

Representing Models of Computation in SystemC

Joachim Falk, Christian Haubelt, Jürgen Teich
Universität Erlangen-Nürnberg, Hardware-Software-Co-Design
Erlangen, Germany
{falk, haubelt, teich}@cs.fau.de

*Due to rising design complexity it is necessary to increase the level of abstraction at which systems are designed. On the other hand, modern embedded system design is still based on design languages like C, C++, Java, VHDL, SystemC, and SystemVerilog which allow unstructured communication. The specification is mostly mapped onto a set of interacting tasks and hardware modules, which interact by the use of shared variables and various ways of message passing. Even worse, nearly all design languages are Turing complete making analysis in general impossible. However, by constraining the type of communication used between tasks and the communication behavior of the tasks, expressiveness can be traded for analyzability. In this paper we, propose a library called *SystemMoC* based on the design language SystemC. This library permits the construction of well known models of computation which are the basis for model-based system design.*

1 Introduction

Models of computation [Lee02], in the following called *MoCs*, are predefined types of communication and strategies for scheduling communicating tasks. Thus, MoCs are comparable to design patterns known from the area of software design [GHJV95]. Limiting the expressiveness of a MoC permits code generators to produce optimized code and automatic and efficient verification to check system properties. Examples of MoCs are *Finite State Machines*, *Dataflow models*, *Communicating Sequential Processes*, etc.

On the other hand, industrial embedded system design is still based on design languages which allow unstructured communication. Even worse, nearly all design languages are Turing complete making analysis in general impossible. To make industry benefit from the best of both worlds, engineers must restrict themselves to use certain subsets of a design language. This results in a model-based design methodology that permits automatic analysis, identification, and and extraction of MoCs at the source code level.

In this paper, we will propose a library, called *SystemMoC*, that provides classes allowing the easy expression of MoCs in SystemC. The overall methodology is depicted in Figure 1.

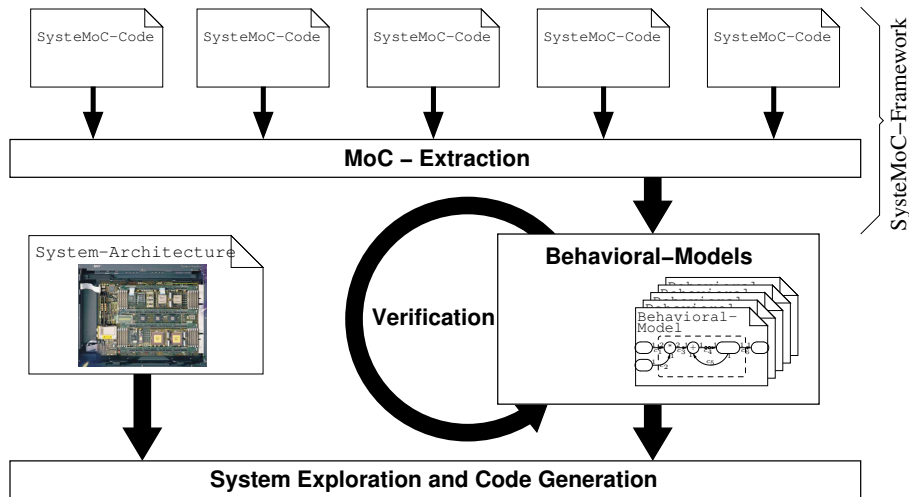


Figure 1: The Goal of the *SystemMoC*-framework is the identification and extraction of models of computation from a subset of SystemC. Later the extracted MoC can be used in a model-based design methodology, e.g, in design space exploration and for code generation.

In this paper, we focus on the *SystemMoC*-framework and MoC-extraction. The rest of this paper is structured as follows: In Section 2, we discuss related work. In Section 3, we present the mathematical background for representing MoCs in SystemC. In Section 4, a SystemC implementation of our framework called *SystemMoC* is presented, and we conclude the present paper in Section 5.

