Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

42: Programmable Models of Computation for the Component-Based Virtual Prototyping of Heterogeneous Embedded Systems

Florence Maraninchi, Tayeb Bouhadiba

Verimag Research Report n^o TR-2009-1

January 2009

42: Programmable Models of Computation for the Component-Based Virtual Prototyping of Heterogeneous Embedded Systems

Florence Maraninchi, Tayeb Bouhadiba

January 2009

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 heterogeneous embedded systems called 42. We aim at expressing fine-grain logical timing aspects and several types of concurrency as MoCCs, but we require that all the MoCCs be described in terms of more basic primitives, as small *programs*. 42 also enforces precise specifications of components, in the form of *control contracts*. 42 is meant to be an abstract description level, appropriate for the *system-level* description of embedded systems, not for the development of the embedded software itself. 42 is meant to be connected to existing validation tools (formal validation, automatic testing, etc.).

Keywords: Heterogeneous embedded systems, component-based design, system-level modelling , semantics, models-of-computation, contracts

Reviewers: Pascal RAYMOND

Notes:

How to cite this report:

@techreport { ,
title = { 42: Programmable Models of Computation for the Component-Based Virtual Prototyping of Heterogeneous Embedded Systems},
authors = { Florence Maraninchi, Tayeb Bouhadiba},
institution = { Verimag Research Report },
number = {TR-2009-1},
year = { },
note = { }
}

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 seems to be a wide agreement on the fact that the main difficulty is due to the intrinsic *heterogeneity* of embedded systems. We list the main causes 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;
- It is usually made of *concurrent* objects, but the concurrency model varies from pure synchrony (e.g., a mono-clock synchronous circuit) to pure asynchrony (e.g., 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 [19] is 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 [32]), 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.

1.2 System-Level Modeling

Despite these causes of heterogeneity, the design of embedded systems requires that we be able to reason precisely on *timing*, *atomicity* and *concurrency*. For instance, in embedded control applications, it is very important to know about the input sampling rate, for the validity of the control laws that are implemented by the computer system. It is also important to be able to express that several computations should be done with *the same values* of the inputs, before considering new ones. The latter implies that we be able to reason on the *atomicity* of partial behaviors.

System-Level modeling concerns the task that has to be done when assembling various components in order to build a system. Even if the individual hardware and software components are of very good quality, and have already been validated by testing and intensive use, problems may happen when they are put together to form a system.

System-Level modeling enables the description of the system as a assemblage of components, and concentrates on what happens at the system-level, for instance synchronizations. A system-level model is often *executable*, and thus usable as a *virtual prototype* of the system, on which various functional and non-functional properties can be evaluated long before the actual system is available. In some particular approaches, a system-level model can also serve as a guide for the implementation, in a *model-driven* approach.

The hardware industry, facing the complexity of modern systems-on-a-chip, has proposed the notion of *transaction-level modeling, or TLM*, for the system-level descriptions of such objects. The language SystemC and its simulation engine are becoming a de facto standard of this industry, to develop virtual prototypes of complex hardware platforms, on which the embedded software can be developed.

The potential lifetime of *wireless sensor networks*, made of thousands of small nodes powered by a battery and communicating by radio, is often estimated by building virtual prototypes with dedicated

simulators. It is compulsory to observe the effects that are visible only at the system-level, when the effects of hardware, the protocols, and the application interfere.

SysML [45], an extension of UML, has also been proposed to tackle specific system-level modeling problems. For the moment it is not executable, and the semantics is not formalized, so it has limited uses.

Metropolis [4] is a formally defined methodology for the development of embedded systems. It allows a separation between the functional model (software components that synchronize by events, plus scheduling constraints) and the architecture model (abstract hardware model considered as an API for the software). The two models are used jointly to define a mapping model. Components communicate via functions calls. Metropolis is a system-level approach, where the MoCCs are fixed, to allow automatic mapping of the software onto the hardware.

More related work can be found in section 6.

