In Proceedings of *Forum on Specification and Design Languages 2006* (FDL 2006)
Darmstadt, Germany, September 19-22, 2006, © ECSI.

Efficient Representation and Simulation of Model-Based Designs in SystemC

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

Abstract

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 paper, we propose a library based on the design language SystemC called *SysteMoC* which provides a simulation environment for model-based designs. We will introduce the syntax and semantics supported by *SysteMoC* as well as discuss the simulation environment and present first results of using *SysteMoC* for modeling and simulation of signal processing applications. The library-based approach unites the advantage of executability with analyzability of many expressive MoCs. Finally, we compare the simulative performance of *SysteMoC* with other executable languages such as C++, regular SystemC, and modelling environments such as Ptolemy II.

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. 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. To make industry benefit from the best of both worlds, engineers must restrict themselves to use certain coding styles and subsets of a design language. This results in a model-based design methodology that permits automatic analysis, identification, and extraction of MoCs at the source code level.

In this paper, we propose the *SysteMoC* approach. The basic MoC of the *SysteMoC*-library is *FunState* (Functions driven by State machines) [STG+01]. FunState models express their communication behavior by *Finite State Machines* (FSM). Analyzing these FSMs together with the topology of a given SystemC design permits the extraction and analysis of the underlying MoC to the given design. This is a prerequisite for later optimization or even for design automation approaches. In this paper, we will focus on the *SysteMoC* syntax and semantics as well as the simulation environment for *SysteMoC* designs.

The rest of this paper is structured as follows: In Section 2, we discuss related work. In Section 3, we present *SysteMoC* syntax and semantics. In Section 4, details pertaining to our implementation of the *SysteMoC* simulation environment are presented. In Section 5, we compare the *SysteMoC* simulation performance with other approaches, and we conclude the present paper in Section 6. We will use the example of an approximating square root algorithm throughout the paper to illustrate our approach.

2 Related Work

The advantages of using a model-based approach has been shown by many examples, e.g., for real time reactive systems [BFM+05] and in the signal processing domain [BLM00, BB00]. SystemC [GLMS02, Bai03] permits the modeling of many different MoCs. Unfortunately, there is no unique representation of a given MoC in SystemC. Moreover, the use of unstructured communication makes the automatic analysis of a SystemC design nearly impossible. The SystemC Transaction Level Modeling (TLM) standard [RSPF04] does not alleviate these problems because it is not concerned with defining representations of MoCs in SystemC. Instead, the TLM standard defines transaction level interfaces via method calls, therefore improving simulation efficiency and providing the foundation for platform-based design in SystemC. Habibi et al. [HTS+06, HMT06] have bypassed this problem by specifying their application as *abstract state machines* which they transform into a SystemC-TLM model or requiring the presence of special helper functions which encode knowledge of the communication behavior of the SystemC design.

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 automatically extracted from the executable specification. Implement-

ing these channels on top of the SystemC discrete event simulation kernel curtails the performance of such an implementation. To overcome these drawbacks, Patel et al. [PS05, PS04] have extended SystemC itself with different simulation kernels for *Communicating Sequential Processes* and *Finite State Machine MoCs* to improve the simulation efficiency of their approach.

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. However, its Java implementation limits the ease of integration of this framework with the arising design flow centered on SystemC used by industry for system level modeling.

In contrast to the approaches discussed above, our methodology divides an actor into its *functionality* responsible for *data processing* and a so called *firing FSM* responsible for the communication behavior of this actor. This enables us to automatically extract this communication behavior from the executable specification for later analysis steps. The MoC underlying our methodology is a generalization of *FunState* (Functions driven by State machines) [STG+01]. A FunState model consists of Petri-Nets where the activation of a *transition* is controlled by an FSM. In contrast to FunState we allow all functions of an actor to share a state. Different less general dataflow MoCs can be expressed in FunState and recognized from their FunState description, e.g., *synchronous dataflow* (SDF), *cyclo-static dataflow* (CSDF), *Kahn process networks* (KPN), non-deterministic dataflow, etc. The basis of the analysis steps we will perform with the extracted communication behavior have been detailed by Strehl et al. [Str00] which uses symbolic techniques based on regular state machines [TTS00] to schedule FunState models.