2 Related Work

SystemC [GLMS02] already allows to easily implement dataflow MoCs via communicating actors and `sc_fifo` channels. However, a dataflow MoC implemented in this way is unstructured and no possibilities exist for deriving information about its communication behavior. The facilities for implementing MoCs in SystemC have been extended by Herrera et al. [HSV04] who have implemented a custom library of channel types like rendezvous on top of the SystemC discrete event simulation kernel. But no constraints have been imposed how these new channels types are used by an actor. Consequently, no information about the communication behavior of an actor can be derived. Implementing these channels on top of the SystemC discrete event simulation kernel curtails the performance of such an implementation. Another approach has been taken by Patel et al. [HDPS04] which have extended SystemC itself with different simulation kernels for *Communicating Sequential Processes* and *Finite State Machine MoCs* to improve simulation efficiency. Moreover, they have implemented hierarchical composition of MoCs following the approach of Ptolemy II. Ptolemy II [Lee04] is a simulation framework for MoCs implemented in Java. Its aim is the exploration of different MoCs and the semantic of hierarchical composition of these MoC with each other. Other work on MoCs can be found in [ZER⁺99] and [STZ⁺01]. In contrast to the approaches discussed above, our methodology restricts the communication behavior in such a way that the identification, extraction, and analysis of different MoCs is permitted.

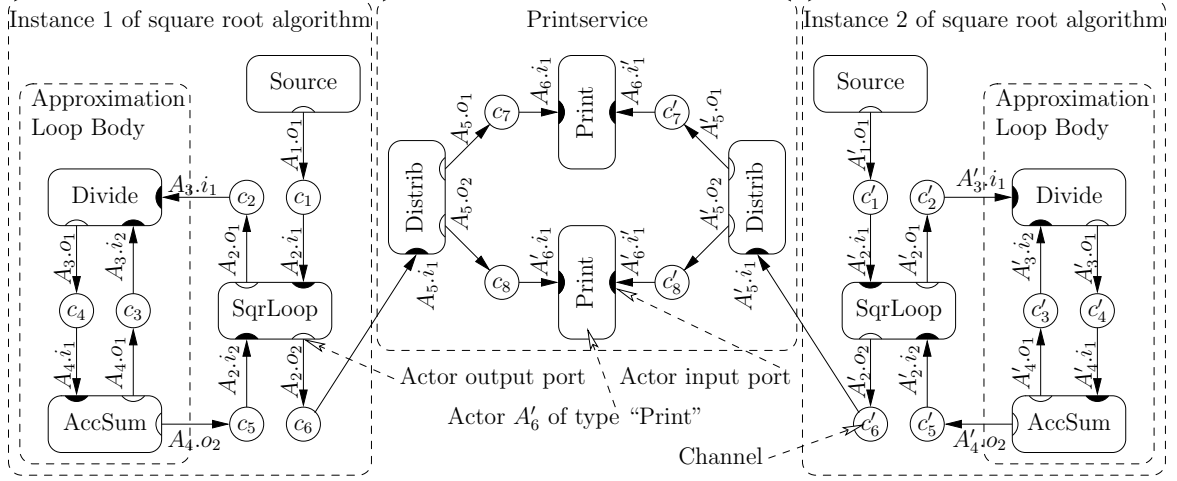


Figure 2: Example of a *network graph* which belongs to a model of two instances of an approximation square root algorithm.

3 Actor-Oriented Design

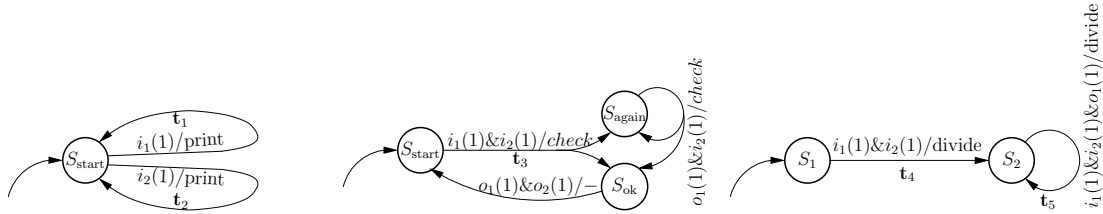
Instead of a monolithic approach for representing an executable specification of an embedded system as done using many design languages, we will use a refinement of *actor-oriented* design. In actor-oriented design, *actors* only communicate with each other via *channels* instead of method calls as known in object-oriented design.

