SysteMoC

Verification and Refinement of Actor-Based Models of Computation

Joachim Falk, Jens Gladigau, Christian Haubelt, Joachim Keinert, Martin Streubühr, and Jürgen Teich

{falk, haubelt}@cs.fau.de

Hardware-Software-Co-Design
University Erlangen-Nuremberg, Germany
Outline

- Introduction to SysteMoC
- Quasi-Static-Scheduling (QSS)
- Scheduling with non-determinate transitions
  - Applying QSS to SysteMoC
  - Handling non-determinate transitions
- Conclusions
SystemC-based designs

Fact: Restricted MoCs allow efficient analysis and represent a good tradeoff between expressiveness and analyzability:
  - Can be exploited by code generators to produce efficient code.
  - Can be used by verification tools to check system properties.

However, there are many (infinite) ways to represent MoCs in SystemC (undecidable).

We need a well defined subset of SystemC

SysteMoC

www12.cs.fau.de/research/systemoc
Why SystemC?

Structure

Behavior

System

Module

Module

Architecture

Architecture

SW

SW

HW

HW

Best

Best

Logic

Logic

Good

Good

Structure

Best

Best

OK

OK

OK

OK

Block

Block

16.11.2006 Friedrich-Alexander-Universität Erlangen-Nürnberg
SystemC-based designs

What MoC?
**SysteMoC**

**SysteMoC Future** SR and CSP - Domains

- Elementary channels separating functionality and communication
- C++ syntax for specifying actor communication behavior as FSMs
- Facilities to extract these FSMs as well as the network connecting the actors
- Scheduler for dataflow MoC domain
- Etc.

**Elementary Channels**
Signal, timer, mutex, semaphore, FIFO, etc.

**Core Language**
- Modules
- Ports
- Processes
- Events
- Interfaces
- Channels

**Data-Types**
- 4-valued logic types (01zx)
- 4-valued logic vectors
- Bits and bit-vectors
- Arbitrary-precision integers
- Fixed-point numbers
- C++ user-defined types

**Event-driven Simulation Kernel**

**C++ Language Standard**
SysteMoC Idea

- **Network graph**
  Provides communication infrastructure for its actors (directed graph, compare with DIF)

- **Channel kind**
  Gives semantics to the arrows in the network graph.

- **Actor functionality**
  Encodes the functions which transform the token values

- **Firing FSM**
  Encodes communication behavior of an actor and triggers the actor functionality
class m_sqrloop:
    public smoc_actor {
    public:
        smoc_port_in<double> i1, i2;
        smoc_port_out<double> o1, o2;
        m_sqrloop(sc_module_name name) …
    private:
        double toSquare;
        // action functions for FSM defined in constructor
        void store() { toSquare = i2[0]; o1[0] = i2[0]; }
        void supply() { o1[0] = toSquare; }
        void forward() { o2[0] = i1[0]; }
        // guard functions for FSM defined in constructor
        bool check() const { return fabs(toSquare – i1[0]*i1[0]) < ACCURACY_BOUND; }
        smoc_firing_state start;
        smoc_firing_state loop;
    };
m_sqrloop(sc_module_name name) : smoc_actor(name, start) {
  start = i2(1) >>
          o1(1) >>
          CALL(m_sqrloop::store) >>
          loop;

  loop = (i1(1) && GUARD(m_sqrloop::check)) >>
         (o2(1)) >>
         CALL(m_sqrloop::forward) >> start|
         (i1(1) && !GUARD(m_sqrloop::check)) >>
         (o1(1)) >>
         CALL(m_sqrloop::supply) >>
         loop;
  ...
}
FunState, Thiele et al. [TSZ+99]

(transformative) A

(f1) f1 \rightarrow q1

(f2) f2 \rightarrow q2

(transformative) B

(q1 \geq 3/f3)

(f3) f3 \rightarrow q3

(q2 \geq 2/f4)

(f4) f4 \rightarrow q4

(q3 \geq 2/f6)

(f6) f6 \rightarrow q6

(q4 > 0/f7)

(f7) f7 \rightarrow q7

(reactive) C

(f2) f2 \rightarrow q2

(f5) f5 \rightarrow q5

(f6) f6 \rightarrow q6

(q4 = 0)
Dynamic scheduling

- Check each actor for enabled transitions
- Execute actions of enabled transitions
- Consume and produce tokens

1) Evaluate expression
2) Execute action
3) Token consumption/production
The Problem

- Dynamic scheduling at runtime costs time and resource overhead.
- Furthermore no predictions about the runtime behavior of the system are available.

Solution:

- Partitioning the state space into compile time and run time determined state spaces and deriving from the compile time determined state space all possible scheduling decisions determinable.

Quasi-Static-Scheduling
Outline

➢ Introduction to SysteMoC

➢ Quasi-Static-Scheduling (QSS)

➢ Scheduling with non-determinate transitions
  ▪ Applying QSS to SysteMoC
  ▪ Handling non-determinate transitions

➢ Conclusions
Symbolic Scheduling, Strehl et al. [STZ+99]
Regular Finite State Machines

- **Regular Finite State Machine (RFSM)** represents all possible state and state transitions of the design
  - Number of token on channels
  - States of control FSM

\[ I_0 = [1, 1]^T \]
Analysis

- Determination of a quasi-static schedule
- Deadlock detection
Conflicts are properties of transitions

Symbolic scheduling as presented by Strehl et al. requires a clear distinction between compile time and runtime decisions, i.e., states are labeled as **alternatives** modeling scheduling possibilities or as **conflicts** representing 1 out of N runtime decisions.

