SystemCoDesigner
Functional Modeling Reference

SysteMoC Version 2.8 (Non-Hierarchical)

Joachim Falk
Christian Zebelein
Martin Streubühr
Jens Gladigau
Christian Haubelt
Jürgen Teich

Department of Computer Science 12
Hardware-Software-Co-Design
University of Erlangen-Nuremberg, Germany
http://www12.cs.fau.de/research/scd

SystemCoDesigner Functional Modeling Reference – SysteMoC Version 2.8 (Non-Hierarchical)

by J. Falk, Ch. Zebelein, M. Streubühr, J. Gladigau, Ch. Haubelt, and J. Teich

Department of Computer Science 12
Hardware-Software-Co-Design
University of Erlangen-Nuremberg, Germany
http://www12.cs.fau.de/research/scd

Contents

1. Introduction

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.

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. In this report, the *SysteMoC* library [FHT06] is presented. *SysteMoC* is based on the design language SystemC and provides a simulation environment for model-based designs. The library-based approach unites the advantage of executability with analyzability of many expressive MoCs.

Using *SysteMoC*, many important MoCs can be expressed such as Synchronous Dataflow [LM87], Cyclo-Static Dataflow [BELP96, EBLP94], Boolean Dataflow, Kahn Process Networks [Kah74], communicating sequential processes [Hoa85], and many others [TSZ$^+$99, Lee97, Lee02, LSV98, EJL$^+$02, GB04].

In order to express different MoCs, the *SysteMoC* library provides different communication channels, e.g., queues with FIFO semantics. Additionally, *SysteMoC* actors are composed of three basic elements, see Figure 1.1:

- *Variables*: Variables are used to store data values locally to an actor.
- *Functions*: Functions describe the transformative part of an actor, i.e., transforming data values. Functions can be either *action functions* or *guard functions*. Guard functions always evaluate either to true or to false and must not change any variable values.
- *Finite State Machine (FSM)*: The behavior of each actor is ruled by an explicit Finite State Machine, called *actor FSM*. State transitions are guarded by conditions checking for, e.g., available input data, input values, and internal values of

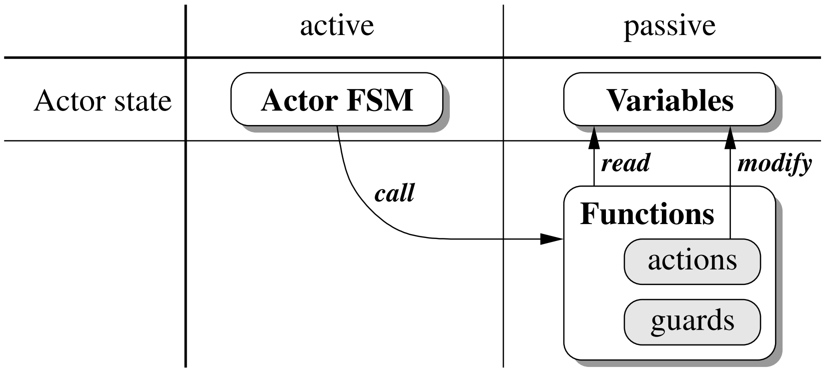

Figure 1.1.: A *SysteMoC* actor is composed of a Finite State Machine (FSM), functions, and variables. The FSM controls the function invocation. Functions are executed atomically and may change variables at the end of their execution.

variables of the actor. These conditions also encode the number of consumed input token and produced output token from input and output channels, respectively. If a state transition is enabled (i.e., the conditions evaluate to true), an associated action function can be performed and, successively, input token are consumed, output token are produced, the the next state is set.

The consumption and production of tokens is locally triggered by transitions of an explicit *actor FSM* required in each actor. The purpose and advantages of this clear separation of data transformation and communication behavior in model-based designs written in *SysteMoC* are:

- *recognizability*: Important Model of Computation can be recognized by analyzing the actor FSM and the channel type [ZFHT08].
- *analyzability*: As a consequence of being able to detect important well-known MoCs within *SysteMoC* models, many important and well-known analysis algorithms such as checking for 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 [FKH$^+$08].
- *simulatability*: Finally, even most complex *SysteMoC* models for which no formal analysis techniques are known can be handled by simply simulating the model. As *SysteMoC* is built on top of SystemC, an event-driven simulation of the exact timing and concurrency among actors is immediately possible [SFH$^+$06].

