ARTIST II Network of Excellence

Workshop on Models of Computation and Communication

November 16-17, 2006, ETH Zurich

Organized by Lothar Thiele[1] and Paul Caspi[2]

Minutes of the workshop by Alain Girault[3]

Edward Lee: Concurrency demands new foundations for computing

The set of functions $Q = (B^{**} \rightarrow B^{**})$ is not countable, where B^{**} is the set of finite and infinite sequences of bits. A program is a computable function, and the set of finite programs is countable. But many subsets $A \subset Q$ are effectively computable functions. Program composition by call/return is the simple composition of functions.

In single-threaded execution, this is fine. But in multi-threaded execution, the essential and appealing properties of computation are lost: Programs are no longer functions, composition is no longer function composition, very large numbers of behaviors may result, and we can't tell when programs are equivalent. Sadly, this is how most concurrent computation is done today.

Multithreading (where two or more programs share the data path) as a programming model is appealing because it remains in the same syntactic domain, but the semantics is thrown away.

The simple observer pattern (from the book "Design Patterns", by Gamma, Helm, Johnson and Vlissides) uses two methods, addListener and setValue. But this does not work in multithreading, because of possible interrupt of one method by the other.

```
public void addListener(listener) {…}
public void setValue(newValue) {
   myValue = newValue;
   for (int i = 0; i < myListeners.length; i++) {
      myListeners[i].valueChanged(newValue)
   }
}
```


[1] Swiss Federal Institute of Technology, Computer Engineering and Networks Laboratory, ETZ-G87, Gloriastrasse 35, CH-8092 Zurich, Switzerland.
[2] CNRS, Laboratoire Verimag, Centre Equation, 2 avenue de Vignate, 38610 Gieres, France.
[3] INRIA Rhône-Alpes, Pop Art team, 655 avenue de l'Europe, 38334 Saint-Ismier cedex, France.

Mutual exclusion solves this problem, but it can lead to deadlock, because valueChanged() may attempt to acquire a lock on some other object and stall; if the holder of that lock calls addListener(), there is a deadlock!

```
public synchronized void addListener(listener) {…}
public synchronized void setValue(newValue) {
   myValue = newValue;
for (int i = 0; i < myListeners.length; i++) {
      myListeners[i].valueChanged(newValue)
   }
}
```

And now this is very difficult to correct:

```
public synchronized void addListener(listener) {…}
public void setValue(newValue) {
   synchronized(this) {
      myValue = newValue;
      listeners = myListeners.clone();
   }
   for (int i = 0; i < listeners.length; i++) {
      listeners[i].valueChanged(newValue)
   }
}
```

The above code still does not work: suppose two threads call setValue(); one of them will set the value last, leaving that value in the object, but listeners may be notified in the opposite order; the listeners may be alerted to the value changes in the wrong order!

Code that had been in use for four years, central to Ptolemy II, with an extensive test suite with 100% code coverage, design reviewed in yellow then green in 2000, caused a deadlock in 2004 during a demo!

So, threads are wildly non deterministic. The programmer's job is to prune away the non-determinism by imposing constraints on the execution order (e.g., mutexes) and limiting shared data accesses (e.g., OO design).

It is not concurrency that is hard, it is threads, that is, the abstraction we have chosen to program concurrent behaviors. This is because threads are sequential processes that share memory. From the perspective of any thread, the entire state of the universe can change between any two atomic actions (itself an ill-defined concept). Yet threads are the basis of many widely used programming languages.

We need some other model of concurrency that is deterministic, and then add non-determinism where it is needed.

To do this, we need to replace the core notion of computation, from:

$$P: B^{**} \rightarrow B^{**}$$

into:

$$F: (T \rightarrow B^{**}) \rightarrow (T \rightarrow B^{**})$$

where T is a partially or totally ordered set. This is the tagged signal model: (T -> B**) is called a signal. Composition of actors is just the intersection of their respective sets of behaviors.

Composition itself does not introduce non-determinism.

The algebraic properties of the tag set T are determined by the concurrency model, e.g., process networks, synchronous/reactive, time-triggered, discrete event, dataflow, rendezvous, continuous time, hybrid systems…

But, this is not what mainstream programming language does, nor what mainstream component models do. So we need an actor-oriented design instead of the classical object-oriented design. In actor-oriented, the model is data-flow rather than call-return. Actually, many languages are based on this principle (Simulink, Ptolemy, Modelica, LabView, Corba, VHDL, Verilog, Occam…), many of which use some form of graphical syntax. The semantics of these languages differ considerably, but all can be modeled as

$$F: (T \to B^{**}) \to (T \to B^{**})$$

with appropriate choices of the set T.

Semantics of these coordination languages is what this MoCC workshop is about.

The simple observer pattern becomes very easy to specify in such a coordination language.

By the way, imperative reasoning is simple and useful, so it should be kept, but it requires careful implementation. Reconciling imperative and actor semantics is done with state-full actors, represented by two functions that give respectively the outputs in terms of inputs and the current state, and update the state.

Axel Jantsch: ForSyDe: a denotational framework for heterogeneous MoCCs

ForSyDe integrates different MOCs. Processes communicate through signals only, are functional, state-full, reading is blocking, and evaluate when their required inputs are available. Signals are sequences of events, they preserve the order of events, they have one writer and several readers, in untimed MoCs events are partially ordered while in timed MoCs signals carry also the timing information.

The ForSyDe design flows starts with an ideal system model (no resource limitation) and obtains an implementation model (with finite resources). Finally, a backend code generator produces a C program, a VHDL design, or a SystemC model.

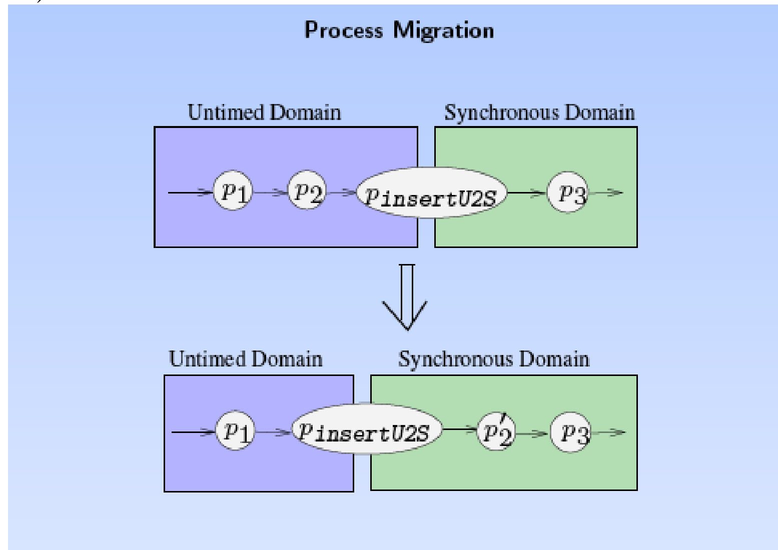
A ForSyDe process consists of a constructor, a function, an initial state, and an invocation condition. Existing MoCs include dataflow, SDF, rendezvous, synchronous, discrete time, and soon continuous time.

Process combinators include sequential combination, parallel composition, and feedback composition.

Process constructor types include state-less processes, FSM machines, Zip / Unzip, and sources and sinks.

A MoC is formally defined as a set of process combinators and a set of process constructors.

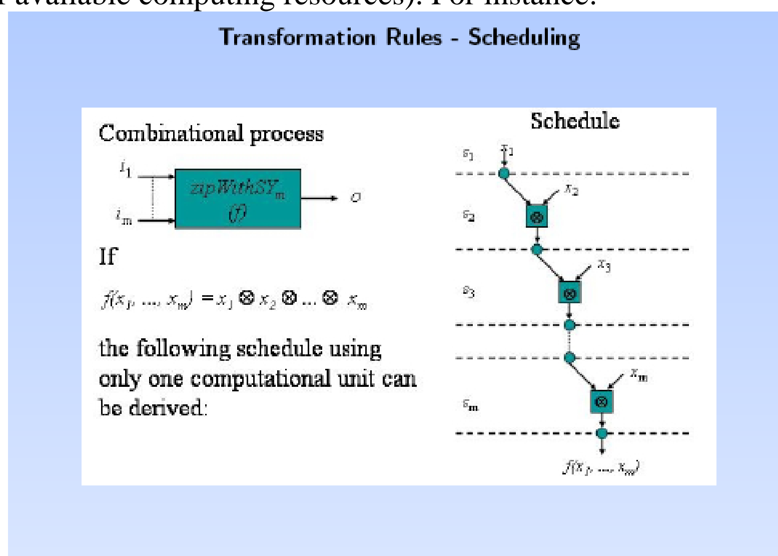
From one MoC to another one, specific actors are required to rearrange the signals (i.e., the sequence of events).



