroot.a1.smoc_port_out_0" type="out" id="id2"/>
    <!-- Contains more tags describing the actor in more detail -->
    ...
  </process>
  ...
  <!-- Process tag describing FIFO channel c1 -->
  <process name="sqrroot.smoc_fifo_0" type="fifo" id="id24">
    <port name="sqrroot.smoc_fifo_0.in"  type="in"  id="id25"/>
    <port name="sqrroot.smoc_fifo_0.out" type="out" id="id26"/>
    <!-- Contains more tags describing the FIFO in more detail -->
    ...
  </process>
  ...
  <!-- Process tag describing SqrLoop actor a2 -->
  <process name="sqrroot.a2" type="actor" id="id4">
    <port name="sqrroot.a2.smoc_port_in_0"  type="in"  id="id5"/>
    <port name="sqrroot.a2.smoc_port_in_1"  type="in"  id="id6"/>
    <port name="sqrroot.a2.smoc_port_out_0" type="out" id="id7"/>
    <port name="sqrroot.a2.smoc_port_out_1" type="out" id="id8"/>
    <!-- Contains more tags describing the actor in more detail -->
    ...
  </process>
  ...
  <!-- Edge tags describing the FIFO connection from a1.o1 -> a2.i1 -->
  <edge name="sqrroot.smoc_fifo_0.to-edge"   source="id2"  target="id25"/>
  <edge name="sqrroot.smoc_fifo_0.from-edge" source="id26" target="id5"/>
  ...
</networkgraph>
```

The actors and channels of a network graph are represented as `process` tags contained in the `networkgraph` tag. The `process` tags are marked via the attribute `type` as representing an actor or FIFO channel. The edges of the network graph are represented as `edge` tags.

Example 3.2 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.

```
<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>
```

4. *SysteMoC* simulation environment

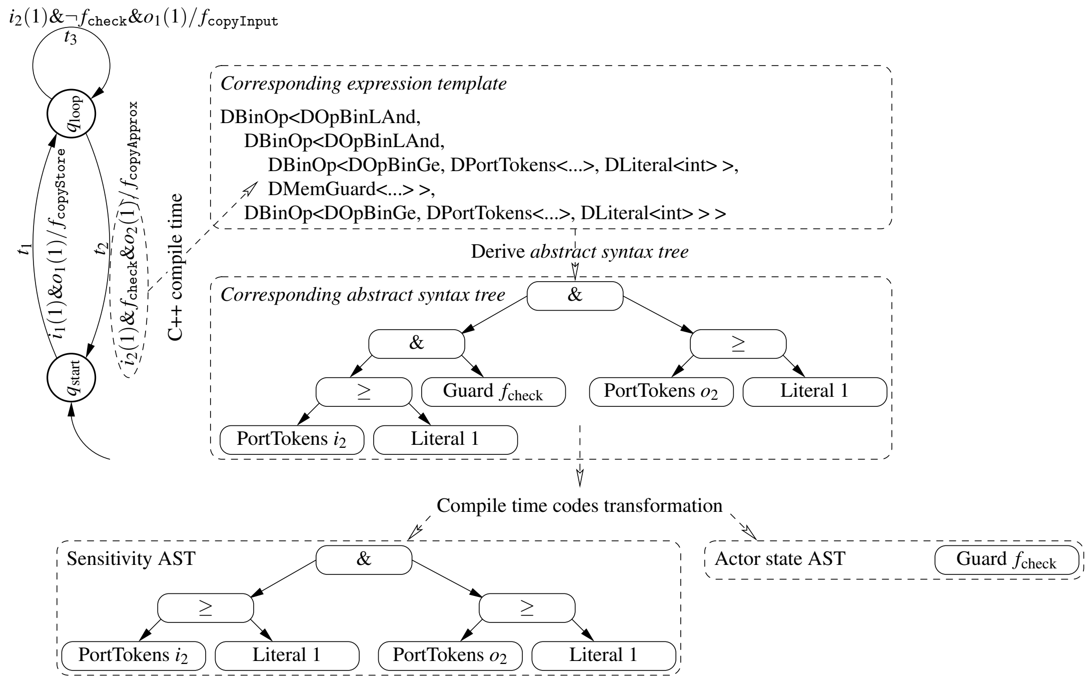
The execution of *SysteMoC* models can be divided into three phases: (i) checking for enabled transitions for each actor, (ii) selecting and executing one enabled transition per actor, and (iii) consuming and producing tokens needed by the transition. Note that step (iii) might enable new transitions. 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 4.1, or (iii) generation of an XML representation of the firing FSM, e.g., as seen in Figure 3.2, 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 4.1. 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



$i_2(1) \& \neg f_{\texttt{check}} \& o_1(1) / f_{\texttt{copyInput}}$
t_3
q_{loop}
t_1
$i_1(1) \& o_1(1) / f_{\texttt{copyStore}}$
t_2
$i_2(1) \& f_{\texttt{check}} \& o_2(1) / f_{\texttt{copyApprox}}$
q_{start}
C++ compile time
Corresponding expression template
Derive *abstract syntax tree*
Corresponding abstract syntax tree
&
&
$\geq$
$\geq$
Guard f_{check}
PortTokens o_2
Literal 1
PortTokens i_2
Literal 1
Compile time codes transformation
Sensitivity AST
&
$\geq$
$\geq$
PortTokens i_2
Literal 1
PortTokens o_2