In an actor-oriented design [Agh97], a model of computation defines [Lee02] the interaction policy between actors. Actors are objects which execute concurrently. An actor A can only communicate with the environment (other actors) through its *actor input and output ports* $A.I$ and $A.O$ respectively. The actor ports are connected with each other via a communication infrastructure. This infrastructure is separated into *network graph* and *channel kind*.

Definition 3.1 A *network graph* is a directed bipartite graph $G_N = (\mathcal{A}, \mathcal{C}, E)$ containing a set of actors \mathcal{A} , a set of channels \mathcal{C} , and a set of directed edges $E \subseteq (\mathcal{C} \times \mathcal{P}_I) \cup (\mathcal{P}_O \times \mathcal{C})$, where $\mathcal{P}_I = \bigcup_{A \in \mathcal{A}} A.I$ and $\mathcal{P}_O = \bigcup_{A \in \mathcal{A}} A.O$ are the sets of actor input and output ports of all actors in the network graph respectively. Each actor $A \in \mathcal{A}$ can 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 $\mathcal{P} = \mathcal{P}_I \cup \mathcal{P}_O$.

The *channel kind* defines the communication semantics of a channel. Since the communication semantics are not influenced by the data type communicated, the actual *type* of a communication channel is derived from the channel kind by parameterizing it with the actual data type. Examples for channel kinds are FIFO channels and rendezvous channels. Channels are used to transport data values in form of so-called *tokens*.

The network graph shown in Figure 2 consists of two instances of an approximation square root algorithm. Note that the network graph is symmetric as the actors $A_1 - A_6$ and $A'_1 - A'_6$ appear in an identical configuration. Each of these subgraphs implements one instance of the approximation algorithm. The input values for this algorithm are created in the source actor A_1 . The input values are transported via channel c_1 to the



(a) Firing rules of the Print actor A_6 (A'_6) in Figure 2 constructed with the *choice node interface*.

(b) Firing rules of the SqrLoop actor A_2 (A'_2) in Figure 2 constructed with the *transact node interface*.

(c) Firing rules of the Divide actor A_3 (A'_3) in Figure 2 constructed with the *fixed transact node interface*.

Figure 3: Example of firing rules of some actors in Figure 2.

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. When this happens, the approximated square root value will be sent via channel c_6 to actor A_5 . This actor will forward the result to one of the two Print actors A_6 or A'_6 whichever is ready first.

In actor-oriented design the node functionality and the firing rules are combined into one opaque block, which can in general not be analyzed. The information in this block would help to reason about communication patterns, to produce optimized code and to check system properties. In our methodology, the actor concept is still used, but we distinguish two different aspects of an actor called *node functionality* and *firing rules*.

Definition 3.2 *An actor is a tuple $A = (\mathcal{F}, \mathcal{R}, I, O)$ containing a node functionality \mathcal{F} , firing rules \mathcal{R} , a set of actor input ports I , and a set of actor output ports O .*

The *node functionality* ($A.\mathcal{F}$) is responsible for transforming the data values which reside inside tokens. The node functionality will still be regarded as an opaque block. Thus, we do not try to analyze it.

Definition 3.3 *The node functionality of an actor A is a tuple $A.\mathcal{F} = (F, Q, q_1)$ containing a set of functions F , a set of functionality states Q (possibly infinite), and an initial functionality state q_1 .*

The functions $f \in F$ of the node functionality map a fixed number of input values $(v_{i1}, v_{i2}, \dots, v_{iN}) = \mathbf{v}_i \in V^N$ and a functionality state $q_i \in Q$ to a fixed number of output values $(v_{o1}, v_{o2}, \dots, v_{oM}) = \mathbf{v}_o \in V^M$ and a new functionality state $q_{i+1} \in Q$, i.e., $f : V^N \times Q \rightarrow V^M \times Q$. The input and output values are provided by the firing rules, as discussed next. For example, the node functionality of the Divide actor A_3 (A'_3) in Figure 2 is described by $\mathcal{F} = (F, Q, q_1)$, $F = \{\text{divide}\}$, $Q = \{\perp\}$, $q_1 = \perp$, and $\text{divide}(\mathbf{v}_i, q) = ((v_{i1}/v_{i2}), q)$.