1.3 Programming vs Architecture-Description Languages

A cause of heterogeneity may be 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 [20], Signal [28]), with notable combinations like 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*, i.e, 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.

1.4 Ptolemy

Since 42 is inspired by Ptolemy [10], we recall here the main characteristics of this component framework. Components are actors in the sense of [24]; 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, or as an automaton, 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. This is a key point in Ptolemy for building the family of *modal models*, in which an automaton is used to control several activities associated with its states, and described with other MoCCs.

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.

1.5 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 [20] or Signal [28]) 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 by means of so-called *observers* [22]. A notion of logical-time *contract* for a synchronous component has been proposed [30]. The Lustre toolbox offers verification tools that can be used to answer the classical

[1] Simulink and Stateflow are trademarks of The MathWorks

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 (the register-transfer-level hardware description languages used for the description of synchronous circuits are equivalent to a synchronous language).

Since the work by Milner [33] in the early 80s, 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 [11, 21]. The main idea is as follows: for modeling purposes, all the components are equipped with *activation conditions* and they do nothing when this condition is false. Everything is composed synchronously, the activations conditions being global additional inputs. If there is no constraint at all on these inputs, the synchronous composition adequately describes the pure asynchrony between the components. If the activation 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 the context in which components 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). The most interesting instance of this principle is the so-called *quasi-synchronous* approach, to describe systems made of several processors that are not explicitly synchronized. Their respective clocks may differ, but not in a completely unknown way. These systems are modeled by a quite liberal constraint on the clocks, namely: *there are never more than two ticks of one clock between two ticks of the other one*. Considering the two processors as perfectly asynchronous would be useless. Robust control laws can indeed be implemented on such a distributed architecture, exploiting the knowledge about the clocks.

1.6 Operational Description of Various MoCCs

Part of the approach described in the previous section relies on the fact that several models of concurrency and several synchronization mechanisms can indeed be encoded into a more expressive formalism. A synchronous language can serve as the unifying formalism, but there are others.

42 belongs to a family of works on the operational descriptions of MoCCs, usually with the objective of giving their semantics in some common framework so that they can be compared.

1.7 Motivations for 42, and contributions

The main motivations for the definition of 42 are the following:

- There is a crucial need for *system-level* descriptions, i.e., descriptions of how individual hardware or software components — or even models of the physical environment — are assembled, and how they behave together. There are several proposals in industrial or academic contexts, but it is quite recent. 42 allows to focus on how heterogeneous components can be assembled.
- There is probably little hope to define and impose *the universal formalism* for such a need, in particular because of very important cultural differences between the domains of embedded system design (critical domains like avionics, consumer electronics, smart cards, etc.); but, in any context where such a formalism is needed, its definition must include the answer to some very crucial questions, among which the type of concurrency and communication/synchronization that should be considered. These questions are fortunately less numerous than the languages of formalisms that already exist or will be designed in the future. 42 is a way of concentrating on such questions, independently of any particular language.
- Assembling components, especially when they may be as heterogeneous as a processor, an operating system, and a model of a radio channel, requires that we be able to specify precisely their "instructions for use". Most of the problems that appear at the system level are due to a bad use of some component; checking assemblages relies on precise specifications of the components. 42 proposes a notion of *protocol* for a rich description of the component *contract* (in the sense of the contracts of [32]).

- Reusing components in the development of embedded systems also means that the components have indeed been developed with this idea in mind. This means that there is a clear understanding of encapsulation, in all the domains considered. For instance, there is a very clear definition of a component in the hardware industry, at the RTL level; IPs (for "intellectual properties") are currently being bought and assembled. For software components this is less generic. 42 is designed with the idea of enforcing the FAMAPASAP principle ("Forget As Much As Possible As Soon As Possible"); this means a systematic analysis of the details that can be hidden in a component, vs the details that have to be exposed; this usually depends on the MoCC in which the components will be used: a component has to be prepared for one or more MoCCs; sometimes it can be made MoCC-generic, sometimes it cannot.