3 *SysteMoC* - syntax and model

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* are objects which execute concurrently and can only communicate with each other via *channels* instead of method calls as known in object-oriented design. *SysteMoC* is a library based on SystemC that allows to describe and simulate communicating actors, which are divided into their *actor functionality* and their *communication behavior* encoded as an explicit finite state machine. In the following, the syntax and semantics of *SysteMoC* designs is discussed.

3.1 Network graph

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 1 for an example of a *network graph*, where c_1 - c_6 denote *FIFO* channels.

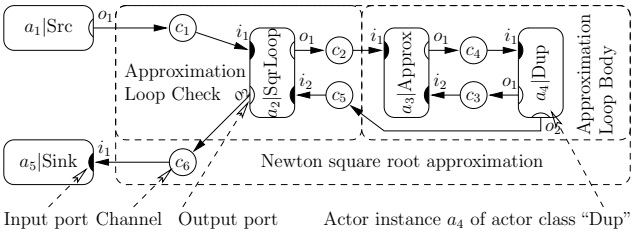

Figure 1: The *network graph* displayed above implements Newton's iterative algorithm for calculating the square roots of an infinite input sequence generated by the `Src` actor a_1. The square root values are generated by Newton's iterative algorithm `SqrLoop` actor a_2 for the error bound checking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

In this paper, we are dealing only with a restricted form of network graphs called *non-conflicting network graphs*, thus allowing us to simply omit the term non-conflicting. A *non-conflicting network graph* allows only *point-to-point* connections per channel. Note, however, that this constraint only applies to the network graph itself. The realization of the communication as expressed by the network graph can be implemented by mapping these channels to buses an other *many-to-many* communication resources. However, the mapping of actors and channels to resources is beyond the scope of this paper and described in more detail in [SFH+06]. More formally, we can derive the following definition:

Definition 3.1 (Non-conflicting network graph) *A* non-conflicting 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* $n \in \mathbb{N}_\infty = \{1, 2, 3, \ldots \infty\}$*, and possibly also a non-empty sequence* $\mathbf{v} \in V^*$ *of initial tokens*[1]*, and finally a set of directed edges* $E \subseteq (C \times A.I) \cup (A.O \times C)$.[2] *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.*

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:

[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].

[2] We use the '.'-operator, e.g., $a.I$, for member access, e.g., I, of tuples whose members have been explicitly named in their definition, e.g., $a \in A$ from Definition 3.2. Moreover, this member access operator has a trivial pointwise extension to sets of tuples, e.g., $A.I = \bigcup_{a \in A} a.I$, which is also used throughout this document.

Example 3.1 Network graph corresponding to Figure 1:

```
// Declare network graph class SqrRoot
class SqrRoot: public smoc_graph {
protected: // Actors are C++ objects
  Src a1; SqrLoop a2; Approx a3; Dup a4; Sink a5;
public: // Constructor assembles network graph
  SqrRoot( sc_module_name name ): smoc_graph(name),
      a1("a1", 50), // parameterize Src actor a1
      a2("a2"), a3("a3"), a4("a4"), a5("a5") {
    // The network graph is instantiated
    // in the constructor
    connectNodePorts(a1.o1, a2.i1);
    connectNodePorts(a2.o1, a3.i1);
    connectNodePorts(a3.o1, a4.i1,
      smoc_fifo<double>(1));    // FIFO  of size 1
    connectNodePorts(a4.o1, a3.i2,
      smoc_fifo<double>() << 2);// Initial token 2
    connectNodePorts(a4.o2, a2.i2);
    connectNodePorts(a2.o2, a5.i1);
  }
};
```

The actors of the network graph, e.g., a_1 - a_5 in Figure 1, are member variables. They can be parameterized via common C++ syntax in the constructor of the *network graph class*, e.g., `a1("a1", 50)` to parameterize `Src` actor a_1 to generate 50 samples. The connections of these actors via FIFO channels are assembled in the constructor of the network graph class, e.g., `connectNodePorts (a1.o1, a2.i1)` which implicitly creates the channel c_1 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 c_n 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.

3.2 Actor classes

As can be seen in Figure 2, each actor can be subdivided into three parts: (i) Actor *input ports* and *output ports*, (ii) Actor *functionality*, and (iii) Actor *communication behavior*, encoded by an explicit *firing FSM*. More formally, we can derive the following definitions:

Definition 3.2 (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 three parts of an actor declaration can also be observed in Example 3.2 below. 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. Accordingly, each *actor instance*, in the following simply called *actor*, is a C++ object of its corresponding actor class.

