42: Programmable Models of Computation for a Component-Based Approach to Heterogeneous Embedded Systems

Florence Maraninchi, Tayeb Bouhadiba

▶ To cite this version:

Florence Maraninchi, Tayeb Bouhadiba. 42: Programmable Models of Computation for a Component-Based Approach to Heterogeneous Embedded Systems. 6th international conference on Generative programming and component engineering, Oct 2007, Salzburg, Austria. pp.53 - 62, 10.1145/1289971.1289981 . hal-00294153

HAL Id: hal-00294153
https://hal.science/hal-00294153

Submitted on 8 Jul 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

42: Programmable Models of Computation for a Component-Based Approach to Heterogeneous Embedded Systems

Florence Maraninchi and Tayeb Bouhadiba
VERIMAG/INPGrenoble, 2 avenue de Vignate, F38610 GIERES
Florence.Maraninchi@imag.fr, Tayeb.Bouhadiba@imag.fr

Abstract

Every notion of a *component* for the development of embedded systems has to take *heterogeneity* into account: components may be hardware or software or OS, synchronous or asynchronous, deterministic or not, detailed w.r.t. time or not, detailed w.r.t. data or not, etc. A lot of approaches, following Ptolemy, propose to define several "Models of Computation and Communication" (MoCCs) to deal with heterogeneity, and a framework in which they can be combined hierarchically.

This paper presents the very first design of a component model for embedded systems called 42. We aim at expressing fine-grain timing aspects and several types of concurrency as MoCCs, but we require that all the MoCCs be "programmed" in terms of more basic primitives. 42 is meant to be an abstract description level, intended to be translated into an existing language (e.g., Lustre) for execution and property validation purposes.

Keywords Heterogeneous embedded systems, component-based design, semantics, models-of-computation

1. Introduction

1.1 Component-Based Approaches for Heterogeneous Embedded Systems

The notion of a *component* for embedded systems has been discussed for some years now, and there are a lot of proposals. The main motivations are the following: as time-to-market decreases, it becomes unavoidable to *reuse* a lot of previous work when designing new systems. Reusing parts of a previous system requires that these parts be properly defined as *components*, equipped with some form of a *specification* (informal or formal). The *specification* groups all information needed for using the component, without knowing in details how it is built. This includes both *functional* and *non functional* aspects, like timing performances, or energy consumption.

There seem to be a wide agreement on the fact that the main difficulty is due to the intrinsic *heterogeneity* of embedded systems, which has numerous causes. We list the main ones below, including *intrinsic* heterogeneity of the components that exist in the final product, and *design* heterogeneity that occurs during the design flow:

- A typical embedded system may be built from hardware and software components (for the latter, both at the operating system level, and at the application level);
- It is usually made of *concurrent* objects, but the concurrency model varies from pure synchrony (in mono-clock synchronous circuits for instance) to pure asynchrony (representing the behaviors of a multi-computer system). Globally-Asynchronous-Locally-Synchronous (GALS) systems are an interesting intermediate case.
- The description of an embedded system may range from the high levels of abstraction where the timing and the structure of the data is not detailed, to the low levels of abstraction often called "cycle-accurate, data-accurate". The emerging "*Transaction-Level-Modeling*" paradigm [14] is a very good example of a component framework for embedded systems, allowing to develop virtual prototypes of systems-on-a-chip at various levels of abstraction.
- At the higher levels of description, components may also be non-deterministic, because they are known as specifications only (*contracts* for instance, in the sense of [25]), not as detailed descriptions yet.
- An embedded system that is the implementation of some control law benefits a lot from the possibility of describing the physical *environment* as a component that lives in parallel of its controller.

Despite these causes of heterogeneity, the system-level design of embedded systems requires that we be able to reason precisely on *timing*, *atomicity* and *concurrency*.

1.2 Programming vs Architecture-Description Languages

Some people also present as a cause of heterogeneity the fact that several *styles* are used to describe embedded systems, ranging from pure imperative languages or explicit automata (Statecharts, Stateflow in Simulink, UML activity diagrams) to pure dataflow (Simulink[1], Lustre/SCADE [15], Signal [21]), with notable combinations like mode-automata [23] or the joint use of Simulink and Stateflow. In this paper, we would like to concentrate on semantical notions, not on the multi-paradigm programming problems. We will give a definition of a component that is independent of the *programming language*. By *programming language* we mean the language that is used to describe the detailed behavior of the individual components. In a component-based framework, there is usually an *architecture description language*, which has little to do with the programming language. It often has a dataflow style.

[copyright notice will appear here]

[1] Simulink and Stateflow are trademarks of The MathWorks

1.3 Ptolemy

Since 42 is inspired by Ptolemy [11], we recall here the main characteristics of this component framework. Components are actors in the sense of [19]; it is possible to form a new actor by putting a set of actors together, with connections between them, and a local *director* that defines how they behave together and what the connections mean. The director is the implementation of a so-called *Model of Computation and Communication* (MoCC). Heterogeneous designs are obtained by using distinct directors, depending on the position in the hierarchy of components. The available MoCCs are formalized independently of each other.

In Ptolemy, the notion of MoCC is somewhat extreme: given a picture made of boxes and arrows, it is possible to consider it, either as a dataflow diagram (with subdivisions of this MoCC into: synchronous dataflow, synchronous reactive, etc.), or as an automaton (the FSM domain), just by changing MoCCs. This means that even the interpretation of the architecture-description (ADL) part is left to the MoCC: the ADL, used to group components at a given level of the hierarchy, may be dataflow (in which case the components are implicitly in parallel), or given as an explicit automaton (in which case the components execute sequentially), or anything else that could be expressed in a new MoCC.

Another important aspect of Ptolemy is to allow the combination of discrete and continuous models in the same system description, which is really useful for embedded systems that may include digital and analog parts.

1.4 Component-Based Virtual Prototyping with the Synchronous Technology

42 is also inspired a lot by more than 10 years of experiments on using a dataflow synchronous language (Lustre [15] or Signal [21]) as a very expressive component-based executable modeling language.

The dataflow style makes it very natural to use the language as an architecture description language, and it is also a programming language for individual components. Lustre is used to specify safety properties (for automatic testing or model-checking) by means of so-called *observers* [16]. A notion of logical-time *contract* for a synchronous component has been proposed [22], using the same notion of observer. The Lustre toolbox offers verification tools that can be used to answer the classical questions about components: is a component detailed description a correct implementation of a contract?, does an assemblage of components make sense, based on the combination of their contracts?, etc.

The synchronous approach is adequate for modeling software and hardware. Since the work by Milner [26] in the early eighties, we also know that synchronous formalisms can be used to model asynchronous parallelism. In fact, the synchronous paradigm may be used to model all kinds of intermediate behaviors, between pure synchrony and pure asynchrony [12, 17].

The main idea is as follows: for modeling purposes, all the components are equipped with so-called *activation conditions* and they do nothing when this condition is false. These components are all composed in a synchronous way, the activations conditions being global additional inputs. If there is no constraint at all on these additional inputs, the synchronous composition adequately describes the pure asynchrony between the components. If the activations conditions are equal, the same composition describes pure synchrony. More complex conditions correspond to intermediate cases. In order to give executable models, such a modeling principle requires that components be equipped with activation conditions, and that the context in which they evolve be described as additional code that *generates* the activation conditions. This additional code is usually non-deterministic (which, again, has to be encoded into the deterministic synchronous formalisms by adding inputs).