Literal 1
Actor state AST
Guard f_{check}

```
DBinOp<DOpBinLAnd,
  DBinOp<DOpBinLAnd,
    DBinOp<DOpBinGe, DPortTokens<...>, DLiteral<int> >,
    DMemGuard<...> >,
  DBinOp<DOpBinGe, DPortTokens<...>, DLiteral<int> > >
```

Figure 4.1: 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.

state dependent AST parts, dedicated code is generated at C++ compile time for their evaluation.

A. Mathdefs

We use the '.'-operator, e.g., $u.x$, for member access, e.g., x, of tuples whose members have been explicitly named in their definition, e.g., $u = (x,y) \in U \subseteq \mathbf{N}^2$. Furthermore this member access operator has a trivial pointwise extension to sets of tuples, e.g., $U.x = \bigcup_{u \in U} u.x$.

Let $\{x_1, x_2\}$ denote a set and $[y_1, y_2, y_3, \ldots]$ denote a set with total order $y_1 \leq y_2 \leq y_3 \leq \ldots$, e.g., $\{x_1, x_2\} = \{x_2, x_1\}$ but $[y_1, y_2] \neq [y_2, y_1]$ and $\{x_1, x_1\} = \{x_1\}$ but $[y_1, y_2, y_1, y_3]$ is a *illformed totally ordered set* because it does not define a total order, as $y_1 \leq y_2 \wedge y_2 \leq y_1$ but $y_1 \neq y_2$.

Let $\mathbb{N} = [1,2,3,\ldots]$ denote the set of natural numbers, $\mathbb{N}_\infty = [1,2,3,\ldots\infty]$ the set of natural numbers including infinity, $\mathbb{N}_n = [1,2,3,\ldots,n] \subseteq \mathbb{N}_\infty$ the counting set from one to n (Note that $\mathbb{N}_0 = \emptyset$), $\mathbb{Z} = [\ldots,-2,-1,0,1,2,\ldots]$ denote the set of integers, $\mathbb{Z}_0^+ = [0,1,2,3,\ldots]$ the set of nonnegative integers, $\mathbb{Z}_0^{+\infty} = [0,1,2,3,\ldots\infty]$ the set of nonnegative integers including infinity.

A totally ordered set X is called *two-sided discrete* if it is *order isomorphic* to a subset of the integers $Z_X \subseteq \mathbb{Z}$, i.e., $\exists \Sigma(X) : Z_X \to X, \Sigma(X)$ is a *bijection* $: \forall i,j \in Z_X : i \leq j \iff \Sigma(X)(i) \leq \Sigma(X)(j)$. Intuitively this means, any two elements $x_1, x_2 \in X$ have only a finite number of other elements between them.

A totally ordered set X is called *one-sided discrete* or *discrete* if it is *order isomorphic* to a subset of the natural numbers $\mathbb{N}_{|X|}$ and let $\Sigma(X) : \mathbb{N}_{|X|} \to X$ denote this order isomorphism, i.e., $\exists \Sigma(X) : \mathbb{N}_{|X|} \to X, \Sigma(X)$ is a *bijection* $: \forall i,j \in \mathbb{N}_{|X|} : i \leq j \iff \Sigma(X)(i) \leq \Sigma(X)(j)$. Intuitively this means, any two elements $x_1, x_2 \in X$ have only a finite number of other elements between them and there exists a least element $x_{least}, \forall x \in X : x_{least} \leq x$. For example, given the discrete set $X = [$'A', 'B', 'C'$]$ then $\Sigma(X)(1) =$ 'A', $\Sigma(X)(2) =$ 'B', and $\Sigma(X)(3) =$ 'C'.