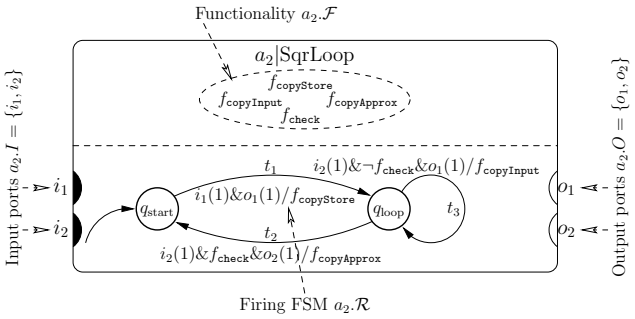

Figure 2: Visual representation of the `SqrLoop` actor a_2 used in the network graph displayed in Figure 1. The `SqrLoop` actor is composed of *input ports* and *output ports*, its *functionality*, and the *firing FSM* determining the communication behavior of the actor.

Example 3.2 Definition of the SqrLoop actor class:

```
// All actor classes are derived from smoc_actor
class SqrLoop: public smoc_actor {
public: // Declaration of input and output ports
  smoc_port_in<double>  i1, i2;
  smoc_port_out<double> o1, o2;
private: // Functionality consists of variables
  double tmp_i1; // storing functionality state,
  // actions transforming func state and data,
  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; }
  // State declaration for the firing FSM
  smoc_firing_state start, loop;
public: // Constructor builds firing FSM
  SqrLoop(sc_module_name name);// code shown later
};
```

The actor port declaration must be located in the *public* part of the actor class to allow the network graph to connect all these actors via their ports. Please note that the usage of `sc_fifo_in` and `sc_fifo_out` ports as provided by the SystemC library would not allow the separation of actor functionality and communication behavior as these ports allow the actor functionality to *consume tokens* or *produce tokens*, e.g., by calling `read` or `write` methods on these ports, respectively. For this reason, the *SysteMoC* library provides its own input and output port declarations `smoc_port_in` and `smoc_port_out`. These ports can only be used by the actor functionality to peek token values already available or to preplace tokens for the actual communication step which is exclusively controlled by the local *firing FSM* $a.\mathcal{R}$ of the actor.

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. The member functions can be partitioned into *actions*, e.g., `copyStore`, and *guards*,

e.g., `check`. 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 are used by the firing FSM to generate tokens for the FIFO channels connected to the actor output ports. In summary, there are two fundamental differences between actions and guards: (i) A guard 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 communication behavior of an actor is encoded in its firing FSM, e.g., as seen in Figure 2. The notion of firing FSM is similar to the concepts introduced in SPI [ZER+99] and an extension of the *finite state machines* in FunState [STZ+01] by allowing requirements for a minimum of space available in output channels before a transition can be taken. The states of the firing FSM are called *firing states*, directed edges between these firing states are called *firing transitions* or *transitions* for short. Each transition is annotated with an *activation pattern*, a boolean expression which decides if the transition can be taken, and an *action* from the *actor functionality* which is executed if the transition is taken. An *activation pattern* may consist of an *input pattern* and an *output pattern*. Input (output) patterns are responsible for checking conditions on the actor input (output) ports, respectively. More formally, we derive the following two definitions:

Definition 3.3 (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 3.4 (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).

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 3.3, the *SysteMoC* representation of the firing FSM of the `SqrLoop` actor a_2, also visually represented in Figure 2, is given.

Example 3.3 Declaration of the firing FSM:

```
SqrLoop::SqrLoop(sc_module_name name)
  : smoc_actor(name, start /* Initial state */) {
  start = // Start state declaration
    // Transition t1 with activation pattern re-
    // quiring at least one token in channel con-
    // nected to port i1 and free space for one
    // token in channel connected to port o1.
    i1(1)                          >> o1(1) >>
    CALL(copyStore)                >> loop; // t1
  loop = // Loop state declaration
    (i2(1) &&  GUARD(check))       >> o2(1) >>
    CALL(copyApprox)               >> start // t2
  | (i2(1) && !GUARD(check))       >> o1(1) >>
    CALL(copyInput)                >> loop; // t3
}
```

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.