1.5 Motivations for 42

For the moment we have managed to describe a wide variety of systems by encoding everything into a synchronous formalism, but the encoding may be heavy. There are at least three reasons for defining 42 (for the name, see [2]). First, it would save time if we could express the main principles used for system-level descriptions as abstract primitives, in some formalism that would then be automatically translated into a synchronous framework; this would remove the burden of detailing again and again the same kind of encodings. Second, a definition of heterogeneous component modeling at a higher level of abstraction than the synchronous encoding is becoming necessary to identify clearly the minimal set of notions that are really needed. Finally, the modeling principles we have used so far require a good knowledge of synchronous formalisms in order to be understood precisely; defining 42 should be a way of explaining these ideas in a more general setting.

1.6 Contributions and structure of the paper

42 is meant to be used as an abstract (but still *executable*) description level for describing the concurrent and timed behavior of heterogeneous embedded systems components. It concentrates on a precise modeling of time and concurrency for functional "system-level" descriptions of embedded systems, where the main and more serious errors appear. It is dedicated to *discrete* systems, and offers a support for the types of heterogeneity we have encountered in a large number of cases-studies.

The 42 formalism is intended to be implemented into an existing language like Lustre for simulation, automatic testing and property validation purposes. In a longer term perspective, we will consider the use of 42 as the new entry point for the model-driven implementation chain already developed around Lustre.

The paper is structured as follows: in Section 2 we define 42 informally; Section 3 shows how some known MoCCs can be described in 42; Section 4 defines some classical problems of component-based framework, in the context of 42; Section 5 is the semantics; Section 6 lists some related work, and summarizes the main choices that we made for 42; Section 7 is the conclusion.

2. Definition of 42

2.1 Components

Figure 1 shows a 42 component. A component is a black-box that has input and output data ports, and input and output control ports. The input control ports are used to ask it to perform one finite-execution computation *step*. Since there may be several input control ports, there may be several *entry-points* that toggle a step. A step corresponds to a terminating (non-necessarily deterministic) piece of code. A component has some internal memory. The input and output data ports are used to communicate data between the components. The output control ports will be used by the components to send information to the controller (see below).

Figure 2 is an example code for such a component. For each control input, the component executes a program that corresponds to its computation step. It should terminate in finite time. The memory `m` is initialized when the component is instantiated somewhere, is persistent across the successive activations of the component, and is common to the various activations. A component does not necessarily use all its data inputs for a given activation, and does not necessarily produce all the control and data outputs (see the specification part in section 2.3.1 below).

2.2 Connections and the Architecture Description language

Components are connected by directed *wires*. An input data port can be connected to an output data port of the same type (we will assume this is always true in the sequel). The control ports are

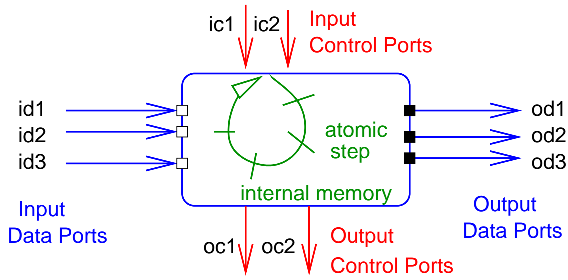

Figure 1. A 42 Component

```
Component C (
   control input ic1, ic2 : bool;
   data input id1, id2, id3 :int;
   data output od1, od2 ,od3 :int;
   control output oc1, oc2: bool) {
var m : some_type = some_init_value ;
  for ic1 do : {
    int cpt = 0 ; if (id1 < 0) id1 = -id1 ;
    while m >= 0 { m = m - id1 ; cpt ++ ; }
    od1 = cpt ; oc1 = (m == 0) ; m = m+cpt ;
  }
  for ic2 do : {   if (m ...) { ... }   }
}
```

Figure 2. Example code for a component

connected to the *controller*, not directly to each other. A wire does not mean a priori any synchronization, nor memorization.

A *system* is made of components connected by wires (the *architecture*) plus a *controller* that activates the components and decides what happens on the wires. The model is hierarchic: an architecture plus a controller form a new *component*. It exposes new input and output control ports, and new input and output data ports.

Figure 3 is an example connection. The components `A, B, C, D` are connected with the wires named `a, b, c, d, e, f`. Some of the data input and output ports of the subcomponents are connected to the input and output ports of the assemblage. All the components have input and output control ports (vertical arrows) implicitly connected to a *controller*.

2.3 The Controller

Once the components have been connected, we need to specify how they behave together. The controller is in charge of *translating* an activation request on one control input port of the encapsulated system (e.g., `xx`, also referred to as a *macro-step* in the sequel), into a sequence of activations of the subcomponents and data exchanges between them (also called *micro-steps*). To achieve this, the controller may use some temporary variables explicitly associated with the wires (because, if there is no connection between a port `p1` on a component `A`, and a port `p2` on a component `B`, the controller may not transmit directly the data produced on `A.p1` to `B.p2`). The memory associated with the wires serves only as temporary storage, to build a macro-step, because not all MoCCs describe situations in which the values produced by a component are immediately consumed by another one. Hence the lifetime of the wires' memory is limited to the macro-step. If there is a need for storing information between two macro-steps, then there should be a component behaving as a memory.

Figure 4 is an example code for the controller, in some simple imperative style. The choice of the temporary memory associated with the wires, and a piece of code for each global control input port like `xx`, constitute a particular MoCC.

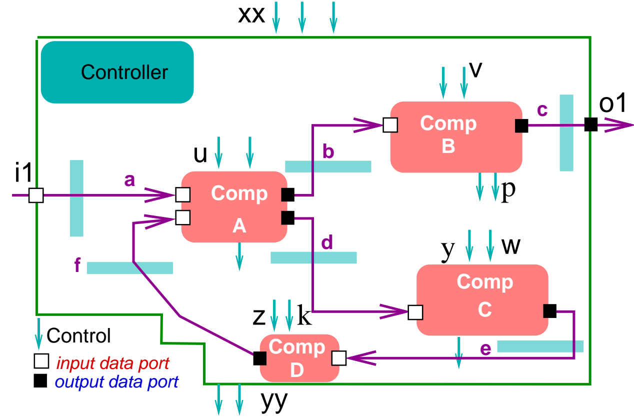

Figure 3. Connecting Components, an Example

```
Controller is
var m : bool = true ;
for xx do : {
  m_a, m_b, m_c: FIFO(1,int); m_d, m_e, m_f: FIFO(4,int);
  if (m) {
     m_a.put ; // reads i1
     m_a.get ; D.z ; m_f.put ; m_f.get ;
     A.u; m_b.put; m_d.put; m_b.get; B.v; m = m or p ;
     m_c.put ;  m_c.get ; // defines  o1
     m_d.get ; C.w ; m_e.put ;  C.y ; m_e.put ;
     m_e.get ; D.k ; m_e.get ; D.k ;  m = ! m ;
  } else { ... }
  yy = true ;
}
```

Figure 4. Example controller code

The controller associates a bounded FIFO with each wire. On Fig. 3, `a, b, c` are associated with the one-place `int` FIFOs `m_a, m_b, m_c`, while the wires `d, e, f` are associated with the 4-places `int` FIFOs `m_d, m_e, m_f`. A FIFO `M` offers three methods: `M.get` (resp. `M.put`) gets a value in `M` (resp. the producer port connected to the wire) and puts it into the consumer port connected to the wire (resp. `M`); `M.init` initializes `M` to an empty FIFO. It is the responsibility of the controller to avoid writing in a FIFO when it is full, or reading from an empty FIFO.