Let $X^* = \bigcup_{n \in \mathbb{Z}_0^+} X^n$ denote the set of all *tuples* of X also called *finite sequences* of X and $X^{**} = \bigcup_{n \in \mathbb{Z}_0^{+\infty}} X^n$ denote the set of all finite and infinite *sequences* of X. A sequence $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in X^n, n \in \mathbb{Z}_0^{+\infty}$ can also be thought of as a function $\mathbf{x} : \mathbb{N}_n \to X$. For example, given the sequence $X = ($'A', 'B', 'A', 'C'$)$ then $X(1) =$ 'A', $X(2) =$ 'B', $X(3) =$ 'A', and $X(4) =$ 'C'. Note that a function is also a relation and a relation is also a set, e.g $\mathbf{x} : \mathbb{N}_n \to X \equiv \{(i, x_i) \mid x_i = \mathbf{x}(i), i \in \mathbb{N}_n\}$. Note also that the Σ function converts a discrete set to a sequence, e.g., $\Sigma([1,2,3]) = (1,2,3)$.

A set with an associated partial order is called a *poset*. Let $\sqsubseteq$ called *prefix order* denote the associated partial order on the set V^{**} of all finite and infinite sequences of V. And let $\mathbf{v}_1 \sqsubseteq \mathbf{v}_2$ denote that $\mathbf{v}_1$ is a prefix of the sequence $\mathbf{v}_2$, i.e., $\mathbf{v}_1 \sqsubseteq \mathbf{v}_2 \iff$

$\forall n \in T(\mathbf{v}_1) : \mathbf{v}_1(n) = \mathbf{v}_2(n)$.

An *upper bound* of a subset $X \subseteq V^{**}$ is an element $\mathbf{v} \in V^{**}$ where every sequence $\mathbf{x}$ in X is a prefix of $\mathbf{v}$, i.e., $\forall \mathbf{x} \in X : \mathbf{x} \sqsubseteq \mathbf{v}$. A *least upper bound* of a subset $X \subseteq V^{**}$, written $\sqcup X$, is an upper bound which is a prefix of every other upper bound of X. A *lower bound* and *greatest lower bound* of $X \subseteq V^{**}$, written $\sqcap X$, are defined respectively.

A *chain* in a poset X is a totally ordered subset of X. A *bottom element* of a poset X, if it exists, is an element of the poset which is also a greatest lower bound of the whole poset $\sqcap X \in X$. A *complete partial order* is a poset X with a bottom element in which every chain $Y \subseteq X$ has a least upper bound $\sqcup Y$. The set V^{**} of all finite and infinite sequences of V is a complete partial order with bottom element λ, the empty sequence.

This definitions can be trivially generalized to tupples of sequences $\upsilon \in (V^{**})^N$, i.e., $\upsilon_1 \sqsubseteq \upsilon_2 \iff \forall n \in \mathbb{N}_N : \upsilon_1(n) \sqsubseteq \upsilon_2(n)$. With this generalization the set $(V^{**})^N$ of all N-tupples of finite and infinite sequences of V is also a complete partial order with bottom element $\Lambda = (\lambda, \lambda, \ldots, \lambda)$, the N-tupple of empty sequences λ.

A function $F : (V^{**})^N \to (V^{**})^M$ is *monotonic* if adding additional elements to the input sequence tupple $\upsilon_{in} \in (V^{**})^N$ results in additional elements on the output sequence tupple $\upsilon_{out} = F(\upsilon_{in}) \in (V^{**})^M$,i.e., $\forall \upsilon_{in_1}, \upsilon_{in_2} \in (V^{**})^N : \upsilon_{in_1} \sqsubseteq \upsilon_{in_2} \implies F(\upsilon_{in_1}) \sqsubseteq F(\upsilon_{in_2})$. This can be considered as a untimed notion of causality where the additional elements to the input sequence correspond to input tokens and the additional elements on the output sequence are the tokens produced from the consumption of these input tokens.