42 is essentially a tool for reasoning on the questions listed above. However, we designed it in such a way that it easily gives solutions, at least in the existing frameworks we know best, and that cover a significant part of the domain: embedded control implemented by distributed systems; systems-on-a-chip used in consumer electronics; virtual prototyping of complex systems like wireless sensor networks. In particular, we will be able to *import* existing software or hardware components, and to *wrap* them into 42 components. 42 is also *executable*, to serve as a virtual prototyping tool.

The last important point to be made clear is the following: a language or formalism that tackles all the points given above as motivations does not already exist yet.

Software components designed for a particular MoCC do exist (for instance, software components to be used in a client-server context); these models may serve as implementation guides on hardware architectures that offer the support for this MoCC; client-server models are a programming model, implementation methods have been developed, and the way the software components have to be prepared for such a use if well defined.

However, when the architectures (or execution platforms) have some variability, there is no unifying programming model. For system-on-a-chip design, there are clearly two situations: 1) a class of architectures is defined, offering a standardized interface to software; it is associated with a programming component-based model, and automatic mapping of the software onto the hardware is possible; see, for instance [36]; 2) the executions platforms are not standardized; in this case, we need a model of the hardware, to be combined with the model of the software, at least for simulation purposes. Such combined models need several MoCCs.

1.8 Structure of the paper

The paper is structured as follows: in Section 2 we define 42 informally; Section 3 details three examples (Kahn Networks [27], synchronous circuits or programs, a heterogeneous system); Section 4 formalizes the semantics of the MoCCs, expressed as *controllers*; Section 5 formalizes the semantics of the components' specifications, or *protocols*; Section 6 lists some related work, and summarizes the main choices that we made for 42; Section 7 is the conclusion.

2 Informal Definition of 42

We first give an example-oriented informal definition of 42: the individual components and their protocols, the way components are assembled; the compatibility questions that can be studied in such a framework.

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). Allowing non-deterministic components means that 42 allows to mix code with specifications.

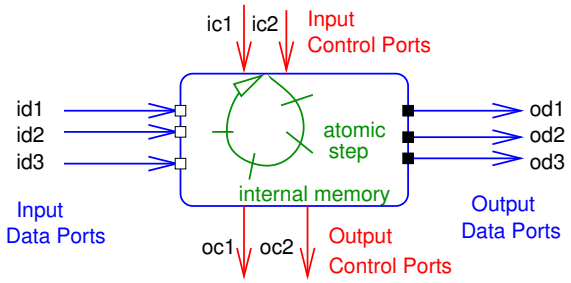

Figure 1: A 42 Component

```
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 component

Figure 2 is an example code for such a component, written in some imperative style. 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 2.4 below).

42 does not impose a particular language for the individual components. When importing existing components (for instance the C code produced by the compiler of a synchronous language), they have to be wrapped so that the data ports correspond to some of their internal variables, and the input control ports to the activation of a piece of code (methods in an object-oriented language, functions in C, ...). For hardware components, the 42 ports usually correspond to wires.

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 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*.

The restrictive constraints on the connections for the data and control ports are meant to enforce the identification of such a data/control classification for the components, as illustrated in the examples of section 3. Notice that, if the controller has to take decisions that depend on a data output `od` of some component `C`, the `od` wire can be connected to the input data wire of some special component `cond` that produces a control output for the controller. Similarly, if a global data output is in fact produced by the controller (and so cannot be connected to a data output of some component), we can just add a special component activated by the controller, and producing the desired data output.

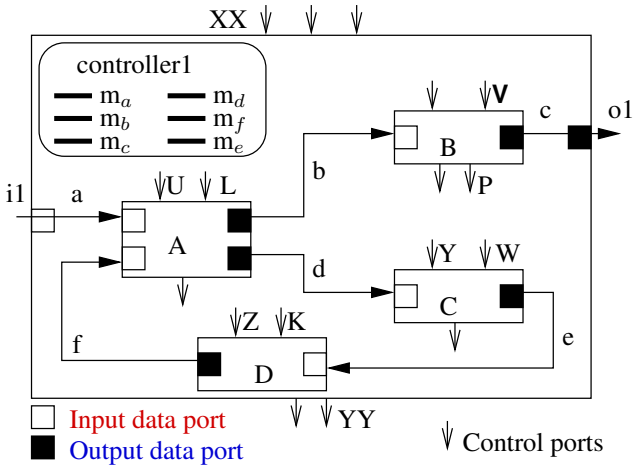

Figure 3: Connecting Components, an Example