It is possible to migrate a process from one MoC to another one by applying a migration pattern (this is left to the user's choice), which transforms an assembly of actors into another assembly of actors, where one actor has migrated to a new MoC (so its behavior has changed). Some patterns yield equivalent programs, while others yield proof obligations left to the user.

Process migration can be used as a refinement step.

Scheduling can be achieved by using a specific transformation rule (each rule being adapted to the number of available computing resources). For instance:



Ongoing work includes a continuous time MoC, modeling non-functional properties, modeling adaptative resources, a GME based tool support for transformations, and a communication refinement method.

Benoît Caillaud: *Microstep endo/isochrony*

The goal of the proposed method is to design a system with one synchronous MoC and then map it *safely* towards a different MoC (asynchronous, distributed, message-passing).

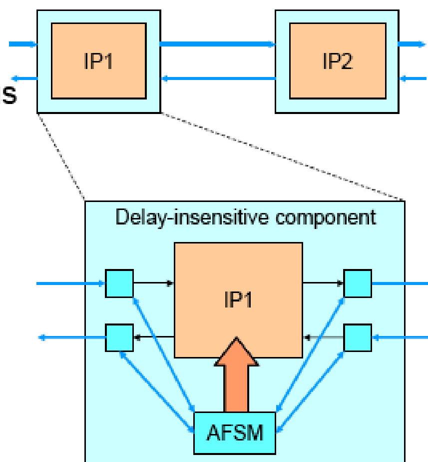
The synchronous MoC has a global clock available everywhere; specification and verification are therefore easier; this is a popular MoC for digital circuits and safety critical systems; efficient code generation tools exist.

The principle of the method is to add a *wrapper* around each synchronous actor, for the asynchronous communication through FIFOs. The wrapper should preserve: functionality, correctness (no extra traces, no deadlock), and parallelism.

What we want

1. Take a modular synchronous specification
2. Replace comm. with asynchronous FIFOs, wrappers
3. Preserve:
   - Functionality
   - Correctness
     - No “extra” traces
     - No deadlocks

   (Kahn processes)
   - Parallelism



This work is related to latency insensitive design and to weakly endo-isochronous systems. It solves the drawbacks of all these previous works (e.g., compositionality…).

The core model is the LTS where transitions are labeled by tuples of variable assignments. Composition is by synchronized product. The generalized model is GCTS (generalized concurrent transition system).

A synchronous 1-place buffer SFIFO can be defined in this model, and synchronous composition is just the product of the actors with the adequate SFIFO actors.

An asynchronous FIFO can be defined in this model, and asynchronous composition is just the product of the actors with the adequate FIFO actors.

So the same system can be mapped into a synchronous implementation or an asynchronous implementation at will.

The formal correctness criterion is that every trace of the asynchronous composition can be completed to one that is equivalent to a synchronous trace.

Microstep weak endochrony requires a compositional delay-insensitive criterion (i.e., signal absence information is not needed). This is the same criterion as weak endochrony but expressed at a lower level of abstraction (i.e., actions are not considered as atomic).

- Axiom 1: determinism
- Axiom 2: in every state, non-clock transitions sharing no common variables are independent.
- Axiom 3: non-contradictory reactions can be united
- Axiom 4: conflict does not change with time
- Theorem 1: microstep weak endochrony is compositional (i.e., the composition of two weakly endochronous systems is weakly endochronous).

If n systems are weak non-blocking and weak endochronous, then their asynchronous composition is correct w.r.t. their synchronous composition.

In conclusion, we have decidable criteria (microstep weak endo and isochrony) for GALS implementation of synchronous specifications, which cover causality and read/write communication, and which are compositional.

Future work will address synthesis: how to make synchronous automata weakly endo/isochronous (there are optimality issues here); to propose heuristics for actual synchronous languages and specifications (there are scaling issues for large specifications); how to produce GALS circuit using asynchronous logic; and how to deal with mode changing latency.

Paul Caspi: From loosely time-triggered systems to a taxonomy of MoCCs

A popular model of system (used in Airbus for the A320 flight control software) is the *loosely time-triggered framework*, which involves several periodic sub-systems communicating through a periodic bus. Clocks are as periodic as possible, but *not synchronized*. When the bus is triggered (by its clock), the current writer's value is stored in the bus. When the bus reader is triggered (by its clock), the value stored in the bus is read.

The tag system model generalizes this framework: signals are sequences of pairs (value,tag), behaviors are tuples of named signals, and processes are sets of behaviors. Process composition is by unification, and heterogeneity is handled by tag morphisms.

Remark: in Lee and Sangiovanni-Vincentelli tagged signal model, a signal is a function from pairs (tag,value) to Booleans, instead of a sequence of (value,tag):

Comparison with Lee & Sangiovanni

Benveniste	LSV
$Signals = Naturals \rightarrow Tags \times Values$	$Signals = Tags \times Values \rightarrow Booleans$
$Behaviours = Names \rightarrow Signals$	$Behaviours = Names \rightarrow Signals$
$Processes = Behaviours \rightarrow Booleans$	$Processes = Behaviours \rightarrow Booleans$

How can we understand it?

Paul Caspi proposes several taxonomy elements for tags: tags can be seen as a partial order (it can be a non-total order POT or a total order TOT), and they can be countable (continuous time, CT) or not (discrete time, DT).

He then proposes taxonomy elements for signals: signals can be non-totally ordered (POS) or totally ordered (TOS), and they can be countable (continuous signals, CS) or not (discrete signals, DS).

So, with respect to the ordering there are three possible cases:

PO = POT + POS
TOS = POT + TOS
TO = TOT + TOS

And with respect to the countability, we have three possible cases:

C = CT + CS
DS = CT + DS
DT = DT + DS

For instance, C and TOT is continuous time, DS and TOS is timed data flow, DS and TOT is discrete events, DT and TOT is synchrony… Other combinations are less obvious: C and PO is surfaces (this notion is unclear even to Paul Caspi), DS and PO is timed trees…

Taxonomy Elements

9 cases :

	PO	TOS	TOT
C	surfaces?	curves?	continuous time
DS	timed trees	timed data flow	discrete events
DT	trees	data flow	synchronous

Not included: determinism, dynamic creation

Note that determinism and dynamic creation are not considered in this taxonomy. Taking them into account would make the global picture much more complex.

How can we move from a MoCC to another one? Such transductions also define the needed interfaces between MoCCs. Some transductions are for free (e.g., from TO to PO, identity can be used since a total order is also a partial order). Some transductions require tag morphisms (e.g., from PO to TO). Some transductions require signal dependent morphisms (e.g., sampling from C to DT where the morphism depends on the sampling clock, or holding from DT to C where the morphism depends on the holding clock).

Sébastien Gérard: Accord-UML: a methodological approach for model-based development and validation of real-time embedded systems

Real-time and embedded (RT/E) quantitative features: deadlines, WCET, periods, and power consumption.

Qualitative features: related to computation (concurrency and synchronization), or to communication (synchronization modes).

Refinement issues: provide specific execution platform for supporting RT/E MoCCs (either HW or SW execution platform), and provide transformations to map systems to these platforms.

UML model: objects with behavior and communicating by message passing.

UML MoComm: operation based message or signal-based message.