A function $F : (V^{**})^N \to (V^{**})^M$ is *continuous* if for every chain Y in $(V^{**})^N$, $F(Y)$ has a least upper bound $\sqcup F(Y)$ which is equal to the function F applied to the upper bound $\sqcup Y$ of the chain Y, i.e., $\forall Y \subseteq (V^{**})^N, Y$ is a chain : $F(\sqcup Y) = \sqcup F(Y)$. Where $F : 2^{((V^{**})^N)} \to 2^{((V^{**})^M)}$ is the pointwise extension of $F : (V^{**})^N \to (V^{**})^M$, i.e., $F(Y) = \{F(\upsilon) \mid \upsilon \in Y\}$. Note that every continuous function is also monotonic but the reverse is not necessarily true.

B. Glossary

English	German
network graph	Netzwerkgraph
actor	Aktor
firing FSM	Feuerungsautomat
activation pattern	Aktvierungsregel
input pattern	Eingangskanalbedingung
output pattern	Ausgangskanalbedingung
functionality condition	Funktionalitätsbedingung
guard function	Guardfunktion
actor functionality	Aktorfunktionalität

C. *SysteMoC* semantic

C.1 Actor behavior

As known from Kahn [Kah74] and Lee [LSV98, Lee97] we use the sequence of tokens $\mathbf{v} \in V^{**}$ transmitted during the lifetime of a system via a FIFO to mathematical describe the FIFO. In order to simplify the description of a bounded FIFO, we will replace it by two unbounded FIFOs. Therefore each actor port $p \in \mathcal{P} = I \cup O$ is connected to a pair of unbounded FIFOs $(\mathbf{v}, \mathbf{f}) \in Z$ instead of simply being connected to a bounded FIFO, where Z is the set of all pairs of unbounded FIFOs used to represent the bounded FIFO channels C of a network graph. The pair of unbounded FIFOs $(\mathbf{v}, \mathbf{f})$ consists of a *value FIFO* $\mathbf{v} \in V^{**}$ and a *free space FIFO* $\mathbf{f} \in \{\circ\}^{**}$, where each *free space symbol* $\circ$ represents an available slot in the value FIFO to store a token. Accordingly an actor not only consumes tokens $v \in V$ on its input ports I and produces them on its output ports O but also produces free space symbols $\circ$ on its input ports and consume them on its output ports, respectively. To reduce the mathematical clutter, we will not distinguish between *value sequences* and *free space symbol sequences* and simply use S to denote the set of all finite or infinite sequences of values or free space symbols, i.e., $S = V^{**} \cup \{\circ\}^{**}$.

Before we can continue to define the behavior of an actor, some mathematical notations for manipulating tuples are needed:

A tuple, e.g., $\mathbf{s} = (\mathbf{v}_{i_1}, \mathbf{v}_{i_2}, \mathbf{v}_{o_1})$, can be viewed as a function, e.g., $\mathbf{s} : \mathbb{N}_3 \to S$, over the counting set $\mathbb{N}_n = \{1, 2, \ldots, n\}$ from one to the tuple size n. Additionally we can associate which each tuple an ordered set, e.g., $\{i_1, i_2, o_1\} \subseteq \mathcal{P}$, and view the tuple as a function over this set, e.g., $\mathbf{s} : \{i_1, i_2, o_1\} \to S$. This allows us to access a tuple member either by its position inside the tuple, e.g., $\mathbf{s}(3) = \mathbf{v}_{o_1}$, or by an indexing element, e.g., the actor port o_1 and its associated tuple element $\mathbf{s}(o_1) = \mathbf{v}_{o_1}$. Moreover, we use the pointwise extension of the tuple member access operator to extract from a tuple its associated ordered sets, e.g., $\mathbf{s}.\mathbb{N} = \{1, 2, 3\}$ or $\mathbf{s}.\mathcal{P} = \{i_1, i_2, o_1\}$.

The position of a sequence in its tuple is equivalent to its name. To achieve the equivalent of renaming or hiding sequences, the position of a sequence in a tuple must be changed or the sequence must be dropped from the tuple. This is done by applying the *projection* function to a tuple.

Definition C.1 (Projection) *A* projection $\pi_I : X^n \to X^m$ *is a function which discards and reorders members $x \in X$ of a n-tuple according to an ordered set of indexes I to form a new m-tuple, where $|I| = m \leq n$. In other words, given $\mathbf{x} \in X^n$ and*

$I = \{i_1,\dots,i_m\}$ then $\pi_I(\mathbf{x}) = (\mathbf{x}(i_1),\mathbf{x}(i_2),\dots,\mathbf{x}(i_m))$. Furthermore a projection π_I can be generalized to sets of tuples $\mathbf{X} \subset X^n$, i.e., $\pi_I(\mathbf{X}) = \{\pi_I(\mathbf{x}) \mid \mathbf{x} \in \mathbf{X}\}$.