```
Controller1 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

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*); it also reports on the activity of the subcomponents, through output control ports like `yy`. 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 an explicit 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.

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` gets a value in `M` and puts it into the consumer port connected to the wire; `M.put` gets a value in the producer port connected to the wire and puts it into `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' contents 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). Finally, it may set a value for the global output ports (e.g., `yy`).

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`.

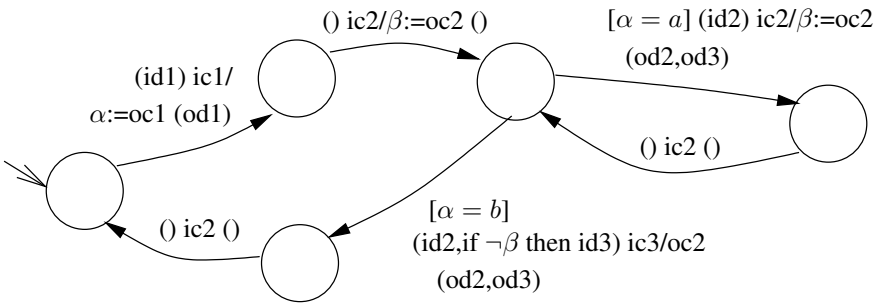

Figure 5: Example protocol for the component of Figure 1

2.4 Specifying Components: Control Contracts

2.4.1 General notions

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 [47]). When specifying a class in an object-oriented framework, a protocol can be used to specify, for instance, 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. A protocol 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 [28]; although Signal does not have a built-in distinction between control and data ports, control ports correspond more or less to *clocks*.

Figure 5 shows a protocol for the component C of Figures 1 and 2. It is an automaton with an initial state (pointed to by the little arrow). Each transition has a label of the form:
`[condition] (data req) control input / control outputs (data prod).`
The variables denoted by Greek letters are used to store the value of control outputs, and may be used later on in the protocol itself; let us note the set of such variables $\mathcal{V}$. The `[condition]` part is built from the variables in $\mathcal{V}$. The `(data req)` part expresses conditional data dependencies; the conditions are built on $\mathcal{V}$ too. For instance `(id2,if` $\neg\beta$ `then id3)` means that the transition needs `id2` and, if the value stored in variable β is false, it also needs `id3`. The `(data prod)` is built similarly, and expresses which outputs are indeed produced. The `control input` is a single control input. the `control outputs` gives the control outputs that are indeed produced, and may indicate that their values are stored in variables of $\mathcal{V}$. The variables in $\mathcal{V}$ should not be used before they are assigned a value.

Initially, the protocol is in its initial state, and then it evolves according to the sequence of activations produced by the controller in response to a sequence of macro-steps. Each macro-step is considered to start in the state where the protocol was at the end of the previous macro-step.

2.4.2 Specifying accepting states

We could enrich this notion of protocol with *accepting* states, in order to enforce macro-steps made of several activations of the same components, in a given order. In this case, each macro-step should place the protocol in an accepting state. We do not use this extension in the examples of the paper.

2.4.3 Talking about fresh values of the inputs

The protocols we defined in the previous section can also be enriched by distinguishing two ways of requiring inputs. A component may specify that it is ready to deal with new inputs or, conversely, that he would like to keep the same inputs as last time it was activated.

An example of such a need will be given in the example of synchronous circuits or programs below (section 3.2).