- *refinement*: Important refinement transformations can be applied to a *SysteMoC* model resulting in a hardware/software target implementation, including automatic platform-based code synthesis. This is part of the *SystemCoDesigner* design methodology [HFK$^+$07, HMSK08] that is based on the *SysteMoC* functional modeling approach.

2. Syntax

This section presents the syntax defined by the *SysteMoC* library for model-based designs. A complete application is modeled *SysteMoC* by a set of *actors* and their interconnection using *channels*. The overall model is therefore a network of actors and channels. 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.

2.1. Network Graph

The creation of a *SysteMoC* model 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 **??**. The network graph is composed of *actor instances* of these actor classes which are connected via *channels*. Before defining a network graph formally, an example is given.

Example 2.1 The approximative square root algorithm in Figure 2.1 is stimulated by an infinite sequence of input token values generated by the actor `Src`. These input token values are transported via channel c_1 to the actor `SqrLoop` 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 `Approx` via channel c_2. This will eventually result in a new better approximated square root value in channel c_4. This iteration repeats until the error bound is satisfied and the approximation result is forwarded via channel c_5 to the actor `Sink`.

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:

```
// Declare network graph class SqrRoot
class SqrRoot: public smoc_graph {
protected:
  // Actors are C++ objects
```

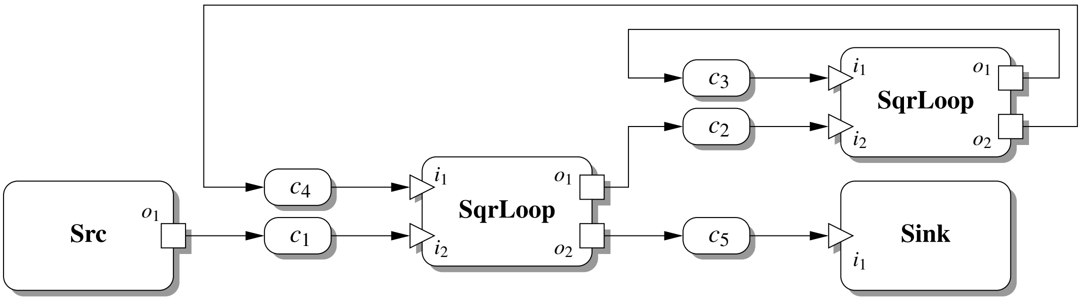

Figure 2.1.: The *network graph* displayed above implements Newton's iterative algorithm for calculating the square roots of an infinite input sequence generated by the actor `Src`. The square root values are generated by Newton's iterative algorithm actor `SqrLoop` for the error bound checking and actor `Approx` to perform an approximation step. After satisfying the error bound, the result is transported to the actor `Sink`.

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

The actors of the network graph 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("src", 50)`. The connections of these actors via channels are assembled in the constructor of the network graph class, e.g., `connectNodePorts (src.o1, sqrloop.i1, smoc_fifo<double>())`. Here, queues with FIFO semantics are used for the communication.

The queues are created by the `connectNodePorts(`o`, `i`[, param])` function which creates a queue with FIFO semantics 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* network graph *is a graph* $g = (A, C, E)$*, where*

- A *is a finite set of* actors
- C *is a finite set of* channels
- E *is a finite set of* edges*, with* $E \subseteq (C \times A.I) \cup (A.O \times C)$[1]*.*

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 $A.\mathscr{P} = A.I \cup A.O$.[2] Actors are discussed in more detail in Section 2.2. A presentation of different channels supported by *SysteMoC* is given in Section 2.3. Finally, the execution semantics are discussed in Chapter 3.

2.2. Actor

Definition 2.2 (Actor) *An* actor *is a tuple* $a = (P, S, F, R, m_{\text{init}})$*, where*

- P *is a finite set of* ports*, partitioned into* input ports I *and* output ports O *(i.e.,* $P = I \cup O$*, with* $I \cap O = \emptyset$*)*
- S *is a finite set of* variables

[1] $A.I = \bigcup_{a \in A} a.I$ and $A.O = \bigcup_{a \in A} a.O$ denote the sets of all input ports, respectively all output ports, of all actors in the network graph.