As an example consider the tuple of sequences $\mathbf{s} = (\mathbf{v}_{i_1},\mathbf{v}_{i_2},\mathbf{v}_{o_1})$, and the ordered set of actor ports $P = \{o_1, i_1\}$ then the projection is given as $\pi_P(\mathbf{s}) = (\mathbf{v}_{o_1},\mathbf{v}_{i_1})$.

With the mathematical notation defined above it is now possible to formally describe the behavior of an actor.

Definition C.2 (Actor behavior) *The behavior of an actor a can be thought of as a relation $\mathbf{P} \subseteq Z^{|\mathcal{P}|}$ which maps a tuple of input sequences $\mathbf{s}_{\text{in}} \in \pi_I(\mathbf{P}).\mathbf{v} \times \pi_O(\mathbf{P}).\mathbf{f}$ into possible multiple tuples of output sequences $\mathbf{s}_{\text{out}} \in \{\pi_I(\mathbf{p}).\mathbf{f} \times \pi_O(\mathbf{p}).\mathbf{v} \mid \mathbf{p} \in \mathbf{P} \wedge \pi_I(\mathbf{p}).\mathbf{v} \times \pi_O(\mathbf{p}).\mathbf{f} = \mathbf{s}_{\text{in}}\}$.*

Where the tuple of input sequences $\mathbf{s}_{\text{in}} \in \mathbf{S}_{\text{in}} \subset S^{|\mathcal{P}|}$ is constraint to be divided into value sequences $\pi_I(\mathbf{s}_{\text{in}}) \in (V^{**})^{|I|}$ and free space symbol sequences $\pi_O(\mathbf{s}_{\text{in}}) \in (\{\circ\}^{**})^{|O|}$. Furthermore the tuple of output sequences $\mathbf{s}_{\text{out}} \in \mathbf{S}_{\text{out}} \subset S^{|\mathcal{P}|}$ is similarly divided into value sequences $\pi_O(\mathbf{s}_{\text{out}}) \in (V^{**})^{|O|}$ and free space symbol sequences $\pi_I(\mathbf{s}_{\text{out}}) \in (\{\circ\}^{**})^{|I|}$.

The execution of an actor is divided into atomic *firing steps*. The possible firing steps an actor can take are dependent on the tuple $\mathbf{s}_{\text{in}}$ of still unconsumed input sequences and the actor state $q = (q_{\text{func}}, q_{\text{firing}}) \in Q = \mathcal{F}.Q_{\text{func}} \times \mathcal{R}.Q_{\text{firing}}$. The Fire-function deduces these possible steps and returns the set of possible produced output sequences $\mathbf{s}_{\text{prod}}$ and the resulting actor state q', i.e., Fire : $\mathbf{S}_{\text{in}} \times Q \to 2^{\mathbf{S}_{\text{out}} \times Q}$. [1] To decide if a transition can be taken, its associated *activation pattern* must be evaluated.

Definition C.3 (Activation pattern) *An* activation pattern *k of an actor $a \in A$ is a boolean function depending on the available input sequences $\mathbf{S}_{\text{in}}$ and the* functionality state *$\mathcal{F}.S_{\text{func}}$ of the actor, i.e., $k : \mathbf{S}_{\text{in}} \times \mathcal{F}.S_{\text{func}} \to \{\text{true}, \text{false}\}$. The activation pattern is used to decide if its associated transition can be taken* (true) *or not* (false).