The programs of the controller may activate the individual components, through their control input ports (e.g., `A.u, B.v`). They may copy the data outputs of the components into the wires, or copy the wires into the data inputs. The program of the controller may also copy the control outputs of the individual components, into some memory local to the controller (`m`), and whose life span may exceed the reaction to `xx` (inter-macro-step memorization).

On the example code, the controller executes `D.z` without providing an input on the wire `m_e`. This is because a component does not necessarily need all of its inputs (resp. produce all of its outputs) at all times (see below). `m_e` receives 2 values before they are consumed by `D`.

2.3.1 Specifying Components: Control Contracts

The example has shown that there is some need for a precise specification of the components, in particular for declaring which of the inputs are needed for each control input, and which of the data and control outputs are produced.

In 42, we adopt the notion of *protocol* widely used in object-oriented designs (see, for instance [33]). When specifying a class in an object-oriented framework, a protocol can be used to specify that method `m1` should always be called before method `m2`, unless method `m3` has been called at least twice. The idea in 42 is similar:

protocols will be used to specify complex sequential constraints between the control inputs and outputs by a finite state machine. It can be viewed as a *control contract* for a component, because it expresses how the component should be activated, but tells nothing on the values that it may accept or deliver. A 42 protocol is also very similar to the notion of *conditional dependency* that can be expressed between inputs and outputs in Signal; control ports correspond to *clocks*.

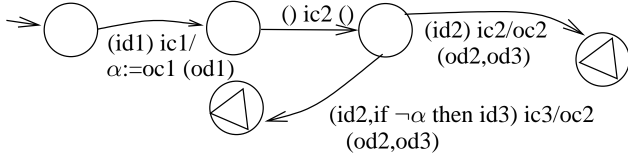

Figure 5. Example protocol for the component of Figure 1

Figure 5 shows an example protocol for the component C of Figures 1 and 2. It is an automaton with an initial state (pointed to by the little arrow) and several final states (denoted by a triangle). Each transition has a label of the form:
`(data req) control input / control outputs (data prod)`.
The `(data req)` part is a list of data input names, or conditional expressions, depending on the values emitted by the component previously, on its output control ports. This automaton expresses that a correct use of C by a controller is such that, within a macro-step executed by the controller, C is activated first with its control input `ic1`, which requires the input `id1`, and produces the control output `oc1` (α`:=oc1` means the value of the output at that point is stored for further references) and the data output `od1`. Then it should be activated through `ic2`, which requires and produces nothing. Then it should be activated with `ic3`, or `ic2`. `ic3` requires `id2` and, if `oc1` was false when it was produced as a result of the first activation, it also requires `id3`; it produces `oc2, od2, od3`. `ic2` requires `id2` only, and also produces `oc2, od2, od3`.

Any use of C by a controller that does not activate C in the correct order, or stops before the protocol of C has reached a final state, is an incorrect use of C. The controllers should be able to store the control outputs produced by components during one activation, in order to know what inputs to provide, and how to activate them later.

3. Some Known MoCCs Expressed in 42

3.1 Mono-Clock Synchronous Circuits or Programs

In a pure synchronous model of computation (for describing synchronous circuits for instance), the controller should be able to express the fact that, at each instant of a global clock, all the components of the circuit take their inputs, and compute their outputs. It may take some physical time to stabilize, but for non-cyclic circuits, it does stabilize.

Example Figures 6, 7 and 8 illustrate the componentization of a Lustre program. We chose Lustre because its graphical form (as used in the commercial tool SCADE) is very close to the diagrammatic view used in synchronous hardware design. The program of Fig. 6 is made of two instances of a basic integrator `Integr`. "`o = i -> pre(o) + i`" means: the output `o` is equal to the input `i` at the first instant, and then, forever, it is equal to `i` plus its previous value `pre(o)`. The two copies are connected in the node `DoubleIntegr`, with another addition operator. Figure 7 is a flat and graphical view of the node `DoubleIntegr`, where the two copies of `Integr` have been expanded.

Individual Components Figure 8 is the component view of the program, at the level of `DoubleIntegr`: the details of the two

```
node DoubleIntegr (i: int) returns (o: int) ;
var x, y : 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.
```

Figure 6. An Example Synchronous Program (in textual Lustre)

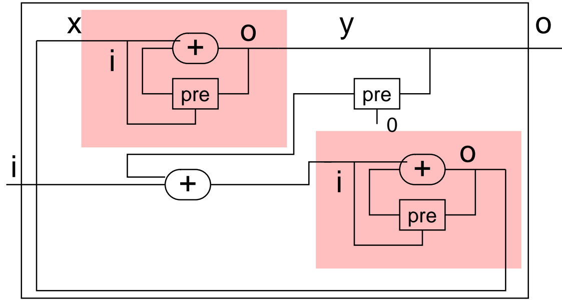

Figure 7. The same program in a graphical form

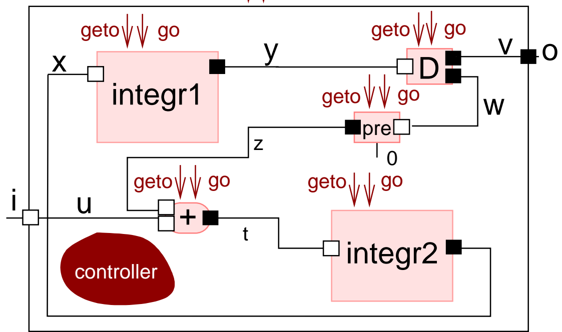

Figure 8. A component view

```
controller is { var u, v, w, x, y, z, t : FIFO(1,int)
 procedure stabilize is {        for geto do: {stabilize ;}
  pre.geto ; z.put ;             for go do: {
  z.get ; u.put // reads i              stabilize ;
  u.get ; plus.geto ; t.put ;           pre.go ;
  t.get ; integr2.geto ; x.put ;        integr1.go ;
  x.get ; integr1.geto ; y.put ;        D.go ;
  y.get ; D.geto ; v.put ; w.put ;      integr2.go ;
  v.get ; } // defines o                plus.go ;   }}
```

Figure 9. The programs of the controller

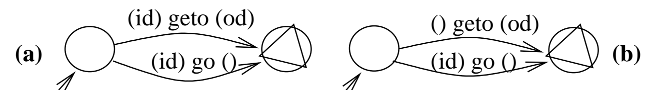

Figure 10. Protocols of the components

```
Component Pre ( control input geto,go : bool;
                data input id:int; data output od:int) {
var m : int = 0 ;
for geto do : { od = m ;} for go do : { m = id ; } }
```

Figure 11. Code of the PRE Component

```
//Component C is the main program. data input i, output o
initializations ;
while true {
    i.put (read()) ; // reads input into the i wire
    C.geto; write(o.get) ;  // writes the value produced
    C.go; }
```

Figure 12. Code of the main program

copies of Integr are hidden, they are considered as basic components as +, pre (a flip-flop, or elementary memory point) or the duplicator D.

In order to obtain the normal behavior of a Lustre program (or a synchronous circuit) with such a component view, the components should be designed in such a way that they offer two control inputs geto and go. When asked with geto, the component delivers its outputs, depending on the internal memory and its data inputs, but without changing internal states; go asks it to change internal states. In this simple case, there is no need for control outputs. The combinational components (the +, and the duplicator) have an empty go function.

The two Integr components could be further described as composite components, in terms of pre and plus. As an example, Figure 11 is the code of the pre component (for integers).

The controller Assembling the two copies of Integr, a pre operator, a plus operator and a duplicator gives a new component, which itself has two control inputs geto and go. The controller associates one-place buffers with all wires. The "programs" it plays when the global geto or go control inputs are activated are given in Figure 9, in some imperative style.