The second part of an actor A which determines its communication behavior is called *firing rules* ($A.\mathcal{R}$). The firing rules determine for each actor port the number of tokens to be receive or to be sent until the associated node functionality can be invoked.

Definition 3.4 *The firing rules of an actor A are a 4-tuple $A.\mathcal{R} = (\mathcal{T}, Q, q_1, \delta)$ containing a set of firing transitions \mathcal{T} , a set of firing states Q , an initial firing state q_1 ,*

and a firing transition function $\delta : Q \rightarrow 2^{\mathcal{T}}$ which maps each state $q \in Q$ to the set of transitions $\mathcal{T}_q = \delta(q)$.

An actor is *blocked* in state q until at least one firing transition $\mathbf{t} \in \mathcal{T}_q$ is enabled, as discussed below. The set of enabled transitions is given by $\mathcal{T}'_q = \{\mathbf{t} \mid \mathbf{t} \in \mathcal{T}_q \wedge \mathbf{t} \text{ is enabled}\}$. Exactly one transition $t \in \mathcal{T}'_q$ will be chosen nondeterministically for execution. For example, if both transitions $\{\mathbf{t}_1, \mathbf{t}_2\} = \delta(S_{start})$ in the firing rules shown in Figure 3(a) of the Print Actor A_6 from Figure 2 are enabled, one of them is executed nondeterministically.

Definition 3.5 A firing transition is a tuple $\mathbf{t} = (k, J)$ containing an activation pattern k and an interface action J .

A transition $\mathbf{t} = (k, J)$ is enabled if its *activation pattern* k is enabled. If a transition \mathbf{t} is enabled and part of the transitions \mathcal{T}_q which leave the current state q the transition \mathbf{t} can be executed, i.e., $(k, J) = \mathbf{t} \in \mathcal{T}_q \wedge k$ is enabled $\implies \mathbf{t}$ can be executed. If a transition $\mathbf{t} = (k, J)$ is executed, the associated activation pattern k and the *interface action* J are executed consecutively. The execution of the activation pattern is responsible for the actor communication via the actor ports. The execution of the interface action leads to function calls in the node functionality.

Definition 3.6 An activation pattern is a function $k : A.I \cup A.O \rightarrow \mathbb{Z}_0^+$, $k \in \mathcal{K}$ which maps each actor port of the actor A to a nonnegative integer.

An activation pattern is a predicate on actor ports. It encodes the number of tokens to be sent (via actor output ports) or to be received (via actor input ports) when the activation pattern is executed. For example, the activation pattern k of the transition \mathbf{t}_4 in Figure 3(c) is expressed as $k(i_1) = k(i_2) = 1$ and $k(o_1) = 0$. However, the specification of an activation pattern by its function poses a great overhead. Therefore, we will use the shorthand $k = i_1(1) \& i_2(1)$ to express the same information.

An activation pattern k is enabled if for each actor input port $p_I \in A.I$ at least $k(p_I)$ tokens can be received and for each actor output port $p_O \in A.O$ at least $k(p_O)$ tokens can be sent. If an activation pattern k is executed it receives $k(p_I)$ tokens via the actor input port p_I and stores them in temporary buffers in these actor input ports. For actor output ports p_O , $k(p_O)$ tokens in temporary buffers of the actor output ports are sent via these ports. These temporary buffers are used to provide the input and output values for the node functionality.

Definition 3.7 An interface action is a tuple $J = (f, Q_{succ})$, $J \in \mathcal{J}$ containing a function $f \in A.\mathcal{F}.F$ of the node functionality $A.\mathcal{F}$ and a set of possible successor states Q_{succ} . Where the set of possible successor states must contain at least one element, i.e., $|Q_{succ}| \geq 1$.