[2] We use the '.'-operator, e.g., $a.\mathscr{P}$, for member access, e.g., $\mathscr{P}$, of tuples whose members have been explicitly named in their definition, e.g., $a \in A$ from Definition 2.2. Moreover, this member access operator has a trivial pointwise extension to sets of tuples, e.g., $A.\mathscr{P} = \bigcup_{a \in A} a.\mathscr{P}$, which is also used throughout this document.

- *F is a finite set of* functions*, partitioned into* action functions F_{actions} *and* guard functions F_{guards}
- *R is the* actor's FSM
- *m_{init} is an* initial assignment *to the variables in S, i.e., $m_{\text{init}} : S \to \mathbb{D}$.*

TODO

Describe following points within the running example (not within own subsection):

- Actor parameter
- Function parameter

2.2.1. Ports

2.2.2. Variables

2.2.3. Functions

2.2.4. Actor finite state machine

Definition 2.3 (Actor FSM) *Let $g = (A, C, E)$ be a network graph. The* Actor FSM *$a.R = (T, Q, q_0)$ of an actor $a = (P, S, F, R, m_{\text{init}}) \in A$ is a tuple, where*

- *T is a finite set of* transitions
- *Q is a finite set of* states
- *q_0 is an* initial state*, with $q_0 \in Q$.*

Definition 2.4 (Transition) *Let $a = (P, S, F, R, m_{\text{init}})$ be an actor, with actor FSM $R = (T, Q, q_0)$. A* transition *$t = (q, k, f_{\text{action}}, q') \in T$ is a tuple, where*

- *q is the current state, with $q \in Q$*
- *k is a boolean function, called* activation pattern
- *f_{action} is an action function, with $f_{\text{action}} \in F_{\text{actions}}$*
- *q' is the next state, with $q' \in Q$.*

2.3. Channel

3. Semantics

A. Backus-Naur Form

B. Glossary

English	German
action (function)	Aktion
activation pattern	Aktvierungsregel
actor	Aktor
actor FSM	Aktor FSM
actor functionality	Aktorfunktionalität
functionality condition	Funktionalitatsbedingung
functionality state	Funktionalitatszustand
guard function	Wächterfunktion
input pattern	Eingangskanalbedingung
network graph	Netzwerkgraph
output pattern	Ausgangskanalbedingung

Bibliography

[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. In *Proc. FDL'06, Forum on Design Languages 2006*, Darmstadt, Germany, September 2006.

[FKH+08] Joachim Falk, Joachim Keinert, Christian Haubelt, Jürgen Teich, and Shuvra Bhattacharyya. A Generalized Static Data Flow Clustering Algorithm for MPSoC Scheduling of Multimedia Applications. In *Proceedings of the International Conference on Embedded software (EMSOFT'2008)*, pages 189–198, Atlanta GA, USA, October 2008.

[GB04] Marc Geilen and Twan Basten. Reactive process networks. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 137–146, 2004.

[GHJV95] R. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[HFK+07] Christian Haubelt, Joachim Falk, Joachim Keinert, Thomas Schlichter, Martin Streubühr, Andreas Deyhle, Andreas Hadert, and Jürgen Teich. A SystemC-based Design Methodology for Digital Signal Processing Systems. *EURASIP Journal on Embedded Systems (JES)*, 2007:Article ID 47580, 22 pages, 2007.

[HMSK08] Christian Haubelt, Michael Meredith, Thomas Schlichter, and Joachim Keinert. SystemCoDesigner: Automatic Design Space Exploration and Rapid Prototyping from Behavioral Models. In *Proceedings of the 45th Design Automation Conference (DAC'08)*, pages 580–585, Anaheim, CA, USA., 2008.

[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.

[SFH+06] Martin Streubühr, Joachim Falk, Christian Haubelt, Jürgen Teich, Rainer Dorsch, and Thomas Schlipf. Task-Accurate Performance Modeling in SystemC for Real-Time Multi-Processor Architectures. In *Proceedings of Design, Automation and Test in Europe*, pages 480–481, Munich, Germany, 2006. IEEE Computer Society.

[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.

[ZFHT08] Christian Zebelein, Joachim Falk, Christian Haubelt, and Jürgen Teich. Classification of General Data Flow Actors into Known Models of Computation. In *Proc. 6th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2008)*, pages 119–128, Anaheim, CA, USA, June 2008.