When the global component is activated with geto, the input i is supposed to be available. The controller asks each subcomponent to produce its outputs, according to the values that are available on its input wires. This is done in an order compatible with the partial order of data dependencies. The only component that can start is the pre component, because its output does not depend on its input. Then the plus can play, then Integr2, then Integr1, then the duplicator D. At the end of this sequence, all the wires have a value, the circuit has stabilized. Respecting the data dependencies means that each x.get is preceded by a x.put.

When the global component is activated with go, the input i is supposed to be available. The controller asks each subcomponent to change states according to its input. Notice that go should perform the stabilization first, for the pre component, and the memories hidden in the two Integr components, to have the correct inputs. The go activations of all the subcomponents are called, in any order.

These two programs are very similar to the code generated by the Lustre or SCADE compilers (except that: copying the values on the wires is not efficient and can be avoided in most cases; the stabilization phase is not performed twice). The component view of a Lustre program requires that there is indeed a possible computation order, meaning that each cycle in the data dependency graph is cut by a pre component, at each level of abstraction. On the contrary, if the sub-components are expanded (see Figure 7), it is sufficient to have all the cycles broken somewhere, but it could be inside the subcomponents.

The Protocols In the simple case described above, the protocols for the components are those of Figure 10. **(a)** describes the protocol of any component in which the output may depend on the input; **(b)** is the protocol of the pre component: od can be obtained without input.

Partial Computation of the Outputs In general, components have more than one input and one output. There are two choices: either we consider that all the outputs depend on all the inputs, and in this case we can apply the previous scheme. Or we can accept more complex designs, in which the outputs do not necessarily depend on all the inputs. In this case, each component has to specify the dependency between its outputs and its inputs, and each component has to be equipped with a go activation, and one geto activation per data output. The control contracts we presented in section 2.3.1 are perfectly adequate to express the dependency between the inputs and outputs. A transition (id1) geto1 / ... (o1,...) in such a protocol means that only the data input id1 is required for the component to be activated with geto1, and it produces o1.

The controller can then ask the components to produce specific outputs, not all at a time, and interleave the computations of the outputs of the subcomponents.

Comments This example shows that the expressive power of the controllers in 42 is sufficient to express pure synchrony, which hides a fix-point computation (the stabilization phase). It is easy to componentize Lustre or a diagrammatic description of synchronous circuits, and the model is exactly the same at all levels of abstraction. The code produced for a main program corresponding to a component C is given Figure 12. The protocols of the subcomponents, plus the architecture that gives the data dependencies, are sufficient to generate the controller: we only need to find a partial order for the computation of the outputs in the geto function.

The simple example extends to conditional dependencies between outputs and inputs, as shown above, and also to multi-cycle synchronous programs (in which a subprogram should be run at speed s1, and another at speed s2 much slower than s1). The principle of the component view can be used for separate compilation of Lustre/SCADE programs, with some optimizations in memory management.

3.2 Asynchronous Processes with Shared Memory

Example The example of Figure 13 shows how to model asynchronous systems in 42. We consider two processes P_1 and P_2 (there could be n) running on the same processor, and accessing a shared memory. We model the two processes P_1, P_2 and the memory M as components, and we encapsulate them with a controller that mimics the behavior of a non-preemptive and nondeterministic scheduler. The controller also takes care of the synchronous communication between the processor and the memory.

Each process may read (resp. write) something in memory (connections named rd1 and rd2, resp. wd1, wd2). This needs an address (connections addr1, addr2). The two processes are connected to the same ports of the memory, which implies Mux components. Similarly, the output port of the memory is connected to the two input ports of the processes, which needs a Dec component.

Individual components A process component can be made with any imperative code, encapsulated in such a way that it exports the data ports for the communication with the memory, and five control ports: getwish asks the process whether its next move will be a memory access (write or read) or an internal move, and whether it will yield at the end of this operation; the process answers with Wsh and yield; getAddr asks the process to emit a memory address on its address data port; op asks the process to execute one of its atomic operations. To model preemptive schedulers, the code of the process should yield after each piece of code which is guaranteed to be atomic (non interruptible) by the hardware, or by languages features like the synchronized keyword in Java. To model SystemC, whose scheduler is non-preemptive, the code of the processes should yield exactly when the SystemC code yields.

The memory has two input data ports (address and value to write) and an output data port (value read). It has two control inputs (read and write) and no control outputs.

The Mux and dec components are used to route data, they are controlled by control inputs m1, m2, d1, d2 to choose the route.

The controller The controller acts as a scheduler (see Figure 16). It selects a process randomly, then asks the process to perform operations until it yields. To activate a process, the controller first asks it whether it is about to perform an internal move, or a read, or a write. Depending on the answer Wsh, it asks the process to perform a single operation, or it starts a synchronization of the process with the memory. For a read operation, the process has to

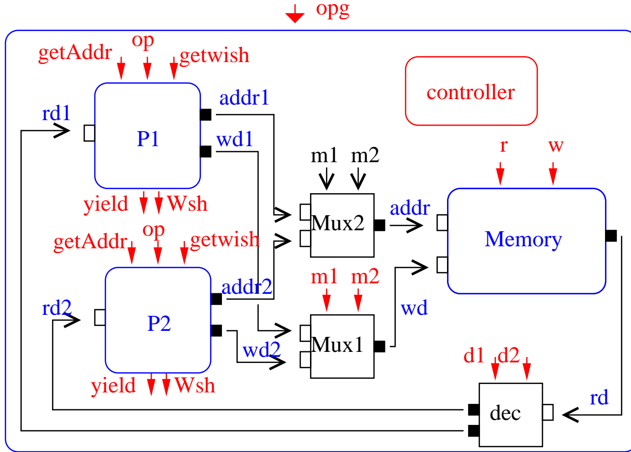

Figure 13. Asynchronous system

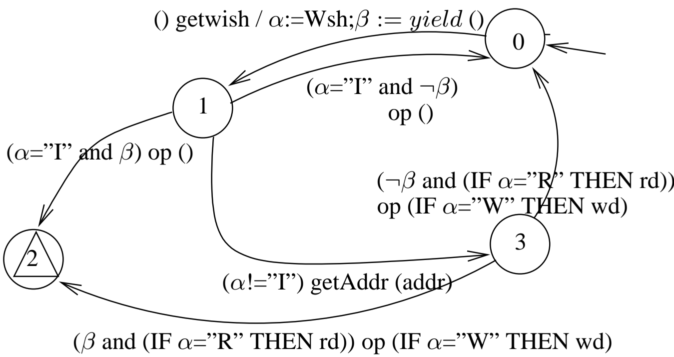

Figure 14. Protocols of the processes

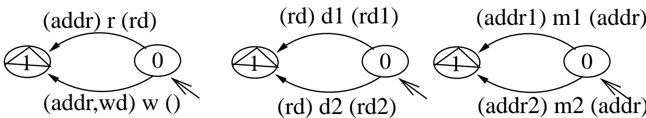

Figure 15. Protocols of the memory, Dec and Mux components

```
Controller is
var alpha in {"R","W","I"} ; beta : boolean ;
for opg do : {
  addr1,addr2,addr,rd1,rd2,rd,wd1,wd2,wd: FIFO(1,int)
P:=random(P1,P2) ;
do{ P.getwish ; alpha:=P.Wsh ; beta:=P.yield ;
   if(alpha="I"){ P.op ;}
   if(alpha!="I"){
         P.getAddr ;
         if(P=P1) { addr1.put ; addr1.get ; Mux2.m1 ; }
         if(P=P2) { addr2.put ; addr2.get ; Mux2.m2 ; }
         addr.put ; addr.get ;
   if(alpha=="R"){ M.r ; rd.put ; rd.get ;
         if(P=P1) { dec.d1 ; rd1.put ; rd1.get ; }
         if(P=P2) { dec.d2 ; rd2.put ; rd2.get ; }
         P.op ; }
   if(alpha=="W"){P.op ;
         if(P=P1) { wd1.put ; wd1.get ; Mux1.m1 ; }
         if(P=P2) { wd2.put ; wd2.get ; Mux1.m2 ; }
  wd.put ; wd.get ; M.w ; }
}}
while(not beta)}}
```

Figure 16. controller code

provide an address first (the controller asks this with `getAddr`) then the memory can be triggered, then the operation of the process. For a write operation, the process also has to provide an address, then it can be triggered (this produces a data to be written) and then the memory can be triggered.