A transition $\mathbf{t} = (k, J)$ is called a *deterministic transition* if the associated interface action J has only one successor state, i.e., $|J.Q_{succ}| = 1$. A transition $\mathbf{t} = (k, J)$ is called a *conflict transition* if the associated interface action J has more than one possible successor state, i.e., $|J.Q_{succ}| > 1$. For example the transition $\mathbf{t}_4 = (k, J)$ in Figure 3(c) with $k = i_1(1) \& i_2(1)$, $J.f = \text{divide}$, and $J.Q_{succ} = \{S_2\}$ is deterministic. But the transition $\mathbf{t}_3 = (k, J)$ from firing state S_{start} to S_{ok} or S_{again} in Figure 3(b) with $k = i_1(1) \& i_2(1)$, $J.f = \text{check}$, and $J.Q_{succ} = \{S_{ok}, S_{again}\}$ is conflicting.

The notion of firing rules is similar to the concepts introduced in SPI [ZER⁺99] and FunState [STZ⁺01]. If a transition $\mathbf{t} = (k, J)$ is executed, the associated activation pattern k and the function $J.f$ of the node functionality are executed consecutively. In case of a deterministic transition, the current firing state is updated to the only successor state $q_{\text{succ}} \in J.Q_{\text{succ}}$. In case of a conflict transition, the successor state actually used is a runtime decision of the function $J.f$.

As the firing rules determine the communication behavior of an actor, we can use the firing rules to distinguish five different operations: (i) *Branch* is an operation which maps a function f of the node functionality and a none empty set of firing state $Q_B \neq \emptyset$ to an interface action J , i.e., $\text{Branch} : A.\mathcal{F}.F \times 2^{A.\mathcal{R}.Q} \rightarrow \mathcal{J}$. If the resulting interface action J is included into a transition \mathbf{t} it permits to create a conflict transition. (ii) *Call* is an operation which maps a node functionality f of the node functionality and a firing state q to an interface action J , i.e., $\text{Call} : A.\mathcal{F}.F \times A.\mathcal{R}.Q \rightarrow \mathcal{J}$. If the resulting interface action J is included into a transition \mathbf{t} , it creates a deterministic transition. (iii) *Transition* is an operation which maps an activation pattern k and an interface action J to an interface transition \mathbf{t} , i.e., $\text{Transition} : \mathcal{K} \times \mathcal{J} \rightarrow \mathcal{T}$. (iv) *Choice* is an operation which maps a set of interface transitions \mathcal{T} to a firing state q , i.e., $\text{Choice} : 2^{\mathcal{T}} \rightarrow Q$. The firing state created by this operation can have multiple outgoing transitions. (v) *Transact* is an operation which maps an interface transition T to a firing state s , i.e., $\text{Transact} : \mathcal{T} \rightarrow Q$. The firing state created by this operation has only one outgoing transition \mathbf{t} .

Together with the network graph and the channel kinds used, these five operations permit the identification of the underlying MoC. As a MoC is the basis for analysis and efficient code generation, it is preferable that a design language provides appropriate language elements to express these aspects. Next, we will show how this can be done in the system design language SystemC. For this purpose, we distinguish three separate *node interface types* by the set of operations available to them: (i) Choice node interface provides the operations *Transition*, *Choice*, *Transact*, *Call* and *Branch* (ii) Transact node interface provides the operations *Transition*, *Transact*, *Call* and *Branch* (iii) Fixed transact node interface provides only the operations *Transition* and *Call*

4 Software Architecture

Due to its high abstraction level and its capabilities in both, hardware and software refinement, we have chosen SystemC [Bai03, GLMS02] as our platform for system design. SystemC is an actor-oriented C++-based design language. In the following, we propose a library, called *SystemMoC*, which augments SystemC with additional services as can be seen in Figure 4 to implement the concepts presented in the previous section. For this purpose, *SystemMoC* provides base classes which define the operations to construct firing rules. This set of operations restricts an actors communication behavior. Moreover, channel kinds like FIFO and rendezvous, and a base class for the network graph are provided.