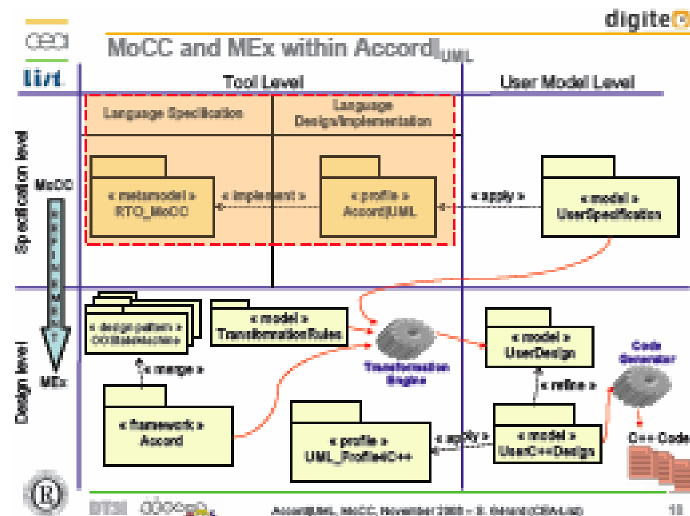
UML MoComp: active objects are concurrent units (they have their own thread of execution), while passive objects are computation resources to be used by active objects.

Accord-UML is a profile that fills some open parts of the UML semantics.

The run-to-completion UML semantics of a state machine involves four steps: initialization, loop {waiting for events, handling an event}, and termination.

Accord-UML provides an RTObject: a usual OO view that encapsulates data and processing. It is based on the UML definition of an Active Object. From the user point of view, it is an object that performs itself the control of its processing, with a standard UML object interface (ports, types, and even interaction protocols).

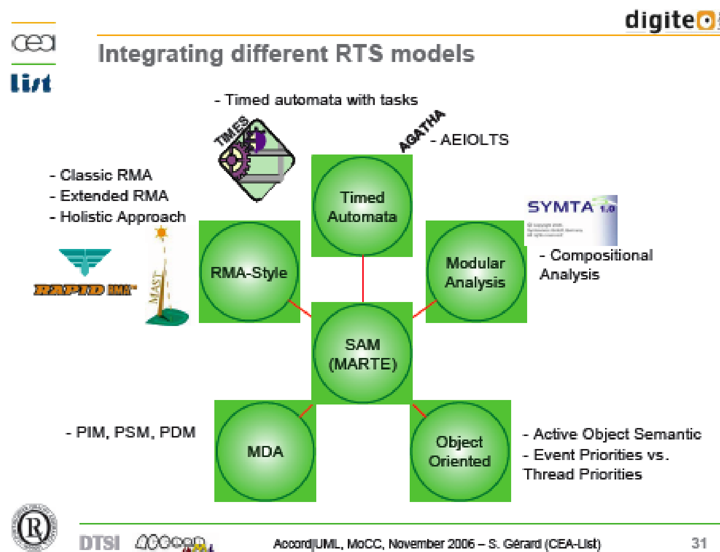
At the specification level, Accord-UML is a MoCC. At the design level, it is a Model of Execution (MEX), which provides a transformation engine and code generators.



The result is that Accord-UML allows the user to perform model-driven design.

This will be applied to the SW plant project ("usine logicielle" in French) within the Systematic French pole of competitiveness.

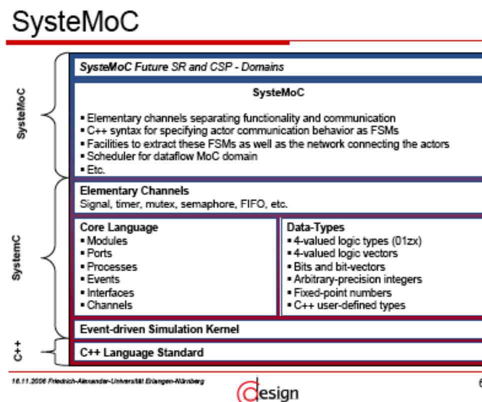
By the way, MARTE is also a profile for RT UML. The difference with Accord-UML is that MARTE is a standard approved by the OMG:



[Christian Haubelt](#) & [Joachim Falk](#): SysteMoC: verification and refinement of actor based models of communication

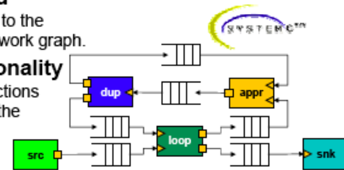
Restricted MoCs allow efficient analysis and represent a good tradeoff between expressiveness and analyzability. They can be exploited by code generators to produce efficient code, and they can be used by formal verification tools to check properties.

[SysteMoC](#) is a subset of SystemC, proposed to represent MoCs within SystemC. SysteMoC has elementary communication channels and actors. An actor has input and output ports, several functions, an FSM, and it implements a functionality.



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



design 7

Dynamic scheduling is used to schedule SysteMoC actors. It involves the following steps: check each actor for enabled transitions, execute the actions of enabled transitions, and consume and produce tokens for the wires. The problem is that dynamic scheduling causes resource overhead and costs time at runtime, and no prediction can be made. The proposed solution is *quasi-static scheduling*: this involves a partition of the state space into compile-time and run-time.

Quasi-static scheduling is based on symbolic scheduling and Regular Finite State Machines (RFSM) by [Strehl et al, 1999]. An RFSM represents all the possible states and transitions of the design. It allows the computation of a quasi-static schedule and the detection of deadlocks.

However, there may be conflicting transitions: the solution involves a partial ordering of the transitions at compile-time. Haubelt and Falk propose several improvements of symbolic scheduling to manage non-determinate transitions:

- Representation of full dynamic schedules;
- Detection of runtime decisions;
- Modeling of transitions with runtime conflicts and different rates;
- Plan only strictly needed paths.

Future work involves integrating the S/R domain into SysteMoC, identifying and implementing formal transformations (e.g., clustering), and extending the symbolic approach used by quasi-static scheduling to support formal functional verification of SysteMoC designs.

***Johan Lilius* & *Lionel Morel*: *Rialto*: a language for heterogeneous computations**

The context is the design of heterogeneous embedded systems (e.g., mobile phones). The goal is to develop a uniform mathematical model for MoCs, and propose a textual language to program them. This is Rialto. The version 2.0 uses reflectivity interfaces (i.e., access to the program state and the program structure) to define policies using the Rialto syntax, and is translated into Rialto 1.0 to gain access to the compiler.

Below are the features of some popular programming languages for embedded systems:

Introduction | Syntax and Semantics | Policies | Conclusions and Future Work

Language features

Feature	Language					
	Esterel	Polis	SDL	SystemC	CCSS	UML Sc
Concurrency	X	X	X	X	X	X
Hierarchy	X	X	X	X	X	X
Preemption	X			X	X	X
Deterministic	X			O	X	
Communication:					X	
Synchronous	X			X		
Buffered		X	X	X		X
FIFO			X	O		X
Procedural	X	O	O	X	O	
FSM	X	X	X	O	X	X
Dataflow		X	X	X	X	
Multi-rate DF					X	
Software	X	X	X	X	X	X
Hardware	X	X		X	X	

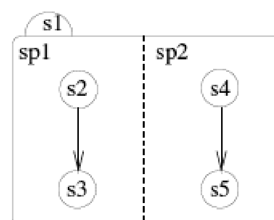
ARTIST2 Workshop on MOCCs J. Lilius & L. Morel Zurich, Nov 16th-17th 8

Many languages use the same syntactic concepts but with different semantics (e.g., for concurrency, interrupts, sequence, choice, atomicity, encapsulation). Programming in Rialto allows us to pinpoint these semantics differences.