The protocols Figures 14 and 15 gives the protocols of the processes, the memory, and the `Mux` and `Dec` components. The protocol of a process describes the dialogue that takes place between the process and the memory before actually exchanging a value.

Comments The controller we gave is consistent with the protocols of the components, but, contrary to the example of section 3.1, the architecture and those protocols are not sufficient to produce the controller. We would need additional constraints related to the master/slave relationships between the processes and the memory. Indeed, if one of the processes declares that it wants to write (or read), then the memory should be activated with the corresponding control input during the same macro-step.

The structure of the composed component does not reproduce the structure of the sub-components. The modeling is not incremental: if they are more processes to be scheduled, they should be put in the same box. But the composed component could be reused with another scheduler, describing a hierarchy of schedulers.

This 42 modeling of asynchronous behaviors with shared memory can be used to study the behavior of algorithms used in asynchronous parallel programming, like the Peterson algorithm for mutual-exclusion. The code of the controller can be used to produce all the possible interleavings of the two process behaviors, together with their effect on the memory, as a Promela/Spin [20] model would do.

4. Compatibility Issues

The code of a component should be *compatible* with its protocol: for instance, if the protocols declares that only input `id` is required, the component should not make use of the other inputs. Checking this property statically, in the general case, is a model-checking problem. It can be checked dynamically if the protocol is used to generate defensive code in the component (as it is done with Eiffel or Java contracts for instance).

On the other hand, if we consider that the controller is written independently of the protocols, then it should be *compatible* with the protocols. For instance, in the asynchronous case, if the controller triggers a process with `op` before having asked what it is about to do (and possibly which address it is about to use), the macro-step is incorrect. It can also be incorrect if the controller does not provide the components with the data inputs they require. For instance, triggering the memory (with read or write) before setting a value for the address is incorrect.

The controller/protocol compatibility property can be checked dynamically, considering the protocols as *monitors* that run "in parallel": at each macro-step, the controller starts the protocols of the subcomponents in their initial states, and makes them evolve according to the control inputs it chooses. If, at the end of the macro-step, some of the component protocols have not reached their final states, this macro-step is incorrect. Similarly, if the controller is about to make a move that is not consistent with the current states of the protocol components (for the availability of data for instance), then the macro-step is incorrect. Again, a static check of the same property is a general model-checking problem.

5. Semantics

In this section, we formalize the components, the architectures and the controllers. Formalizing the protocols and the compatibility notions of section 4, or the way controllers can be generated from protocols, is outside the scope of this paper. The semantics does not express any error detection mechanism. The micro-steps and

the macro-steps are supposed to terminate in bounded time; we assume that the controller is compatible with the protocols of the components. As a consequence, it never reads in an empty FIFO, nor write in a full one. We express the fact that a component does not always need all its inputs, nor produce all its outputs, by considering *partial valuations* of the inputs and outputs. See below.

5.1 Individual Components and Architectures

An individual component has an *interface*: the sets of input and output data signals, the sets of input and output control signals. The signals take their values in some domain that can contain Boolean values, numerical values, etc. A component has an internal "state", belonging to a finite set Σ. The most general definition of the behavior of a component is a set of relations corresponding to its possible activations through its control inputs. For each control input, the component behavior (which may be non-deterministic) is given as a relation that relates values for some of the data inputs, the current state, values for some of the control and data outputs, and a new state. Let us note $\mathcal{D}$ the union of all data types. The *partial* valuations of the interface signals are represented by partial functions to $\mathcal{D}$. We note $dom(f)$ the domain of such a partial function.

Definition 1 : Components
A component is a tuple: $C = (\Sigma, \Sigma^{\text{init}}, IC, OC, ID, OD, \mathcal{B})$ where Σ is the set of internal states, $\Sigma^{\text{init}} \subseteq \Sigma$ is the set of initial states, (one initial value is chosen when the component is instantiated) and IC, OC, ID, OD are the sets of names for the control inputs, control outputs, data inputs, data outputs, respectively. $\mathcal{B}$ is the behavior of the component, it is a total function $\mathcal{B} : IC \longrightarrow \mathcal{R}$ where $R \in \mathcal{R}$ is a relation: $R \subseteq (\Sigma \times (ID \longrightarrow \mathcal{D}) \times (OD \longrightarrow \mathcal{D}) \times (OC \longrightarrow \mathcal{D}) \times \Sigma)$. □

Definition 2 : Architectures
Let us consider a set of components:
$\{C_i = (\Sigma_i, \Sigma_i^{\text{init}}, ID_i, OD_i, IC_i, OC_i, \mathcal{B}_i)\}_I$. An architecture for combining them is a tuple $(ID_g, OD_g, IC_g, OC_g, L)$ where the first two fields describe the data ports of the assemblage, the two successive fields describe the control ports of the assemblage, and L is the set of directed links between the data ports of the components, or between the data ports of the assemblage and the internal ones: $L \subseteq (\bigcup_I OD_i) \times (\bigcup_I ID_i) \cup ID_g \times (\bigcup_I ID_i) \cup (\bigcup_I OD_i) \times OD_g$. The input and output control ports are implicitly linked to the controller. □

5.2 Controllers

Let us consider a set of components $\{C_i = (\Sigma_i, \Sigma_i^{\text{init}}, IC_i, OC_i, ID_i, OD_i, \mathcal{B}_i)\}_I$, and an architecture $A = (IC_g, OC_g, ID_g, OD_g, L)$, defining a new component C.

The controller has some internal memory in the set Σ_C that can be used across the various activations of C (on the examples of section 3, this corresponds to a control point in the program, and to the controller variables which are not attached to the links). It also has some internal memory associated with the wires $\Sigma_L : L \longrightarrow M$ that is reinitialized for each new activation. M is the union of the FIFO types.

The controller associates with each global control input icg $\in IC_g$ a program that activates the subcomponents through their control inputs, stores their data outputs into Σ_L, and gives them data inputs taken in Σ_L. These programs may be non-deterministic, and they have a final state. The controller may store the control outputs in its state Σ_C, and all its actions depend on Σ_C.