The notation will be as follows: for each data input `id`, two forms of it may appear in the conditional specification of the inputs required: `id`, or $\overline{\texttt{id}}$. $\overline{\texttt{id}}$ means the component requires the input `id` to be the same as the previous time it was activated (or it is the beginning of time); `id` means that it does not care about inputs being new or not.

2.5 Compatibility Issues

2.5.1 Compatibility between a component and its protocol

The code of a component should be *compatible* with its protocol: for instance, if the protocol 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 complex program analysis problem. In most programming languages, determining which variables are read by a particular piece of code is undecidable. The compatibility property can be checked dynamically if the protocol is used to generate defensive code in the component (as it is done for the contracts of the language Eiffel [32]).

Designing a correct pair (component, protocol) is the responsibility of the component provider.

2.5.2 Compatibility of a controller with the components' protocols

When several components are assembled, with a controller, to form a new component, the controller should be *compatible* with the protocols of the components. In particular, it cannot activate a component with a control input `ic` that requires input `id` without first providing a value for `id`. While responding to a sequence of macro-steps, it should not produce a sequence of activations for a component C that does not respect the protocol of C (does not belong to the language described by the protocol of C, see section 5 for a formal definition).

We will formalize this controller/protocol compatibility property. It can be checked dynamically, considering the protocols as *monitors* that run "in parallel": at each macro-step, the controller makes the protocols of the subcomponents evolve according to the control inputs it chooses. If the controller is about to make a move that is not consistent with the current states of the components' protocols (for the availability of data for instance), then the macro-step is incorrect. The decidability of a static check of the same property depends on the expressive power of the language used for the controller.

3 Detailed Examples

3.1 Kahn Networks

The description of Kahn Networks in 42 is an interesting exercise which shows the main difference between: 1) the potential memory present as a component, and the volatile memory associated with the wires for a particular MoCC; 2) the code of a component, and the piece of program that is used to describe the controller in a particular MoCC.

To describe a Kahn Network in 42, we need to use explicit infinite FIFO queues between the processes. These FIFO components have an explicit control input `test` that allows to test them for emptiness; the answer `r` is an explicit control output. The idea is that this information may be used by the controller (which describes the semantics of KPN) but not by the components themselves, as defined by the KPN model. The FIFO components also have two control inputs `in` (to accept a value) and `out` (to deliver a value).

The processes are 42 components with a single control input `GO`.

The controller selects a process to be executed, and verifies that all its inputs are available, by testing the input FIFOs. If yes, it takes the inputs in the corresponding FIFOs, the process is activated with `GO`, and it produces all its outputs, that are stored in the corresponding FIFOs.

The protocols are illustrated Figure 9. The protocol of a component representing the process P3 is given Figure 9-(a): for any activation by `GO`, it needs all its inputs, and produces all its outputs; the protocols of