The syntax of Rialto is a textual version of the classical hierarchical concurrent FSMs. Below is an example of a Rialto program:

Introduction | Syntax and Semantics | Policies | Conclusions and Future Work

Language features (example)



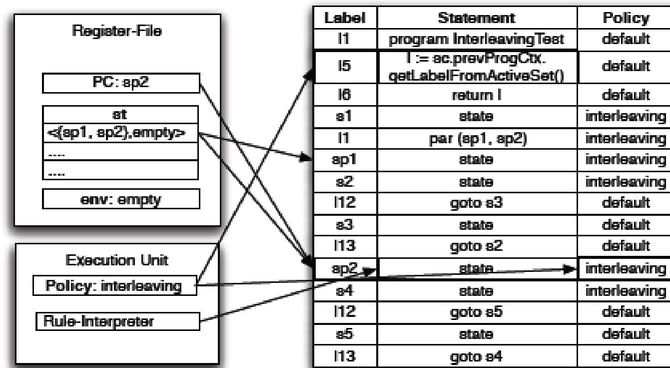

```
1 begin
2 s1 : state
3       policy interleaving;
4 begin
5 l1 : par
6 sp1 : state
7         begin
8 s2 :       state
9            begin
10              goto s3;
11           endstate; // s2
12 s3 :      state
13           begin
14              goto s2;
15           endstate; // s3
16        endstate; // sp1
```

```
17 ||
18 sp2:  state
19        begin
20 s4:       state
21           begin
22              goto s5;
23           endstate; // s4
24 s5:       state
25           begin
26              goto s4
27           endstate; // s5
28        endstate; // sp2
29 endpar;
30 endstate; //s1
31 end;
```

```
1 program InterleavingTest
2     policy interleaving
3         var l: label;
4         begin
5             l := sc.prevProgCtx.
6                 getLagelFromActiveSet();
7             return l;
8         end;
```

ARTIST2 Workshop on MOCCs J. Lilius & L. Morel Zurich, Nov 16th-17th 10

There is a hierarchy of blocks, and a matching hierarchy of policies that define how the blocks must be understood. The semantics is 2-level based on a SOS formalization. Rialto has an execution machine, which implements the execution policies.

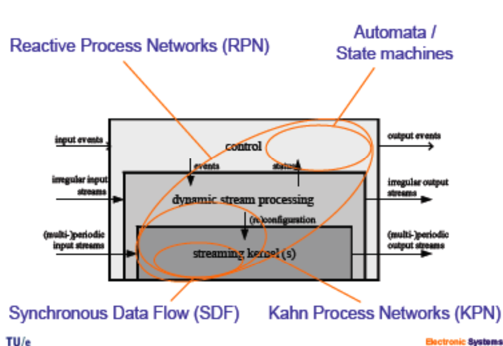


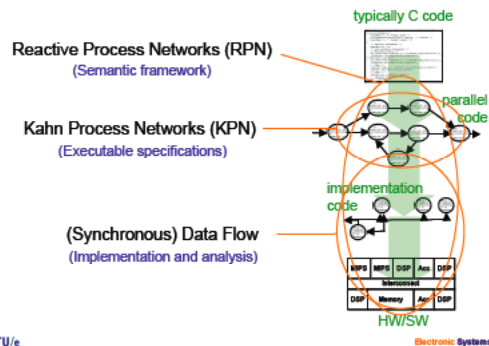
It has a UML front-end, so it allows the definition of a semantics for (subsets of) UML.

Future work involves the exploration of efficient HW/SW implementations from Rialto. Connecting Rialto to the tagged value model or to ForSyDe would allow us to give it a denotational semantics (trace semantics). This would then allow us to prove that the traces of a Rialto program in a certain MoC have the properties specified in the tagged value model.

Marc Geilen: Modeling, analysis, and scheduling with dataflow models

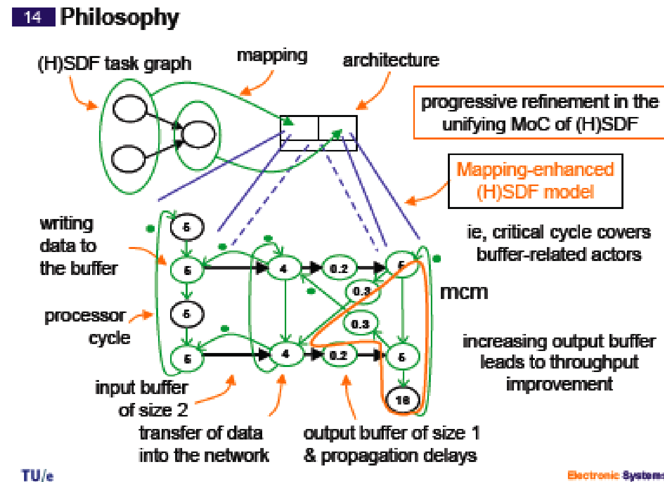
The application domain is multimedia. The model has three layers: SDF, dynamic stream processing, and control:





Expressiveness hierarchy: HSDF (Homogeneous SDF), SDF, BDF, KPN (Kahn Process Networks), DDF, RPN (Reactive Process Networks).

The principle is to perform progressive refinements in the unifying MoC of HSDF:



The formal analysis of a system therefore requires the translation of SDF into HSDF, which leads potentially to an exponential explosion. Examples of analyses: throughput computation; latency definition for SDF; liveness, boundedness, and consistency (based on throughput analysis); computation of the minimum buffer sizes (based on state-space exploration and critical cycle analysis); buffer size versus throughput tradeoffs (Pareto optimal tradeoffs). These analyses are implemented in the SDF3 tool.

The inherent limitations of SDF are: a static and periodic behavior, fixed execution times, and no dynamic reconfiguration. The considered extensions are KPH, Scenarios, and RPN.

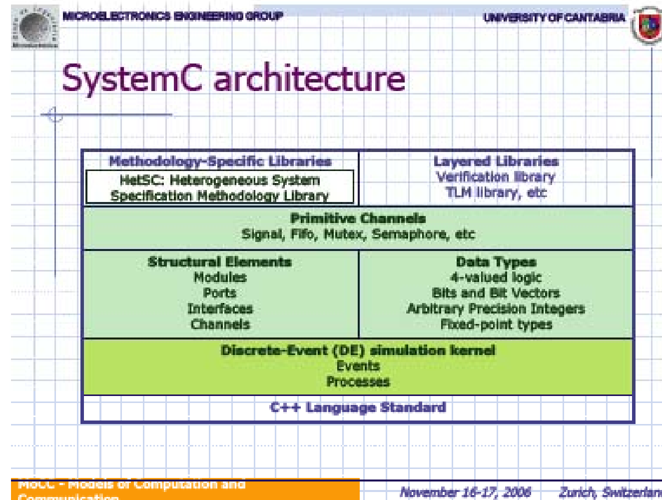
The translation of KPN requires some buffer management at run time. The reason is that, if buffers are too small, then it can lead to artificial deadlocks. The solution is dynamic and due to Tom Parks [PhD Berkeley, 1995]: buffer sizes are increased dynamically each time a deadlock occurs. But it does not always obey the Kahn's semantics!

Marc Geilen has proposed an improved KPN scheduler that is correct for every *bounded* and *effective* KPN: schedule enabled processes in a fair way until a local deadlock occurs, then resolve the deadlock by increasing the smallest full buffer.

Marc Geilen has also proposed RPN as an extension of KPN to integrate reactive behavior, but at the cost of lost determinacy, and Scenario-Aware Data Flow (SADF) to support dynamic reconfiguration in order to design streaming applications (e.g., MPEG4).

Eugenio Villar: SystemC as an heterogeneous system specification language

Challenges: algorithmic heterogeneity (DSP, control, protocol stack, multimedia…), constraint heterogeneity (RT, reliability, resources, performance…), resource heterogeneity (ASIC, FPGA, GPP…), and methodological heterogeneity (languages and tools).



The problem is how to model several heterogeneous MoCCs with SystemC: from bottom to top, we have a Discrete Event simulation kernel (DE), a Discrete Time MoCC (DT), a Clocked Synchronous MoCC (CS), a Synchronous Reactive MoCC (SR), and then untimed MoCCs: (Kahn) Process Networks, SDF, CSP…

Which are the MoCCs that can be abstracted from the DE MoCC? The strict answer is that only strict-timed MoCCs can (strict time means that the logical time is tied to some physical time, e.g, microseconds).

How to represent untimed events onto the DE MoCC? It requires the breaking of the order relationship between δ-cycles (the lowest level loop in the SystemC scheduler).

The DE MoCC was chosen because it can model any algorithm running on any computer, and it is efficient since it can hide unnecessary details.