Definition 3 : controller
A controller for an architecture $A = (IC_g, OC_g, ID_g, OD_g, L)$ and a set of components $\{C_i = (\Sigma_i, \Sigma_i^{\text{init}}, IC_i, OC_i, ID_i, OD_i, B_i)\}_I$ is a tuple $(\Sigma_C, \Sigma_C^{\text{init}} \subseteq \Sigma_C, \Sigma_c^{\text{final}} \subseteq \Sigma_C, IC_g \longrightarrow \text{Progs}, S)$. A program in Progs is a tuple $(T_{\text{put}}, T_{\text{get}}, T_{\text{act}}, F)$ where $T_{\text{put}} \subseteq \Sigma_C \times L \times \Sigma_C$ (resp. $T_{\text{get}} \subseteq \Sigma_C \times L \times \Sigma_C$) is the set of all possible "put" actions (resp. "get" actions) of the controller, from a state, on a link $\ell \in L$; $T_{\text{act}} \subseteq \Sigma_C \times \bigcup_I IC_i \times \Sigma_C$ is the set of all component activations the controller may execute, from a state. $F \subseteq \Sigma_c^{\text{final}} \times (OC_g \longrightarrow \mathcal{D})$ associates final states of the controller with partial valuations for the control outputs. $S \subseteq \Sigma_C \times (\bigcup_I OC_i \longrightarrow \mathcal{D}) \times \Sigma_C$ defines how the controller stores partial valuations of control outputs of the components into its state. □

5.3 Combining Components

Combining components means considering a finite sequence of subcomponents activations and memory storage on the internal links (micro-steps), as a *macro-step* corresponding to a global activation. In order to describe how this is done, we will first describe the micro-steps and how they can be combined into sequences. Then we will define which of these micro-step sequences are the macro-steps of the new component.

In all the section below, we consider a controller $(\Sigma_C, \Sigma_C^{\text{init}} \subseteq \Sigma_C, \Sigma_c^{\text{final}} \subseteq \Sigma_C, IC_g \longrightarrow \text{Progs})$ for an architecture $A = (IC_g, OC_g, ID_g, OD_g, L)$ and a set of components $\{C_i = (\Sigma_i, \Sigma_i^{\text{init}}, IC_i, OC_i, ID_i, OD_i, \mathcal{B}_i)\}_I$.

We also consider a particular control input $ic_g \in IC_g$, and its associated program $(T_{\text{put}}, T_{\text{get}}, T_{\text{act}}, F)$.

5.3.1 State of an assemblage

The state of an assemblage of components is made of: the state of the controller (an element of Σ_C), the states of the components ($\Sigma_I = \prod_I \Sigma_i$), the states of the links ($\Sigma_L : L \longrightarrow M$, where M is the union of all FIFO types associated with the links), the states of the data ports ($\Sigma_P : (\bigcup_I ID_i \cup \bigcup_I OD_i \cup \bigcup_I OC_i \cup ID_g \cup OD_g \cup OC_g \longrightarrow \mathcal{D})$. For sake of simplicity, we assume a unique naming of all ports.

Notice that we need a state of the data ports to express the fact that a component makes some of its outputs available (resp. consumes some of its data inputs), but does not copy them onto the links (resp. from the links). The `put` and `get` operations of the FIFOs associated with the links do the job (see section 2.3). The method `put` will be represented in the semantics by a function put $: M \times \mathcal{D} \longrightarrow M$ where the assigned value is explicit and put(m, v) is the new value of m after the action `m.put(v)`. The method `get` will be represented by a function get $: M \longrightarrow \mathcal{D}$.

We will denote such a global state by a tuple $(\sigma_C, \sigma_I, \sigma_P, \sigma_L)$.

5.3.2 Micro-steps

For a given ic_g, the micro-steps that corresponds to what the controller does with the components and the links are described by the following three rules.

The rule [put] expresses that, if from its state σ_C, the controller may put a value on a link ℓ between ports P_1 and P_2, then the global state evolves with a change in σ_C and σ_L only: the link ℓ receives a new value computed by `put` with the value of its producer port P_1.

$$\frac{(\sigma_C,\ \ell = (P_1, P_2),\ \sigma'_C) \in T_{\text{put}}}{\begin{array}{l}(\sigma_C, \sigma_I, \sigma_P, \sigma_L) \longrightarrow \\ (\sigma'_C, \sigma_I, \sigma_P, \sigma_L[\text{put}(\sigma_L(l), \sigma_P(P_1)) \ /\ \ell])\end{array}} \quad \text{[put]}$$

The rule [get] expresses that, if from its state σ_C, the controller may get the value of link ℓ between ports P_1 and P_2, then the global state evolves with a change in σ_C and σ_P only: the consumer port P_2 of link ℓ receives the value taken from the link.

$$\frac{(\sigma_C,\ \ell = (P_1, P_2),\ \sigma'_C) \in T_{\text{get}}}{(\sigma_C, \sigma_I, \sigma_P, \sigma_L) \longrightarrow (\sigma'_C, \sigma_I, \sigma_P[\text{get}(\sigma_L(\ell))/P_2], \sigma_L)} \text{ [get]}$$

The rule [act] expresses that, if from its state σ_C, the controller may activate the component γ through its control input ic_γ, then the first three fields of the global state evolve. The state of the controller is modified because it stores the control outputs of the component that is activated; the state of the component that is activated is modified; the state of the ports is modified, because some of the output ports of the activated components take new values.

$$\frac{\begin{array}{c}(\sigma_C, ic_\gamma,\ \sigma'_C) \in T_{\text{act}} \\ \exists od, oc, id, \sigma'_\gamma.(\sigma_\gamma, id, od, oc, \sigma'_\gamma) \in \mathcal{B}(ic_\gamma) \\ \text{and } id = \sigma_P(dom(id))\end{array}}{(\sigma_C, \sigma_I, \sigma_P, \sigma_L) \longrightarrow (\sigma''_C, \sigma'_I, \sigma'_P, \sigma_L)} \text{ [act]}$$

$dom(id)$ is the set of inputs whose value is required by the component γ, to take the transition $(\sigma_\gamma, id, od, oc, \sigma'_\gamma)$. $\sigma_P(dom(id))$ is the valuation of these inputs, taken in the global state of ports σ_P. If the values of these inputs in id are exactly those available in σ_P, then the transition can be taken. σ''_C is the modification of σ'_C by storing the values of oc: $(\sigma'_c, oc, \sigma''_C) \in S$. If $\sigma_I = (\sigma_1, \sigma_2, ...\sigma_\gamma, ..., \sigma_n)$ then $\sigma'_I = (\sigma_1, \sigma_2, ...\sigma'_\gamma, ..., \sigma_n)$. The outputs ports are modified: $\sigma'_P = \sigma_P[od/dom(od)][oc/dom(oc)]$, where $dom(oc)$ and $dom(od)$ are the sets of control outputs and data outputs produced by the component during its transition.

5.3.3 Macro-steps

The global component is of the form $(\Sigma_g, \Sigma_g^{\text{init}}, IC_g, OC_g, ID_g, OD_g, \mathcal{B}_g)$. $\Sigma_g = \Sigma_C \times \Sigma_I$, because the state of the links and the state of the ports are not persistent across global activations. $\Sigma_g^{\text{init}} = \Sigma_C^{\text{init}} \times \prod_I \Sigma_I^{\text{init}}$.

The behavior $\mathcal{B}_g(ic_g)$ of the composed component for the particular control input ic_g we've been considering so far is a relation $R \subseteq \Sigma_g \times (ID_g \longrightarrow \mathcal{D}) \times (OD_g \longrightarrow \mathcal{D}) \times (OC_g \longrightarrow \mathcal{D}) \times \Sigma_g$.

The rule [mac] shows that the tuples of this relation R are deduced from the sequences of micro-steps that end in a final state of the controller. A macro-step only "remembers" the initial state and the final state of this sequence, the valuations of the data inputs and outputs are deduced from the state of the global ports, and the valuation of the control outputs is given by the values associated with the final state of the sequence.