The problem of extracting the communication behavior of an actor is caused by the fact, that the communication methods in SystemC are all accessible to all SystemC modules. Moreover, execution of these methods is controlled by the Turing complete coding possibilities in the actor. To remedy these problems, we split the actor into a *node functionality* and a *firing rules* part, as discussed in Section 3. Instead of discussing the

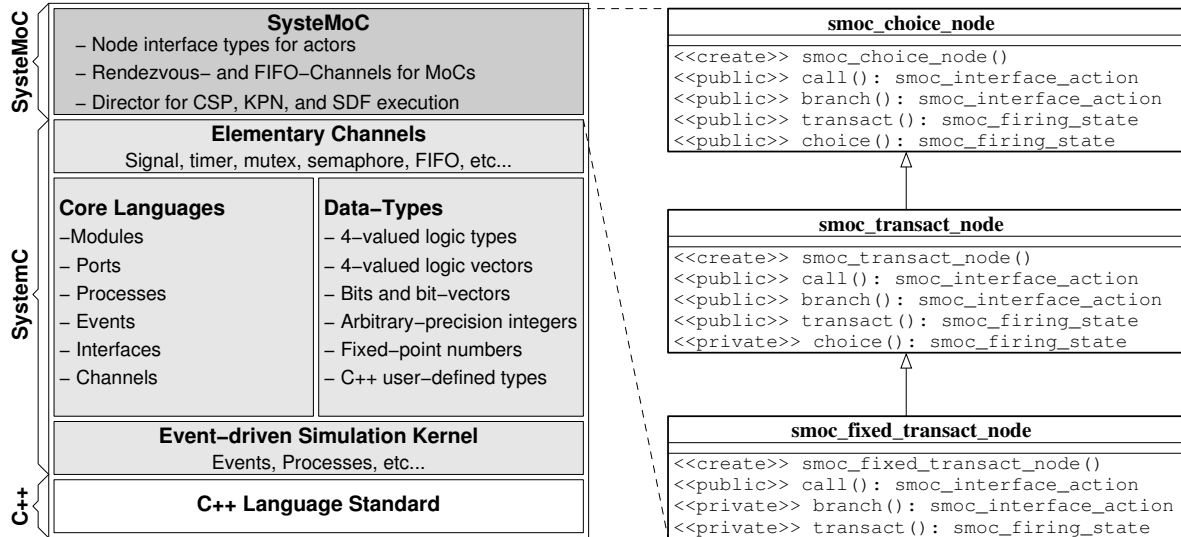


Figure 4: *SystemMoC* is a library built on top of the design language SystemC which allows easy and efficient construction of well defined MoCs. The construction is mainly based on the node interface hierarchy.

SystemMoC library in detail, we will provide some examples of implementing actors with *SystemMoC* next. As such an example Figure 5 shows the Divide actor A_3 in Figure 2. The firing rules are a finite state machine which determines the communication behavior of an actor. The node functionality is only used for algorithmic transformations of data values and any communication with other actors not provided by the firing rules is prohibited. Unfortunately, this cannot be strictly enforced, because it is generally impossible to detect if the C++ code communicates with other actors directly omitting the firing rules which should handle all communication. Hence, we require the *SystemMoC* user to assure that the node functionality does terminate and does only communicate with other actors via the firing rules.

However, some MoCs, e.g., Kahn Process Networks, need a Turing complete control of their communication behavior. One example would be the SqrLoop actor A_2 in Figure 2 which makes an input data dependent decision, e.g, if the approximated square root value is good enough. To accommodate this, we again allow the node functionality to indirectly influence the communication behavior via a conflict transition created by the *Branch* operation. An example of this is shown in Figure 6.

Another case is the possibility to distinguish arrival times of tokens on different ports, e.g., as necessary for a nondeterministic merge actor like the Print actor A_6 in Figure 2. This is implemented via the *Choice* operation as shown for example in Figure 7.