- Horizontal heterogeneity: ability to combine several MoCCs in the same specification.
- Vertical heterogeneity: ability to transform one MoCC into another while preserving the equivalence.

Link to implementation: SW synthesis by substitution of the simulation kernel by the equivalent RTOS functions, and HW synthesis by a new generation of behavioral synthesis from C.

Future work will involve the formal foundation of HetSC based on ForSyDe.

Lothar Thiele: Modular performance analysis

How can MoCs be classified and compared?

Models of Computation

How can we classify and compare them?

stepwise refinement
concurrency
hierarchy
incremental design
beauty
modular
safe
simple
accuracy
expressive
tools
formal
compositional
easy to use
efficient implementation
executable
scope
scalable

ETH Swiss Federal Institute of Technology
Computer Engineering and Networks Laboratory TIK

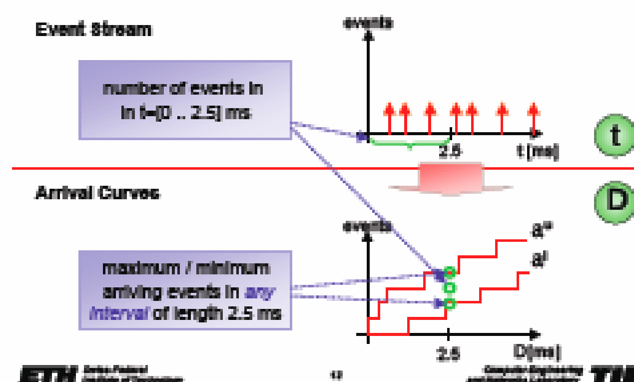
Why is it difficult? Because many aspects cannot be quantified, and MoCs cover different scenarios. The goal here is to compare models and methods that analyze the timing properties of distributed systems.

Obligatory part is to define a set of benchmark examples that cover the intersection between MAST, TIMES, SymTA, and MPA. The optional part is to extend it to MPA.

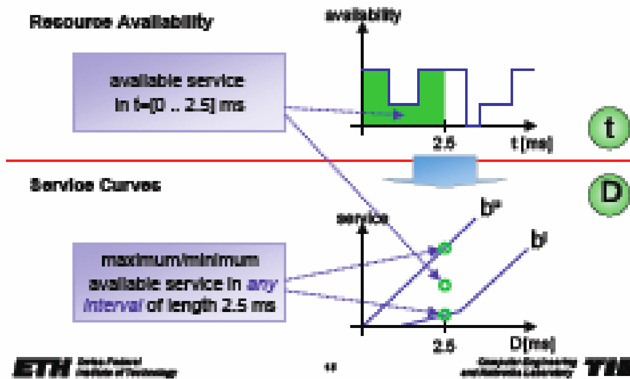
SymTA is based on classical RT analysis (periods, jitter…), with simplified relations and adaptors to achieve modularity.

TIMES is based on timed automata (and the Uppaal model-checking tool).

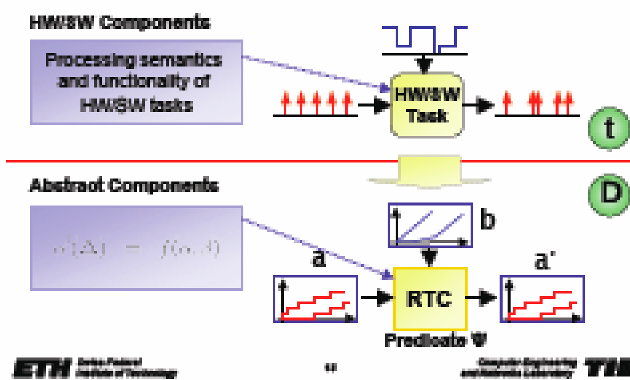
MPA is Modular Performance Analysis: it uses an abstract stream model where, for each time bound d, you have an interval of the number of events that occurred during the interval [0,d]. Then various models can be defined: periodic stream, periodic with jitter, periodic with burst…



There is also an abstract process model, and an abstract service model, which allows the definition of various kinds of services (full, bounded delay, TDMA, periodic...), and so various kinds of resources (memory, computation, communication, energy…).



The processing model mixes HW and SW:



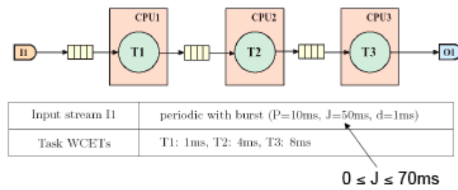
Finally, there are scheduling and arbitration components, which allow the modeling of various resource usages (EDF, TDMA, fixed priority, GPS) and various processing semantics (greedy processing, greedy shaper, blocking…).

Being actually able to perform the timing analyses requires some abstraction (because they are timed systems).

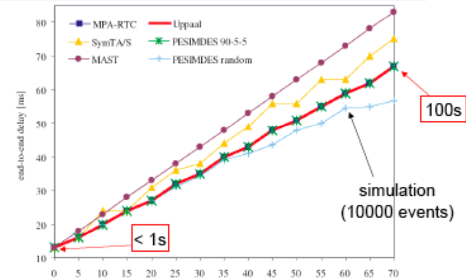
For a simple benchmark (three CPU put in sequence that process some periodic data with burst), obtaining a precise timing analysis is already complex:

Benchmark 1

- Pay Bursts Only Once



Benchmark 1



More complex benchmarks have also been studied: with cyclic dependencies and with a feedback loop.

There are two conclusions:

1. In models for timed systems, abstraction matters.
2. Knowledge about MoCCs, which also talks about resource usage, is far less understood (this is based on the actual observation of what the timing analysis tool does on benchmarks).

Michael Gonzalez-Harbour: MAST: a timing behavior model for embedded systems design

Motivation: the latest schedulability analysis techniques are difficult to apply by hand.

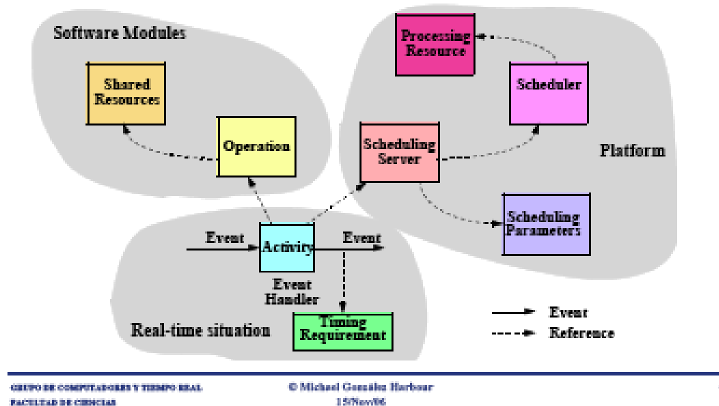
There is a need for a rich and flexible model of the real-time system:

- distributed, multiprocessor, or single processor
- composable software modules
- separation of architecture, platform, and software modules
- rich set of event-driven patterns

The goal is to develop a model for describing the timing behavior of event-driven distributed systems. It should be open (ability to evolve), and should be supported by a set of tools for schedulability analysis, synchronization blocking computation, discrete-event simulation in soft real-time, priority assignment, and sensitivity analysis.

The RT model uses the notion of *transaction*:

2. Real-Time Model: Overview



The case study is a teleoperated robot: its model includes the processing resources (schedulers, drivers, and timers), the scheduling servers, logical operations (e.g., a lock on a shared resource), the transactions (e.g., a distributed one), the external events (e.g., periodic, burst, sporadic…), the event handlers (e.g., merge, join, fork, branch, rate divisor…), and finally the timing requirements.

This model is then parameterized with the elementary WCET of the bottom elements, because these depend on the actual execution HW platform. The deployment tool instantiates the parameterized component models, provides the platform model, integrates them with the RT model, and then performs the schedulability analysis. In case of success it gives the schedulability margin, in case of failure it gives the needed speedup on the WCET that would be necessary.

The MAST environment includes Ada and UML as model building tools. Future work involves a graphical editor, the simulator, and missing tools (multiple event analysis, full support for EDF…).