The states of the ports and links are not persistent across the activations of the composed component. It means that each macro-step starts with the initial value of the links $\forall l \in L.\sigma_L^0(l) = m$ where m is the new value of m after `m.init` (see section 2.3). For the ports, the initial state is irrelevant because, if the controller is compatible with the components' protocols, then a port is never read before being written to: $\forall p.\sigma_P^0(p) = ??$.

$$\frac{\begin{array}{l}(\sigma_C^0, \sigma_I^0, \sigma_P^0, \sigma_L^0) \longrightarrow (\sigma_C^1, \sigma_I^1, \sigma_P^1, \sigma_L^1) \longrightarrow ... \\ \longrightarrow (\sigma_C^n, \sigma_I^n, \sigma_P^n, \sigma_L^n) \\ \text{ and } \sigma_C^n \in \Sigma_c^{\text{final}} \text{ and } (\sigma_C^n, oc_g) \in F\end{array}}{\begin{array}{l}((\sigma_C^0, \sigma_I^0),\ \sigma_P^0(ID_g), \sigma_P^0(OD_g), oc_g, (\sigma_C^n, \sigma_I^n)) \\ \in \mathcal{B}(ic_g)\end{array}} \text{ [mac]}$$

6. Related Work and Main Choices for 42

A lot of approaches have been proposed for heterogeneous modeling frameworks based on the notion of model of computation and communication. For instance, [31] claims that various MoCCs are needed for the various modeling and design phases; the way MoCCs interact is not formalized. [18] is very close to the motivations of 42, offering a kind of programming language in which the MoCCs can be described and executed; it is quite recent and not entirely formalized yet; for the moment it seems less adequate than 42 for describing fine grain temporal behaviors.

Other proposals, like the reactive modules [5], do not introduce MoCCs as a solution to the problem of modeling heterogeneity, but address almost the same questions, and are entirely formalized. However, reactive modules are a *language* in which a number of choices are built-in, while 42 is a general modeling framework in which new MoCCs can be described.

As far as the formalization of the various MoCCs and their interactions is concerned, the most relevant work is the family of TAG semantics, denotational or operational as in [7]. In some sense, 42 is an intermediate point of view, between the way MoCCs are programmed but not fully formalized in Ptolemy, and the way they are formalized but far from programming purposes in the TAG semantics.

A lot of other works can be considered as related work, and it would be very difficult to be exhaustive. As a complement to the above references, we discuss the main choices that have been made for the design of 42, and we compare them to the choices made in other component frameworks.

Continuous vs discrete models 42 is limited to the discrete case. When we need to include the physical environment in a model, we can consider components that are non-deterministic discretized versions of some continuous models, but we do not study how to mix continuous and discrete MoCCs. Ptolemy addresses this problem. Other proposals, like VHDL-AMS (IEEE norm 1076-1999) concern the modeling of mixed digital-analog systems, but they do not address the component aspects. Similarly, Matlab/Simulink designs can mix continuous and discrete parts, but the notion of a component is not dealt with specifically.

Strict Hierarchy A basic component, or a composed component built as an assemblage of other components, are perfectly undistinguishable in any 42 context. This is true also for Ptolemy, Fractal [10], and to some extent SystemC-TLM [14] (used for the description of systems-on-a-chip at the so-called transactional level), but not for other component models like BIP [6], in which there is no dedicated notion of encapsulation that could hide the details of an assemblage and allow to consider it as a basic component. We consider this strict hierarchy property to be a key property of component-based frameworks, because it allows to forget as much as possible about the details of the components, as soon as possible.

Oriented connections vs non-oriented ones 42 adopts a dataflow style architecture description language, with oriented connections. In Ptolemy, or the modeling tool Spice [1] for electronic circuits, this is not necessarily the case, allowing the modeling of electric behaviors. Even for modeling computer behaviors, some models choose symmetric synchronization primitives like rendez-vous, thus relying on non-oriented connections (see for instance the "Architectural Interaction Diagrams", or AIDs [28]). In 42, we concentrate on computer systems, and we claim that rendez-vous-like mechanisms are not adequate for modeling reactive systems in which the notion of inputs and outputs is a central one. Hence we choose oriented wires.

What should a modeling framework encompass? We think that a modeling framework for heterogeneous embedded systems should be usable to describe pure synchrony, because this is what exists in synchronous hardware components. It is a quite strong requirement, because it means that the definition of the MoCCs should allow to describe the fix-point computation which is the basis of any synchronous formalism (the stabilization phase illustrated in section 3.1). A lot of component-based frameworks based on a set of available *connections* (blocking, non-blocking, ...) between components do not have this expressive power. See next point.

Do connections express some behavior? In 42, the connections, or wires, only express that some information may flow from one component to another. There is no synchronization or memory attached to the connections, *a priori*. The controller may decide to manage some temporary memory corresponding to the wires in order to describe complex communication patterns, but this does not mean that the wire behaves as memory for the connected components, since the lifetime of this memory is limited to the macro-step. Moreover, the choice of the memory attached to the wires is part of a particular MoCC, this is not built in the 42 general modeling framework. [18] adopts the same point of view.

Some component frameworks rely on communication patterns expressed directly by the connections, for instance point-to-point connections with a finite (small) set of synchronization effects made available, like: blocking write on one side, non-blocking read on the other side; this is quite restrictive. The need for more complex communications patterns usually leads to the following solution: if a communication pattern has a complex behavior, it is a component, not a connection. This is the case for SystemC-TLM [14], where buses, or even networks-on-a-chip, are considered as components. In this case, the remaining connections between components (including the communication components) are only links between ports, as in 42.

Architecture Description Languages and MoCCs In 42, the architecture description language has a dataflow style. Since the connections have no meaning until the MoCC is defined as a controller, this only means that we express data dependencies explicitly in the ADL, and control aspects in the MoCC. As we mentioned in section 1.3, Ptolemy is more liberal; Rialto [9] follows Ptolemy on this point. For 42, we chose only one style of architecture description language, because we want a simple formal semantics.

Atomicity 42 is built in such a way that the operation that encapsulates subcomponents C_i to build a new component C also defines what atomicity is, for C: starting from a notion of atomicity as viewed by the C_is (their various activations), the controller defines the activations of C by considering a sequence of actions as atomic. The activations of C can be viewed as *macro-steps*, corresponding to a sequence of *micro-steps*. In other words, forgetting details about the internals of a component also means considering its activations are atomic. It is compulsory to be able to reason locally on a component, without caring about potential interrupts from the context in which it will be used.

Time and Temporal granularity It may seem at first sight that 42 does not allow to deal with time. In fact, it adopts the principle of synchronous languages called *multi-form time* first introduced by G. Berry for Esterel [8]: physical time is nothing special but a sequence of events (seconds, milliseconds, ...), and any sequence of events (counting meters, or beacons on a track) can be considered as a time scale for the reactive system that perceives these events. Using timed automata [4] to deal with time in 42 would limit drastically the manipulation of time-related notions. For instance, in TAID (a timed extension of the AIDs already mentioned above), it is impossible to model distributed systems, because the way time is modeled implies a notion of global clock.

The multi-form time principles also allows to consider several related time scales (seconds and milliseconds, ...).

Specifying components, and the notion of a contract Protocols are used to specify components with a behavioral description [27]. Component protocols are usually expressed with some finite-state-machine formalism. For instance, in [13], each transition is labeled by a message suffixed by a symbol defining whether the message is passed or received. Other approaches (see, for instance [32]) express multiple protocols for one component, each protocol being related to a specific component interface. Protocols described as behaviors are usually exploited to determine some properties over a set of components such as compatibility, composability and substitutability [13].