To further express the behavior of a firing step, we need to define the prefix order $\sqsubseteq$ on the set of sequences S, i.e., given $\mathbf{u},\mathbf{v} \in S$ then $\mathbf{u} \sqsubseteq \mathbf{v} \equiv \mathbf{u}.\mathbb{N} \subseteq \mathbf{v}.\mathbb{N} \wedge \forall n \in \mathbf{u}.\mathbb{N} : \mathbf{u}(n) = \mathbf{v}(n)$. As an example consider the two sequences $\mathbf{u} = (v_1, v_2, v_3) \in S$ and $\mathbf{v} = (v_1, v_2, v_3, v_4, v_5) \in S$ then $\mathbf{u}$ is a prefix of $\mathbf{v}$, i.e., $\mathbf{u} \sqsubseteq \mathbf{v}$. This prefix order has a trivial pointwise extension to tuples of sequences, i.e., given $\mathbf{s}_1, \mathbf{s}_2 \in S^n$ then $\mathbf{s}_1 \sqsubseteq \mathbf{s}_2 \equiv \mathbf{s}_1.\mathbb{N} = \mathbf{s}_2.\mathbb{N} \wedge \forall n \in \mathbf{s}_1.\mathbb{N} : \mathbf{s}_1(n) \sqsubseteq \mathbf{s}_2(n)$. Additionally we will use the $\# : S \to \mathbb{N}_\infty$ operator to denote the length of a sequence, e.g., given $\mathbf{u} = (v_1, v_2, v_3)$ then $\#\mathbf{u} = |\mathbf{u}.\mathbb{N}| = 3$, and its pointwise extension to tuples of sequences $\# : S^n \to \mathbb{N}_\infty^n$, e.g., given $\mathbf{s} = (\mathbf{u}, \mathbf{v})$ then $\#\mathbf{s} = (3, 5)$.

With the notation given above we can now derive the following formal definition:

[1] We use 2^X to denote the power set of X, the set of all subsets of X, i.e., $2^X = \bigcup_{X_{\text{subset}} \subseteq X} \{X_{\text{subset}}\}$.

Definition C.4 (Firing step) *The execution of an actor is divided into atomic* firing steps.

$$
\begin{aligned}
\mathrm{Fire}(\mathbf{s}_{\mathrm{in}}, q) = \{ & & \quad (C.1)\\
& (\mathbf{s}_{\mathrm{prod}}, q') & \quad (C.2)\\
\mid \; & (q.q_{\mathrm{firing}}, k, f_{\mathrm{action}}, q'.q_{\mathrm{firing}}) \in \mathcal{R}.T & \quad (C.3)\\
\wedge \; & \mathbf{s}_{\mathrm{cons}} \sqsubseteq \mathbf{s}_{\mathrm{in}} & \quad (C.4)\\
\wedge \; & k(\mathbf{s}_{\mathrm{cons}}, q.q_{\mathrm{func}}) & \quad (C.5)\\
\wedge \; & \forall \mathbf{s}_{\mathrm{prefix}} \sqsubset \mathbf{s}_{\mathrm{cons}} : \neg k(\mathbf{s}_{\mathrm{prefix}}, q.q_{\mathrm{func}}) & \quad (C.6)\\
\wedge \; & (\pi_O(\mathbf{s}_{\mathrm{prod}}), q'.q_{\mathrm{func}}) = f_{\mathrm{action}}(\pi_I(\mathbf{s}_{\mathrm{cons}}), q.q_{\mathrm{func}}) & \quad (C.7)\\
\wedge \; & \#\pi_I(\mathbf{s}_{\mathrm{prod}}) = \#\pi_I(\mathbf{s}_{\mathrm{cons}}) \} & \quad (C.8)
\end{aligned}
$$

The possible firing steps an actor can take are dependent on the tuple of still unconsumed input sequences $\mathbf{s}_{\mathrm{in}} \in \mathbf{S}_{\mathrm{in}}$ *and the* actor state $q \in Q$ *(C.1) The* Fire-*function deduces these steps and returns a set containing* tuples of produced output sequences $\mathbf{s}_{\mathrm{prod}}$ *and the* resulting actor state q' *(C.2). The firing steps corresponds to the* execution of a transition $t \in \mathcal{R}.T$ *of the* firing FSM $\mathcal{R}$ *(C.3). Each firing step of an actor consumes a tuple of finite sequences* $\mathbf{s}_{\mathrm{cons}}$*, which must be a prefix of the tuple of input sequences (C.4). To decide if a transition can be taken its associated activation pattern must be evaluated (C.5). The tuple of input sequences consumed by a firing step is the shortest possible prefix tuple still satisfying the activation pattern (C.6). The associated action of the transition taken by the firing step is executed resulting in a new functionality state and a tuple of output sequences (C.7). Finally the number of free space symbols generated on the input ports equals the number of tokens consumed on them (C.8).*

Furthermore we can define the set of reachable output sequences obtained by multiple actor firings depending on an actor state and the available input sequence.

Definition C.5 (Reachable output sequences) *The set of reachable output sequences is a set containing* tuples of produced output sequences.