Alain Girault: Adaptor synthesis for real-time components

The goal is to propose a lightweight component model for real-time systems. Components are assumed to be *black-box* (e.g., from the shelf components). Because of that reason, the components may have non-matching behaviors when they are assembled. The goal of the proposed method is to generate adaptors to make the component communicate harmoniously, i.e., without deadlock and with bounded buffers.

A component has several input ports, several output ports, and a single clock port. At instantiation, the input ports will be connected to other component's output ports or to the environment, output ports will be connected to other component's input ports or to the environment, and the clock port will be connected to a periodic clock.

When the clock is 0, the component is disabled and it can only let the time elapse. When the clock is 1, the component is enabled and it can also perform an action.

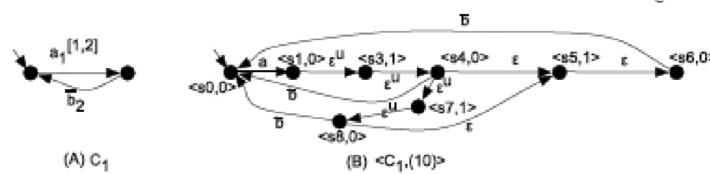
The behavior of each component is abstracted as an LTS specifying its communication protocol: this LTS is labeled with input port readings (e.g., ‘ $\overline{a}$’) or output port writing (e.g., ‘b’). Each such action also has a latency (an integer), a duration (an integer interval) and a controllability tag (either ‘u’ for uncontrollable or ‘’ for controllable).

The proposed method first involves the computation of the semantics LTS of the component, by expanding each action according to a pattern:

Component semantics

Latency is a QoS requirement:

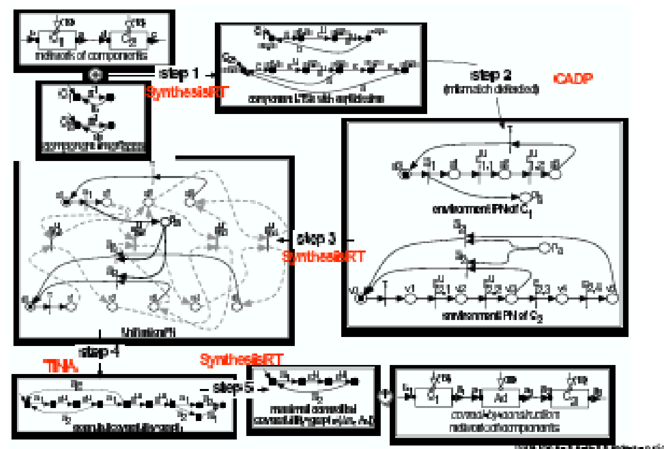
Duration is a constraint of the execution platform:



Each state is labelled with the global time instant modulo the periodic clock’s length (e.g., $\langle s_4, 0\rangle$ and $\langle s_5, 1\rangle$)

The method is implemented inside a tool chain using CADP to detect mismatching behaviors, SynthesisRT to compute the semantics LTS, to produce the expected environment Petri Net (PN) of each component (i.e., how the component expects its own environment to behave), to unify all those environment PNs into a single PN, and to compute the controllable coverability graph of this unification PN, and TINA to compute the extended coverability graph:

Tool chain



The method guarantees that the produced adaptor is deadlock-free, bounded, and correct. Correct means that the behavior of an assembly with the adaptor is a subset of the behavior of that assembly without the adaptor.

The obtained adaptor can, for instance, reorder events emitted by two components so that they communicate harmoniously. For instance, it component A first produces an event on port a (with action ‘ $\overline{a}$’) while the component B first produces an event on port b (with action ‘ $\overline{b}$’),

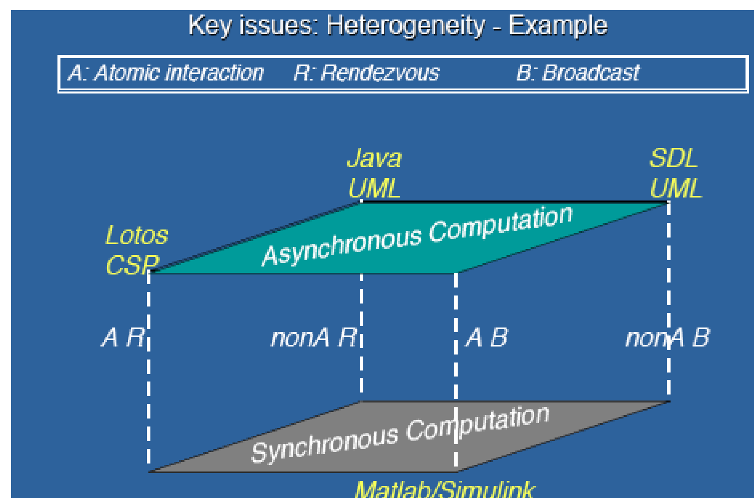
then the adaptor will non-deterministically accept either ‘a’ or ‘b’. Without an adaptor, the synchronous composition of those two components causes a deadlock. Another situation is when the adaptor buffers input event before emitting them.

Open problems include the incrementality issue: assume you have 3 components A, B, and C, is it equivalent to build first an adaptor X for (A,B) and then a second adaptor Y for ((A,X,B),C) or to build a single adaptor Z for (A,B,C)?

Joseph Sifakis: Component-based construction of heterogeneous real-time systems in BIP

MoCCs are related to component-based construction for heterogeneous systems. Fundamental notions are component and composition. Low-level theories are process algebra and automata. Many SW frameworks, coordination languages, and system modeling languages and tools exist: Corba, Javabeans, .net, Linda, Eclipse, Ptolemy, SystemC, Simulink/Stateflow, Metropolis… But they lack a model to address interactions, time, and resources.

Heterogeneity of interaction: atomic or not, rendezvous or broadcast, binary or n-ary. Heterogeneity of execution: synchronous, asynchronous, or combinations. Heterogeneity of granularity. For instance:



Framework for component-based construction: build a component C satisfying a given property P, from a set of components Ci modeling behaviors, and a set of glue operators on components that do not add more behavior. The semantics of Ci is its behavior.

The first problem is to find a suitable set of glue operators, with *incrementality*. In process algebra, the parallel composition operator is associative, so incrementality is achieved. But here, we do not know if the glue operators are associative. This raises the issue of decomposition. Is the following assembly:

(gl, C1, …, Cn)

equivalent to:

(gl1, C1, (gl2, C2, …, Cn))

It also raises the issue of flattening. Is the following assembly:

(gl12, (gl1, C1, C1’), (gl2, C2, C2’))

equivalent to:

(gl, C1, C1’, C2, C2’)

The second problem is *compositionality*: how to build correct systems from correct components? A related notion is *composability*, which says that integrated components preserve essential properties. Formally, assume:

(gl, C1, …, Cn) sat P

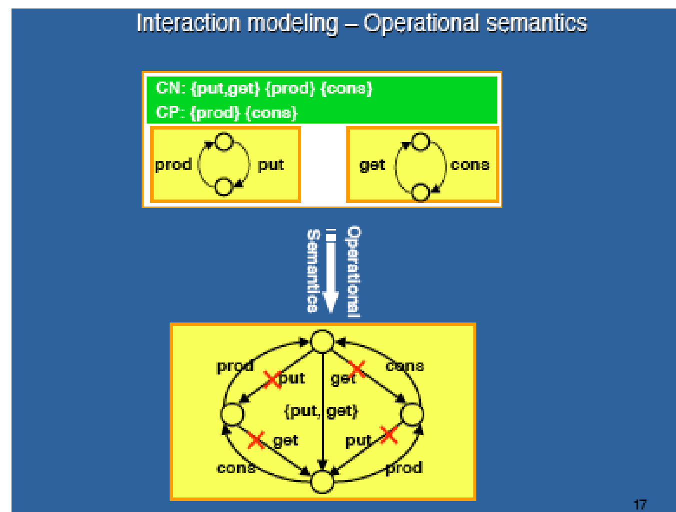
(gl’, C1, …, Cn) sat P’

Then, what must be the properties of the + operation such that:

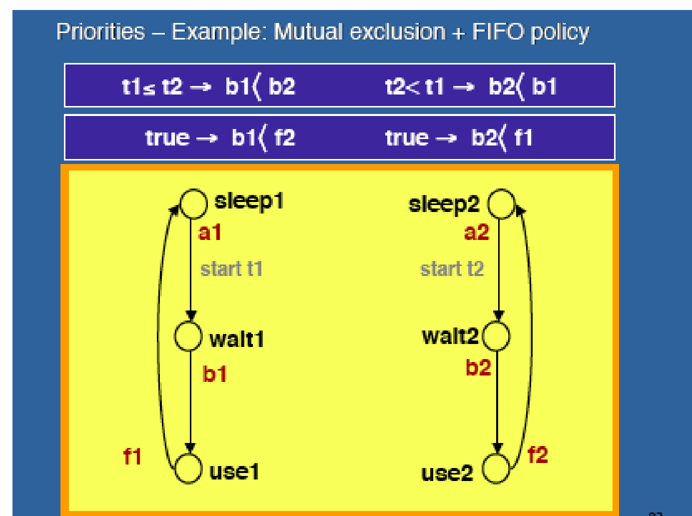
(gl+gl’, C1, …, Cn) sat (P and P’)

The BIP framework: This is a layered model: Behavior, Interaction (which model communication and cooperation), Priority (which model conflict resolution).

Each component has its own interaction rules, and the composition operations also have their own interaction rules (e.g., interaction should be maximal). This results in some pruning of transitions in the parallel composition of the components’ behavior. For instance:



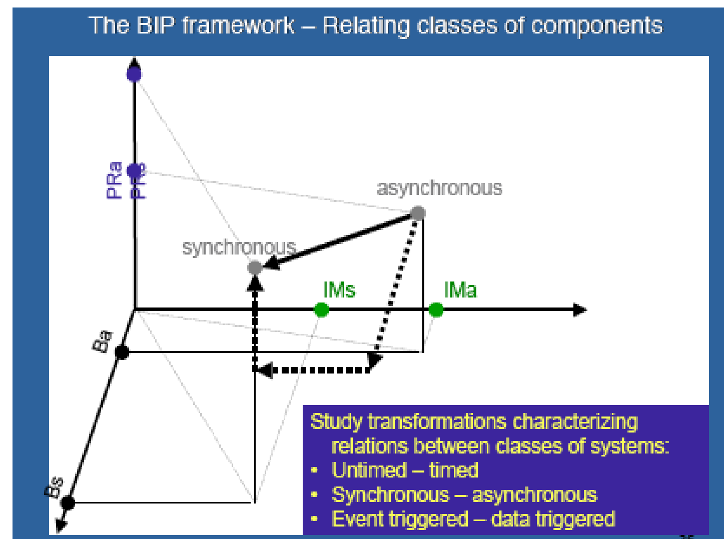
Priorities are used to restrict components interactions. For instance, mutual exclusion can be achieved with priorities:



BIP can be either compiled to C++ (running on the BIP platform, model-checked on the IF platform), or translated to the Think implementation of Fractale. In the BIP execution

platform, each component runs in its own thread. The thread assignment preserves the priority semantics.

Thanks to the orthogonality of the three concepts (BIP), the transformation of a system in some MoCC into its equivalent counterpart in another MoCC can be understood as a sequence of three transformations along the three axes B, I, and P:



BIP also allows you to understand and compare different programming languages, by decomposing all their features in terms of B, I, and P. This seems to be very useful for evaluating the expressive power of a programming language or for comparing two programming languages.

Christoph Kirsch: Shaping process semantics (and the JAviator: a flying MoCC laboratory)

The idea is to apply traffic shaping technology known in the networking community to SW processes. SW processes invoke system calls to access resources, perform I/O operations… Such system calls can be seen as packets in a network, hence the analogy.

Process shaping involves for instance prioritizing the processes. It can be applied to processes, to disk accesses, to network accesses… It is based on tokens that are generated and circulated among the processes that compete for some resource.

Process shaping will complement, not replace, the notion of serving processes ASAP. It is claimed that faster processors and lower kernel latency, in analogy to shorter packet transmission times, will make process shaping more effective.

Experiments on two web servers running on a single Linux machine shows that, without process shaping, the two web servers increase the load on the system until the maximal load is reached (because of the network bandwidth capacity), then there is a drop out and one server wins over the other. With process shaping, both servers share almost equally the resource and there is no drop out in performances.

The interesting question is how to find automatically the best “shape”.

Unrelated to the first part of the talk, the JAviator project is a helicopter whose SW is entirely written in Java. The helicopter itself is a "quatre-doigts", i.e., a rigid cross with four engines that control each one rotor located at the end of the cross. The rotating speed of each engine allows the control of the helicopter (roll, pitch, elevation, and vertical speed).

Tom Henzinger: Some thoughts on component models

An actor has a limited state space, and dependencies that are bounded in time and in space (because the memory is bounded). However, in most actor models, the dependencies are static, while to model systems with mobility, dynamic dependencies would be useful (like in the pi-calculus).

An ideal component model would be such that value dependencies are the *only* computation and communication primitives, component dependencies are *bounded* in both time and space, but *dynamic* in both time and space, components can be *aggregated* in both time and space, and components have *multiple* behaviors.

Reactive Modules [Alur & Henzinger 1996] are almost that, except that the component dependencies are dynamic only in time, not in space.

Florence Maraninchi: 42: The question of components, embedded systems, and everything

Component-based design actually exists in HW: IP blocks thanks to the sequential Boolean abstraction of the electric behavior. In SW it also exists, at least in non concurrent framework: the central notion is encapsulation.

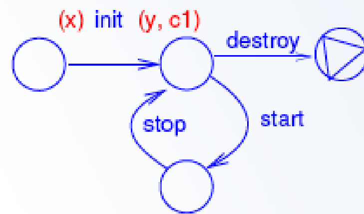
Synchronous languages are very good tools for the modular design and analysis of embedded systems (HW/SW). They are also good candidates for a component-based framework: The declarative style is close to the style of ADLs; asynchrony may be encoded in a synchronous formalism; code generation is well understood; time and concurrency are dealt with in a very precise way; automatic abstractions and analyses are possible; execution platforms and physical environments can be modeled.

The goal behind 42 is to isolate the main ideas of a component-based framework, focusing on encapsulation and component protocols, conditions for an assembly, how to build atomicity, and hierarchy. The basic principles are: behaviors are in the components, oriented connections are only wires, a *director* is added to each assembly to characterize the MoCC (it manages the reactions of the components, the semantics of the wires, their memory…, and what remains visible outside the assembly). We want to write the director as a small program in term of basic operations.

The protocol defines how components can be used and check that an assembly is correct. It has an automaton structure specifying the correct sequence of method calls (used for control inputs), with several accepting states specifying what sequences of activations are complete (w.r.t. the atomicity of the component), and each transition is labeled by a control input indicating what data inputs it requires and what data or control outputs it produces:

42: Definition

42 Component Protocols, General Definition



control inputs:
init, start, stop,
destroy

data input : x
data output : y
control output : c1

FM, TB (Verimag/INPG) 42 November 17, 2006 14 / 33

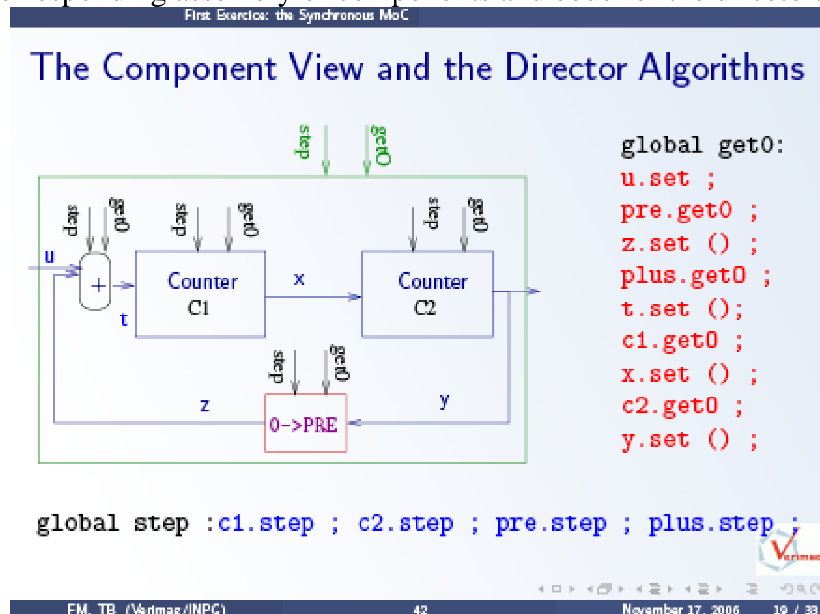
For instance, consider the synchronous MoCC. The running example is a double integrator:

```
node DoubleIntegr (i : int) returns (o : int) ;
var x, y, z : int ;
let
        x = Integr (i + (0->pre y)) ;
        y = Integr (x) ;
        o = y ;
tel.
node Integr (i : int) returns (o : int) ;
let
        o = i -> pre(o) + i ;
tel.
```