42 has very expressive *control contracts*, i.e., protocols that talk only about the control inputs/outputs and the *availability* of data, not about the *values* of the data exchanged. In Signal, the powerful notion of constraint can mix control (clocks) and data values in the same expression. It is also the case in Lustre, with a reduced expressive power. The choice we made for 42 is to distinguish between control and data (it is also the principle of the LAAS architecture for intelligent robots [3]), because the control between subcomponents, and the moments when data is available, have a lot to do with the MoCC described, while the values of data have not.

Expressive Power of the Controller 42 is a very liberal framework, since the MoCCs can be programmed. However, the expressive power of the controller is limited by several points. The fact that the memory associated to the wires is not persistent over global activations means that any memory needed by the components to synchronize with each other has to be described explicitly, as an additional component, it cannot be "hidden" in the controller.

Conversely, the inter-macro-step memory in the controller is needed because the way the controller activates the subcomponents is not necessarily memoryless. For instance, a fair scheduler for asynchronous processes will remember the last choice. Moreover, the controller sometimes needs some inter-macro-step memory in order to make sure that the components' protocols are met: for instance, if a component emits a control output `o` meaning: next time I'll be activated, I'll need my input `i`.

An interesting question (orthogonal to the previous remarks) is whether we need parallelism in the controller. Since the controller is there to express the semantics of parallelism and communication between the components, it seems that this would just move the problem to another level. In fact, for *modeling* purposes, we conjecture that a non-deterministic controller that produces interleavings of the components' activities is enough. For *implementation* purposes, it might not be the case.

7. Conclusion and Further Work

We have defined the general framework 42 for component-based modeling of heterogeneous systems, and shown its use for synchronous or asynchronous systems. For the moment 42 is tested under the Java implementation of Fractal [10]. The components are implemented in Java, respecting both the Fractal and 42 forms. We require that each of the components implement interfaces specific to the MoCC where it evolves, so that we can use generic controllers, implemented in Java. When the controller can be generated from the components' protocols and the architecture, we can hide the controller for the user.

We will study an implementation in Lustre to benefit from the associated execution, testing and verification tools. For instance, a description of both the protocols, and the controller, in a formally defined language, could help in proving that the assemblage of components is correct. Further work will be devoted to semantical issues, to the definition of concrete languages for the controller, and to the extension of 42 with non-functional aspects.

Concerning *semantical issues*, we will compare our quite operational semantics with the family of TAG semantics, along the lines of [7]. The idea is to relate the operational view of MoCCs implemented by controllers, with the fully declarative view of MoCCs as expressed in the TAG semantics. We will also study how to characterize the expressive power of the controller and how to formalize the notions of compatibility of section 4.

In order to use 42 as a high-level modeling framework, we need to define *concrete languages* for the controller. For the examples we

have used an imperative style with calls to a `random` function when needed, but we could think of a language based on constraints, to avoid explicit calls to `random`. Reusing the language Lutin [29] that has been defined for the generation of test scenarios or the simulation of non-deterministic systems, could be an idea. This is related to the idea of generating the controller automatically from the protocols and possibly additional constraints (like the master/slave constraints of section 3.2).

Finally, in order to extend 42 with the modeling of *non-functional* aspects, we will study how to integrate into 42 the ideas that we used for the modeling of energy consumption in sensor networks [30, 24]: each component has a non-functional model (an automaton with consumptions attached to the states) and the parallel composition of two such models defines precisely what are the consumptions attached to the combined states. In 42, the components could have a functional and a non-functional parts, and the controller could also have a functional part (as described in this paper) and a non-functional part describing how the non-functional models of the components are composed, depending on the MoCC.

References

[1] The spice page. `bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/`.

[2] Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. 1979.

[3] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research, Special Issue on Integrated Architectures for Robot Control and Programming*, 17(4), 1998.

[4] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[5] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[6] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12. IEEE Computer Society, 2006.

[7] A. Benveniste, B. Caillaud, L. P. Carloni, and A. L. Sangiovanni-Vincentelli. Tag machines. In Wayne Wolf, editor, *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*, pages 255–263. ACM, 2005.

[8] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[9] D. Björklund and J. Lilius. A language for multiple models of computation. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES-02)*, pages 25–30, New York, May 6–8 2002. ACM Press.

[10] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in java. *Softw, Pract. Exper*, 36(11-12):1257–1284, 2006.

[11] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):0, 1994.

[12] P. Caspi, C. Mazuet, and N. Reynaud Paligot. About the design of distributed control systems: The quasi-synchronous approach. In Udo Voges, editor, *Computer Safety, Reliability and Security, 20th International Conference, SAFECOMP 2001, Budapest, Hungary, September 26-28, 2001, Proceedings*, volume 2187 of *LNCS*, pages 215–226. Springer, 2001.

[13] "L. de Alfaro and T. A. Henzinger". "interface automata". In *"ESEC/FSE-9: Proceedings of the 8th European software engineering conference"*, pages 109–120, New York, NY, USA, 2001. ACM Press.

[14] F. Ghenassia. *Transaction Level Modeling With SystemC: TLM Concepts And Applications for Embedded Systems*. Springer-Verlag, 2005.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[16] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[17] N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Sixth International Conference on Application of Concurrency to System Design, ACSD 2006*, Turku, Finland, June 2006.

[18] C. Hardebolle, F. Boulanger, D. Marcadet, and G. Vidal-Naquet. A generic execution framework for models of computation. In *International Workshop Series on Model-based Methodologies for Pervasive and Embedded Software (MOMPES)*, 2007.

[19] C. Hewitt. A universal, modular actor formalism for artificial intelligence. In *Proc.International Joint Conference on Artificial Intelligence*, 1973.

[20] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[21] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[22] F. Maraninchi and L. Morel. Logical-time contracts for the development of reactive embedded software. In *30th Euromicro Conference, Component-Based Software Engineering Track (ECBSE)*, Rennes, France, September 2004.

[23] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[24] F. Maraninchi, L. Samper, K. Baradon, and A. Vasseur. Lustre as a system modeling language: Lussensor, a case-study with sensor networks. In *SLA++P'07, ETAPS'07 Satellite Workshop on Model-driven High-level Programming of Embedded Systems*, Braga, Portugal, March 2007. ENTCS.

[25] B. Meyer. *Eiffel: An Introduction*. Interactive Software Eng., 1988.

[26] R. Milner. A calculus of communication systems. In *LNCS 92*. Springer Verlag, 1980.

[27] S. Plasil, F.; Visnovsky. Behavior protocols for software components. *Software Engineering, IEEE Transactions on*, 28:1056– 1076, 2002.

[28] A. Ray and R. Cleaveland. Architectural interaction diagrams: AIDs for system modeling. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 396–407, Piscataway, NJ, May 3–10 2003. IEEE Computer Society.

[29] P. Raymond, Y. Roux, and E. Jahier. Specifying and executing reactive scenarios with lutin. In *SLA++P'07, ETAPS'07 Satellite Workshop on Model-driven High-level Programming of Embedded Systems*, Braga, Portugal, March 2007. ENTCS.

[30] L. Samper, F. Maraninchi, L. Mounier, and L. Mandel. Glonemo: Global and accurate formal models for the analysis of ad-hoc sensor networks. In *InterSense: First International Conference on Integrated Internet Ad hoc and Sensor Networks*, Nice, France, May 2006. IEEE.

[31] I. Sander and A. Jantsch. System modeling and transformational design refinement in forsyde. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(1):17–32, 2004.

[32] "A. Vallecillo, V. T. Vasconcelos, and A. Ravara". Typing the behavior of software components using session types. *Fundam. Inf.*, 73(4):583–598, 2006.

[33] Jan van den Bos. PROCOL: A protocol-constrained concurrent object-oriented language. *Information Processing Letters*, 32(5):221–227, 1989.