However, the world is not always that simple.
Outline

- Introduction to SysteMoC
- Quasi-Static-Scheduling (QSS)
- Scheduling with non-determinate transitions
  - Applying QSS to SysteMoC
  - Handling non-determinate transitions
- Conclusions
Conflicts are properties of transitions

- Example: packet rewrite engine
  - Actions $f_a$ and $f_b$ transform packets classified by guards $g_A$ and $g_B$
  - Thus, $t_1$ is in conflict with $t_2$, and vice versa
  - Some packets require both transformations ($g_A$ & $g_B$)
  - And $t_1$ and $t_2$ are an alternative for $t_3$

Solution:
Order graph (later)

$t_1$: $i_1(1) \& o_1(1) \& g_A / f_a$
$t_2$: $i_1(1) \& o_1(1) \& g_B / f_b$
$t_3$: $i_1(1) \& o_1(1) \& g_A \& g_B / f_{a,b}$
Multirates can be involved in conflicts

Input and output patterns of conflicting transitions may differ.

The conflicting transitions $t_1$ and $t_2$ may not always be enabled at the same instant of time.

$t_1$: $i_1(2) \& o_1(2) \& g_A / f_a$  
$t_2$: $i_1(1) \& o_1(1) \& \neg g_A / f_b$

$t_3$: $i_1(1) \& o_1(1) \& g_B / f_c$
$t_4$: $i_1(1) \& o_1(1) \& \neg g_B / f_d$
Scheduling with non-determinate transitions

- Our scheduling methodology improves on symbolic scheduling by the following points:
  - Representation of full dynamic schedule: We use the unmodified existent model for scheduling and so avoid explicit construction of a schedule specification automaton.
  - Detection of runtime decision: Transitions determine runtime decisions. A runtime decision is needed, if a non-determinate transition is used. This is more natural to the model designer than (manual) classification of state machine states into alternative and conflict states.
  - Nature of conflicts: We can model and in particular cases schedule transitions with runtime conflicts and different rates. This case was not considered by Symbolic-Scheduling.
  - Paths to plan: we plan only strictly needed paths, if similar transitions occur in conflicts. This is achieved by using so called order graphs.
Order graph

- Partial order on equivalence classes of transitions
  - Transitions are equivalent if their functionality condition is equivalent and merged into equivalence classes.
  - A partial order on these equivalence classes is defined by the implication operation on their functionality conditions.

- Order graph determines which transitions must be included in the quasi-static schedule.

\[
\begin{align*}
  t_1 &: i_1(1) \& o_1(1) \& g_A \quad / f_{a,1} \\
  t_2 &: i_1(1) \& o_1(1) \& g_B \quad / f_b \\
  t_3 &: i_1(1) \& o_1(1) \& g_A \& g_B \quad / f_{a,b} \\
  t_4 &: i_1(1) \& o_1(1) \& g_A \quad / f_{a,2}
\end{align*}
\]

Order graph:
Outline

- Introduction to SysteMoC
- Quasi-Static-Scheduling (QSS)
- Scheduling with non-determinate transitions
  - Applying QSS to SysteMoC
  - Handling non-determinate transitions
- Conclusions
Applying QSS to SysteMoC

- The compile time determined state spaces is defined by the fill sizes of all channels and the firing states of all actors.
- For each firing state in all actors the associated order graph is computed.
- The order graph encodes for each transition $t$ leaving this state its set of conflicting transition equivalence classes $conflict(t)$.
- If a transition $t$ is used in a path at least one transition per equivalence class of $conflict(t)$ must be included in the QSS to satisfy runtime conditions.
Handling non-determinate transitions

\[
scheduleGraph(a, G, T_{disabled}, T_{cover}) = \text{findPath}(a, G, T_{disabled}, T_{cover})
\]

expand \( G \) by \((x_0 - t_0 -> x_1 - t_1 -> ... - t_{n-1} -> x_n)\)

for \( i = 0 \) to \( n-1 \)

\[ C = \text{conflict}(t_i) \]

\[
\text{foreach } c \in C
\]

\[ G = scheduleGraph(x_i, G, \text{implies}(t_i) \cup T_{disabled}, c) \]

return \( G \)
Outline

- Introduction to SysteMoC
- Quasi-Static-Scheduling (QSS)
- Scheduling with non-determinate transitions
  - Applying QSS to SysteMoC
  - Handling non-determinate transitions
- Conclusions
Conclusions

- SysteMoC as basis to identify the underlying MoC in a SystemC design
- Quasi-static scheduling by enhancing the approach by Strehl et al.
- Current and future work
  - Integrating the S/R domain into SysteMoC
  - Identify and implement formal transformations, e.g., clustering, for synthesis on SysteMoC designs
  - Extend symbolic approach used by quasi-static scheduling to support formal functional verification of SysteMoC designs