As can be seen from Figure 5 to Figure 7, the restricted communication behavior of an actor is established by choosing an appropriate base class. In *SystemMoC*, the set of operations available for each node interface type are organized in a hierarchy of decreasing execution capabilities and correspondingly increasing analysis capabilities. The hierarchy of the node interface types is transformed into a C++ inheritance hierarchy which can be seen in Figure 4. Operations which are no longer available for node interface types at lower levels in the hierarchy are disabled by declaring them to be private.

Besides the node interface, *SystemMoC* provides operations to construct the network

```

// node interface type is 'fixed transact node'
class Divide: public smoc_fixed_transact_node {
    smoc_port_in<int> i1,i2; smoc_port_out<int> o1; // input port i1,i2, output port o1
    ...
    void divide()          { // A function of the node functionality
        o1[0]=             // update position 0 of the temporary buffer in o1
            i1[0]/i2[0]; } // access position 0 of the temporary buffer in i1 and i2
    ...
    Divide( sc_module_name name ): smoc_fixed_transact_node(name,
        (i1(1)&i2(1)&o1(1)) >> call(divide) /* specification of firing rules */ ) {} };

```

Figure 5: Node functionality and firing rules of the Divide actor A_3 (A'_3) in Figure 2 encoded in the *SystemMoC* framework. As a special exception of the fixed transact node interface the FSM of the firing rules is not explicitly given but instead only a transition t without successor state, e.g, $(i_1(1) \& i_2(1) \& o_1(1)) \gg \text{call}(\text{divide})$, is provided in the constructor. This transition t has the activation pattern $i_1(1)\&i_2(1)\&o_1(1)$ and the interface action \mathcal{J} with function $\mathcal{J}.f = \text{divide}$. The corresponding FSM which is constructed from this transition by the fixed transact node interface can be seen in Figure 3(c). The transition t_5 in the FSM corresponds exactly to the transition t given in the constructor and the transition t_4 in the FSM is constructed by the fixed transact node interface from the transition t by deleting all actor output ports from the activation pattern of the transition. This transition corresponds to an initialization phase where the inputs are acquired which are necessary before the first outputs can be produced.

graph, and define the *node functionality*. The node functionality of an actor is defined in certain member functions, which are called by the firing rules when their requirements for input data is fulfilled. This member functions are not allowed to call communication operations. All required input and output data values are kept in temporary buffers in the actor ports. These buffers are managed by the firing rules which store new data received via actor input ports in temporary buffers in these actor input ports and send data in temporary buffers in actor output ports to the channels connected via these ports. For example, the node functionality of the the Divide actor A_3 from Figure 2 can be seen in Figure 5.

The information about the connections of the actors is stored in the *network graph* which is represented by a user provided C++ class, in the following called network graph class. This class is derived from the *SystemMoC* graph class `smoc_graph`. The network graph is assembled in the constructor of the network graph class by usage of the normal SystemC methods. Instead of the normal SystemC channels like `sc_fifo`, `sc_signal`, etc, the two *SystemMoC channel kinds* `smoc_fifo_kind` and `smoc_rendezvous_kind` are available to connect the actors to each other. The channel kind determines the communication semantics of a channel but makes no assumptions about the data type of the tokens. Further differences between *SystemMoC* channels and SystemC channels are the absence of user callable methods for communicating on the *SystemMoC* channels. Providing those communication methods on the channels would contradict the separation of node interface and channel kind.

Due to space limitations, we only sketch the extraction of the underlying MoC from a SystemMoC description. Note that it is possible to combine different MoCs in *SystemMoC*.

```

class SqrLoop: public smoc_transact_node { // node interface type is 'transact node'
    smoc_port_in<int> i1,i2; // input port i1, i2
    smoc_port_out<int> o1,o2; // output port o1, o2
    ...
    smoc_firing_state ok, again;
    const smoc_firing_state &check();
    ...
    smoc_firing_state fireRules() { // function for construction of the firing rules
        smoc_firing_state start = transact( (i1(1)&i2(1)) >> branch(check, ok|again));
        ok = transact( (o1(1)&o2(1)) >> call(NULL,start));
        again = transact( (o1(1)&i2(1)) >> branch(check, ok|again));

        return start; }
    ...
    SqrLoop( sc_module_name name ): smoc_transact_node(name, fireRules() ) { } };