$$
\begin{aligned}
\mathrm{Reachable}(\mathbf{s}_{\mathrm{in}}, q) = \{ \quad & \mathbf{s}_{\mathrm{out}} & \quad (C.9)\\
\mid \; & (\mathbf{s}_{\mathrm{prod}}, q') \in \mathrm{Fire}(\mathbf{s}_{\mathrm{in}}, q) & \quad (C.10)\\
\wedge \; & \#\pi_I(\mathbf{s}_{\mathrm{cons}}) = \#\pi_I(\mathbf{s}_{\mathrm{prod}}) & \quad (C.11)\\
\wedge \; & \mathbf{s}_{\mathrm{in}} = \mathbf{s}_{\mathrm{cons}} {}^\frown \mathbf{s}'_{\mathrm{in}} & \quad (C.12)\\
\wedge \; & \mathbf{s}_{\mathrm{out}} = \mathbf{s}_{\mathrm{prod}} {}^\frown \mathbf{s}'_{\mathrm{out}} & \quad (C.13)\\
\wedge \; & \mathbf{s}'_{\mathrm{out}} \in \mathrm{Reachable}(\mathbf{s}'_{\mathrm{in}}, q') & \quad (C.14)
\end{aligned}
$$

This output sequences $\mathbf{s}_{\text{out}} \in \text{Reachable}(\mathbf{s}_{\text{in}}, q)$ *can be derived from an initial actor state q and the available input sequence* $\mathbf{s}_{\text{in}}$ *(C.9) by recursively applying firing steps (C.10). Whereas the consumed input sequence of a single firing step* $\mathbf{s}_{\text{cons}}$ *is constraint to be of the same length as the corresponding sequence of generated free space symbols (C.11) and furthermore must be a prefix* $\mathbf{s}_{\text{cons}} \sqsubseteq \mathbf{s}_{\text{in}}$ *of the available input sequence* $\mathbf{s}_{\text{in}}$ *leaving a tail* $\mathbf{s}'_{\text{in}}$ *(C.12) of the input sequence to be consumed by the successive firing steps (C.14). Furthermore each reachable output sequence has at its prefix* $\mathbf{s}_{\text{prod}}$ *(C.13) the produced output sequence from a single firing step and at its tail a output sequence produced by successive firing steps (C.14).*

With the above definitions we can now describe the behavior of an actor derived from its actor functionality and its firing FSM to be defined as follows:

Definition C.6 (Derived actor behavior) *The behavior of an actor a can be thought of as a relation* $\mathbf{P} \subseteq Z^{|\mathcal{P}|}$ *(C.15) which can be derived from its actor functionality* $a.\mathcal{F}$ *and its firing FSM* $a.\mathcal{R}$ *as follows:*

$$\begin{aligned}
\mathbf{P} = \{ \quad & \mathbf{p} && \text{(C.15)}\\
| \quad & \pi_I(\mathbf{p}).\mathbf{v} \times \pi_O(\mathbf{p}).\mathbf{f} = \mathbf{s}_{\text{in}} && \text{(C.16)}\\
\wedge \quad & \pi_I(\mathbf{p}).\mathbf{f} \times \pi_O(\mathbf{p}).\mathbf{v} = \mathbf{s}_{\text{out}} && \text{(C.17)}\\
\wedge \quad & \mathbf{s}_{\text{in}} \in \mathbf{S}_{\text{in}} && \text{(C.18)}\\
\wedge \quad & \mathbf{s}_{\text{out}} \in \text{Reachable}(\mathbf{s}_{\text{in}}, (\mathcal{F}.q_{0\text{func}}, \mathcal{R}.q_{0\text{firing}}))\} && \text{(C.19)}
\end{aligned}$$

This relation maps a tuple of input sequences $\mathbf{s}_{\text{in}}$ *(C.16) into possible multiple tuples of output sequences* $\mathbf{s}_{\text{out}}$ *(C.17). Whereas the tuple of output sequences* $\mathbf{s}_{\text{out}}$ *must be reachable (C.19) from the initial actor state and some tuple of input sequences* $\mathbf{s}_{\text{in}}$ *(C.18).*

C.2 Network graph behavior

Bibliography

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

[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.

[FHT06] Joachim Falk, Christian Haubelt, and Jürgen Teich. Efficient representation and simulation of model-based designs in systemc. Proc. FDL'06, Forum on Design Languages 2006, Darmstadt, Germany, September 2006.

[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.

[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.