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

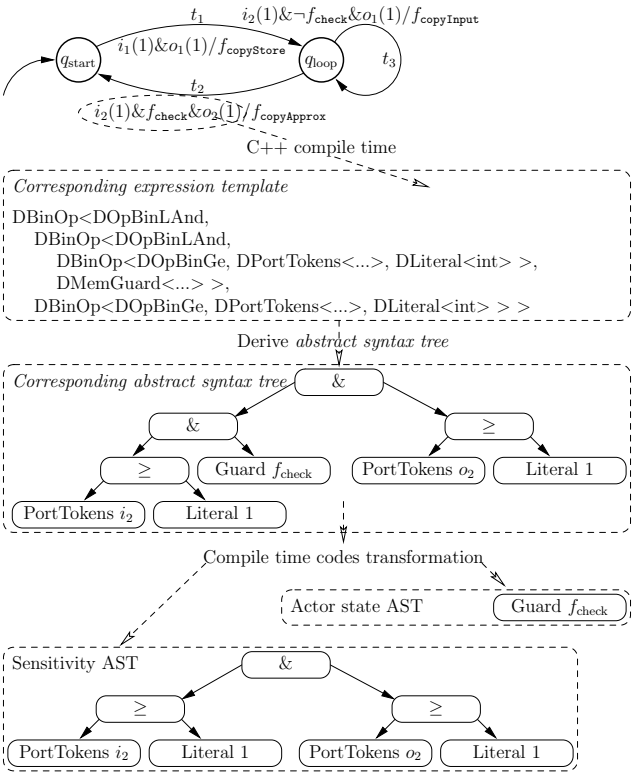

Figure 3: Compile time code transformation of an activation pattern into a sensitivity list used for scheduling and a functionality state-dependent part used to check transition readiness after the scheduling step.

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 3, or (iii) generation of an XML representation of the firing FSM, e.g., as seen in Figure 4, 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 3. 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 sensitivity AST is only evaluated after a change in the number of available or free tokens in the monitored FIFO channels. The actor state-dependent AST part is only evaluated after its corresponding sensitivity AST evaluates to `true`. Note that in general, arbitrarily complex parts dependent on the actor state of an activation pattern can be identified. For these actor state dependent AST parts, dedicated code is generated at C++ compile time for their evaluation.

5 Experimental Results

In this section, we will compare simulative performance of our simulation framework with other approaches using executable specifications. As a case study, we use a two dimensional *inverse discrete cosinus transform*

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

Figure 4: 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.

Table 1: Table comparing the simulation performance for 10000 8x8 blocks of the two dimensional *inverse discrete cosinus transform* (IDCT) implemented in different execution environments.

	C++	*SysteMoC*	SystemC	CAL
Time	9.5ms	2.06sec	2.65sec	33sec

(IDCT) for 8x8 blocks, as displayed in Figure 5. The execution times for the IDCT in different simulation environments were measured on an Intel Pentium IV CPU with 2.80 GHz and 512 MB of main memory via the linux `time` command. In Table 1 the user CPU time needed by the simulator for executing 10000 IDCT iterations is shown.

It can be seen that our approach clearly outperforms the interpretative approach of the *CAL actor language* [WEJ02] in the Ptolemy II modelling environment. However, the SystemC design is of two orders of magnitude slower a pure C++ implementation. On the other hand, due to the reduction of the number of events, our *SysteMoC* approach is faster than an implementation based on SystemC threads and FIFOs. Based on our unique approach to model the communication behavior, we will integrate static scheduling analysis into the *SysteMoC* elaboration phase.That way, we expect to provide a simulation environment with a performance similar to SystemC TLM designs [SD06].