```

Figure 6: Firing rules of the SqrLoop actor A_2 (A'_2) in Figure 2 encoded in the *SystemMoC* framework. The FSM of the firing rules which corresponds to the code in method `fireRules` can be seen in Figure 3(b). In the *SystemMoC* framework a firing state is represented by an `smoc_firing_state` object, e.g, `smoc_firing_state start` for S_{start} , `smoc_firing_state ok` for S_{ok} , and `smoc_firing_state again` for S_{again} . In this example because of the transact node interface only one outgoing transition per firing state is allowed. These firing states are created via transact method calls, e.g, `start = transact((i1(1) & i2(1)) >> branch(check, ok | again))`. The previous code snippet creates the state S_{start} and its outgoing conflicting transition $t_3 = (k, J)$ which has an activation pattern $k = i_1(1) \& i_2(1)$ (encoded as `i1(1) & i2(1)`) and an interface action J (encoded as `branch(check, ok | again)`). This interface action references the $\text{check} = J.f$ function of the node functionality and has two possible successor states $J.Q_{\text{succ}} = \{S_{\text{ok}}, S_{\text{again}}\}$ (encoded as `ok | again`).

To identify the MoC, in a (partial) *SystemMoC* description, we must investigate the node interface of all actors and the channel kind of connected channels. The following table shows some actor oriented MoCs that can be constructed using *SystemMoC*.

Channel kind	Node interface		
	Choice Node	Transact Node	Fixed Transact Node
Fifo	No well known name	KPN	SDF
Rendezvous	CSP	No well known name	No well known name

5 Conclusions

We have implemented a framework which restricts the communication behavior of actors in SystemC in such a way that the identification, extraction, and analysis of the underlying MoC is permitted. In actor-oriented design, actors only communicate with each other via channels instead of method calls as known in object-oriented design. In our proposed methodology, the specification is distinguished in four different aspects, namely *node functionality*, *node interface*, *network graph*, and *channel kind*. As a result, the SystemC library named *SystemMoC* was developed which allows model-based system design on top of the design language SystemC.

```

class Print: public smoc_choice_node { // node interface type is 'choice node'
    smoc_port_in<int> i1,i2; // input port i1, i2
    ...
    void print();
    ...
    smoc_firing_state fireRules() { // function for construction of the firing rules
        smoc_firing_state start = choice( i1(1) >> call(print,start) |
                                           i2(1) >> call(print,start) );

        return start; }
    ...
    Print( sc_module_name name )
        : smoc_choice_node(name, fireRules() ) {} };

```

Figure 7: Firing rules of the Print actor A_6 (A'_6) in Figure 2 encoded in the *SystemMoC* framework. The FSM of the firing rules which corresponds to the code in method `fireRules` can be seen in Figure 3(a).

References

- [Agh97] G. Agha. Abstracting interaction patterns: A programming paradigm for open distribute systems, 1997.
- [Bai03] Mike Baird, editor. *SystemC 2.0.1 Language Reference Manual*. Open SystemC Initiative, San Jose, 2003.
- [GHJV95] R. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- [GLMS02] Thorsten Grötter, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
- [HDPS04] Deepak A. Mathaikutty Hiren D. Patel and Sandeep K. Shukla. Implementing Multi-MoC extensions for systemc: Adding csp & fsm kernels for heterogeneous modeling. Technical report, FERMAT Research Lab. Center for Embedded Systems for Critical Applications, Virginia Polytechnic Institute and State University Blacksburg, Virginia, 24060, USA, 2004.
- [HSV04] Fernando Herrera, Pablo Sánchez, and Eugenio Villar. Modeling of csp, kpn and sr systems with systemc. pages 133–148, 2004.
- [Lee02] Edward A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press London, London, September 2002.
- [Lee04] Edward A. Lee. Overview of the ptolemy project, technical memorandum no. ucb/erl m03/25. Technical report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, 94720, USA, July 2004.
- [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.
- [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.