Below is the corresponding assembly of components and code for the director:

First Exercice: the Synchronous MoC

The Component View and the Director Algorithms



FM, TB (Verimag/INPG) 42 November 17, 2006 19 / 33

To be able to implement pure synchrony in a component-based manner, we need to distinguish between getO and step(s). If we get a piece of code with this interface, it can be

used as a black box in our component model. The director needs only setting the values of the wires and activating the components. The values on the wires are not meant to be persistent: they are used only during the global step. This is the essence of synchronous communication. The director can be deduced from the dataflow graph and the components' protocols (Lustre structural interpreter, electricity in synchronous circuits!)

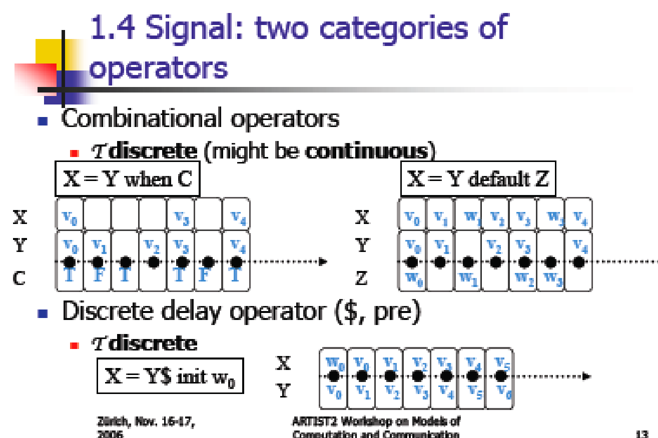
Other MoCCs can be defined in the same way of course. Then, to produce the code for the corresponding director, what is needed is the processes' codes, the connections, and a global indication, for instance "a global step should be exactly one step OF EACH component" for the synchronous MoCC, or "a global step should be one step of process 1 plus its consequences XOR one step of process 2 plus its consequences" for the asynchronous MoCC.

Thierry Gautier: Polychronous MoCC for open systems

The time domain has a partial order (infinite semi-lattice). Each signal is totally ordered. Any family of sets has an upper bound (this will allow the computation of the default operator). A process is then a set of traces.

In the endochronous time, any clock is a functional sampling of the global clock. In the polychronous time, cloks are defined by general relations (instead of functions). In Esterel, inputs are exochronous (they may occur at any time) and outputs are endochronous (determinism). In Lustre, inputs and outputs are endochronous. In Signal, inputs and outputs are endochronous and/or exochronous; it is a mixed relationship.

So, Signal has two categories of temporal operators: combinational (when and default) and discrete delay ($ and pre). Clocks are then nested inside a clock hierarchy:



Signal also has assume/guarantee specifications, through the usage of relations between clocks (e.g, A => G, where A is the assume part and G the guarantee part). Relations can be static, dynamic, state relations (obtained from automata translation), or conjunctions of relations (when operator). For instance, a component with an always increasing numerical input I and an output O that is never true at two consecutive instants will have:

A: I >= $I

G: not (O and not $O)

These assume/guarantee properties are combined when two components are put together. For instance, if we have two components P1 and P2:

P1 |— A1 • G1

P2 |— A2 • G2

Then the parallel combination of P1 and P2 is such that:

P1 | P2 |— (A1 • A2) • (A1 • G1 • A2 • G2)

Discussion

This section is a loose transcript of the discussion that took place at the end of the workshop. Paul Caspi opened the discussion with a list of unanswered questions and sub-questions (see the titles and sub-titles in boldface below). Some of them were addressed while others were left open at the end of the discussion.

Are MoCCs needed? How many of them?

Is a taxonomy needed and how to get it?

Is continuous time a MoCC and do we need it?

What is expressiveness?

Edward Lee (EAL) argues that yes: constraints must be imposed on the interactions between components, and a MoCC is exactly that. But you pay some price for it.

Ton Henzinger (TH) argues against the need for heterogeneity in a MoCC. Haskell should be sufficient for everybody.

Paul Caspi (PC) claims that heterogeneity arises naturally in embedded systems (signal processing, electricity, discrete time). So it's not just Haskell.

Joseph Sifakis (JS) shows 3 slides about the concept of MoCC. The problem comes from the fact that the denotational semantics model does not encompass any reasonable concept of MoCC. It emerges only in models that explicitly talk about components and interactions. Concerning Ptolemy (which can be viewed as an implementation of the tagged signal model), how are directors related to the denotational semantics. An approach for a common understanding would be to agree on a concept for components and their compositions, then define tagged sequence machines for this model, and then study how the two can be related.

EAL argues that this is the "favorite MoCC" view: choose your favorite MoCC and encode everything else in it. The tagged signal model cannot be because it is too abstract, so it cannot be executed. It has to be specialized (e.g., by SDF).

JS gives some meta-thoughts. A well-known trend in science is that when you change the level of granularity, some new phenomena/properties/concept emerges.

EAL claims that you do not want to change the semantics of a MoCC, never. Everybody seems to agree.

JS want separation of concerns, hence the separation of the behavior with the rest (e.g., interactions and priorities in BIP). This complies with the black box approach. Then, the glue around behaviors should be purely state-less functions.

EAL shows three slides: you can have several abstract semantics, each finer than the previous one (at each refinement some MoCCs are ruled out), and at the end (i.e., the finest one) you have a concrete semantics with full abstraction, and this would be a MoCC.

TH says that some semantics are too abstract to capture key features, while others are not, so he thinks that this view of nested semantics is not the good one. Actually, some abstract semantics (as defined by EAL) are not real semantics, and EAL agrees with that (hence he calls them abstract semantics and not semantics).

What are transducers between MoCCs? How do MoCCs communicate?

EAL says that abstract semantics can be used for just that. You have a tree of abstract semantics, and if you want to compose to actors that comply with different semantics, you have to choose the director that lies at the root of the subtree that encompasses those two semantics. This scheme does not require adaptors, but it requires a tree of abstract semantics and an implementation of each of them.

Eugenio Villar (EV) asks how to include an SR actor inside a CT domain.

EAL gives the example of SDL (Turing complete) and Lustre (not Turing complete). Lothar Thiele says that there are two ways to do it (methods A and B), and he raises the question of the usefulness of each method compared with the other. PC says that one might not give the same result as the other.

How to describe a MoCC?

With a denotational or operational semantics?

Predefined or built from primitives?

With layers (actors and directors) like, e.g,, Rialto and BIP?

What about modeling, simulation, execution?

This part was not addressed during the discussion.

How to manage the interactions between components: With glue, Adaptors? Interfaces? Hierarchy?

EAL claims that hierarchy is needed, and he has plenty of examples to support his claim. FM

What tools are required?

This part was not addressed during the discussion.

How to specify non-functional properties in components?

EAL claims that some non-functional properties are in fact functional for some people (e.g., timing for reactive systems). The same goes for energy consumption according to LT. For this, the domain of functions must be defined: does it include time (in that case temporal

properties are functional), does it include power (in that case power consumption properties are functional), and so on. What is outside the domain of functions is therefore non-functional.