6 Conclusions

We have implemented a framework which *separates actor functionality* and *communication behavior*. The communication behavior is *machine extractable* from the executable specification. As a result, the SystemC library named *SysteMoC* was developed which provides the basis for representing executable specifications within a model-based design flow. Currently, we applied *SysteMoC* to model an industrial relevant applications, and developed a model-based design flow for design space exploration. In future work, we will inte-

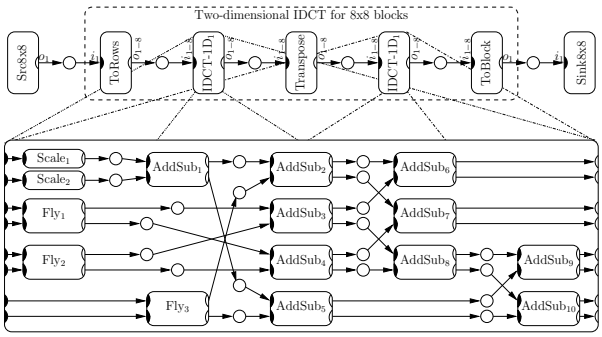

Figure 5: Network graph of the two dimensional *inverse discrete cosinus transform* (IDCT) for 8x8 blocks, e.g., as used in the JPEG and MPEG algorithm. The two dimensional IDCT is composed out of two one dimensional IDCTs, which themselves are composed of primitive arithmetic operations like, e.g., `Scale` or `AddSub`.

grate support to connect *SysteMoC* actors into TLM designs to make industry benefit from analyzable *SysteMoC* designs integrated into standard TLM designs.

References

[Bai03] Mike Baird, editor. *SystemC 2.0.1 Language Reference Manual*. Open SystemC Initiative, San Jose, 2003.

[BB00] B. Bhattacharya and S. Bhattacharyya. Parameterized Dataflow Modeling of DSP Systems. In *Proc. of the International Conference on Acoustics, Speech, and Signal Processing*, pages 1948–1951, Istanbul, Turkey, June 2000.

[BFM+05] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, U. Freund, E. Schlenker, and H.-J. Wolff. Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*. IEEE Computer Society, March 2005.

[BLM00] S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for signal processing systems. *IEEE Transactions on Circuits and Systems*, 47(9), September 2000.

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

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[HMT06] A. Habibi, H. Moinudeen, and S. Tahar. Generating Finite State Machines from SystemC. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, pages 76–81. IEEE Computer Society, March 2006.

[HSV04] Fernando Herrera, Pablo Sánchez, and Eugenio Villar. Modeling of csp, kpn and sr systems with systemc. pages 133–148, 2004.

[HTS+06] A. Habibi, S. Tahar, A. Samarah, D. Li, and O. A. Mohamed. Efficient Assertion Based Verification using TLM. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, pages 106–111. IEEE Computer Society, March 2006.

[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.

[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.

[PS04] Hiren D. Patel and Sandeep K. Shukla. Towards a heterogeneous simulation kernel for system level models: a SystemC kernel for synchronous data flow models. In *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 248–253, New York, NY, USA, 2004. ACM Press.

[PS05] Hiren D. Patel and Sandeep K. Shukla. Towards a heterogeneous simulation kernel for system-level models: a SystemC kernel for synchronous data flow models. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 24, pages 1261–1271, Washington, D.C., August 2005. IEEE Press.

[RSPF04] Adam Rose, Stuart Swan, John Pierce, and Jean-Michel Fernandez. Transaction level modeling in SystemC. 2004.

[SD06] Gunar Schirner and Rainer Dömer. Quantitative Analysis of Transaction Level Models for the AMBA Bus. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, pages 230–235. IEEE Computer Society, March 2006.

[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, March 2006. IEEE Computer Society.

[STG+01] Karsten Strehl, Lothar Thiele, Matthias Gries, Dirk Ziegenbein, Rolf Ernst, and Jürgen Teich. FunState - An Internal Design Representation for Codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, August 2001.

[Str00] Karsten Strehl. *Symbolic Methods Applied to Formal Verification and Synthesis in Embedded Systems Design*. PhD thesis, Swiss Federal Institute of Technology Zurich, February 2000.

[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.

[TTS00] Lothar Thiele, Jürgen Teich, and Karsten Strehl. Regular State Machines. *Journal of Parallel Algorithms and Applications*, 15:265–300, December 2000.

[Vel95] Todd Veldhuizen. Expression Templates. In *C++ Report*, Vol. 7 No. 5, pages 26–31. SIGS Publications, New York, June 1995.

[WEJ02] Edward D. Willink, Johan Eker, and Jörn W. Janneck. Programming Specifications in CAL. In *OOPSLA 2002, Workshop Generative Techniques in the context of Model Driven Architecture*, November 2002.

[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.