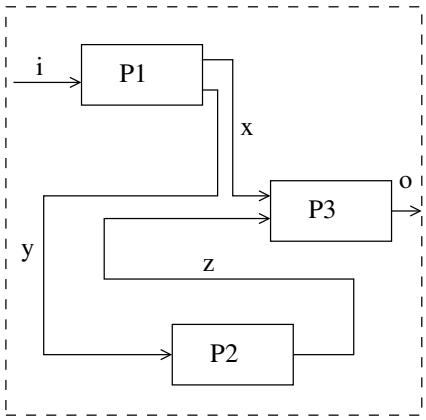

Figure 6: An Example Kahn Process Network

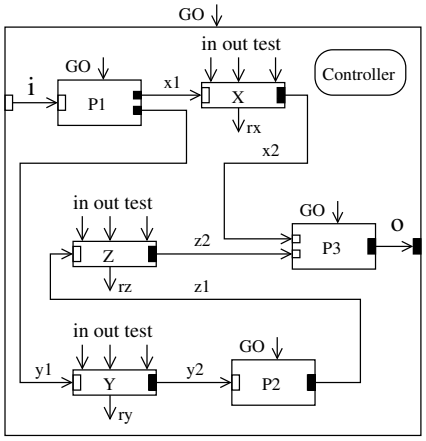

Figure 7: The 42 View of the KPN Example

```
controller is {
var x1,x2,y1,y2,z1,z2,i,o : FIFO(1,int)
var alphax,alphay,alphaz : boolean;
for go do: {
  int process =random(1,3);

  switch(process){
    case 1: i.put; i.get; P1.go;
            x1.put; x1.get; X.in;
            y1.put; y1.get; Y.in;
    case 2: Y.test;
            alphay:=ry;
            if(alphay) then{
              Y.out;
              y2.put; y2.get; P2.go;
              z1.put; z1.get; Z.in;
            }
    case 3: X.test; Z.test;
            alphax:=rx; alphaz:=rz;
            if(alphax && alphaz) then{
              X.out; x2.put; x2.get;
              Z.out; z2.put; z2.get;
              P3.go; o.put; o.get;
            }}
```

Figure 8: The programs of the controller

the others processes are similar. Figure 9-(b) is the protocol of the FIFO `X`. At any time, the FIFO may be activated with `in`, since it is unbounded, and by `test`, to tell whether it is empty. It may be activated by `out` only after a `test` activation that has answered true. The protocols of the other FIFO's are similar.

3.2 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. In the context of a component model like 42, with components and interactions between them, being able to express pure synchrony means that we should be able to describe the steps of the stabilization phase; it is not a simple interaction.

3.2.1 Example

Figures 10, 11 and 12 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

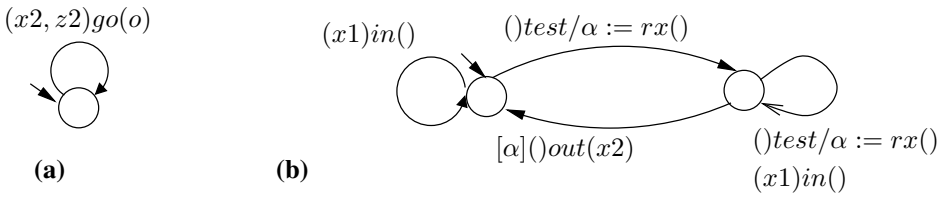

Figure 9: Protocols of the components. (a) Process P2; (b) FIFO X

```
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 10: An Example Synchronous Program (in textual Lustre)

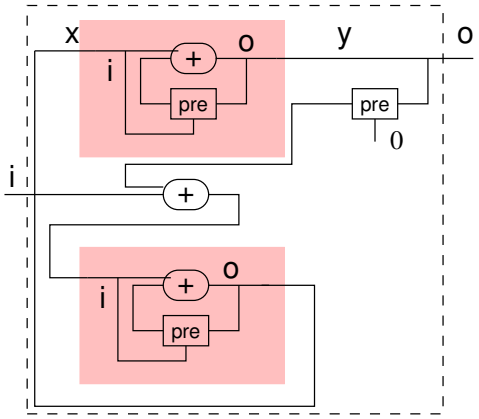

Figure 11: The same program in a graphical form

synchronous hardware design. The program of Fig. 10 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 11 is a flat and graphical view of the node `DoubleIntegr`, where the two copies of `Integr` have been expanded.

3.2.2 Individual Components

Figure 12 is the component view of the program. The level of `controller1` is the level of `DoubleIntegr`. The details of the instance `integr1` are hidden, `integr1` is considered as a basic component, as +, `pre` (a flip-flop, or elementary memory point) or the duplicator `Dup`. The instance `integr2` is described as a 42 assemblage of more primitive components.

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. Figure 17 is the code of the `pre` component (for integers).

3.2.3 The controller

At the two levels of the hierarchy, the controllers associate one-place buffers with all wires. The "programs" they play when the global `geto` or `go` control inputs are activated are given in Figures 13 and 14, in some imperative style.

When a component (`DoubleIntegr` or `Integr2`) is activated with `geto`, the inputs are 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

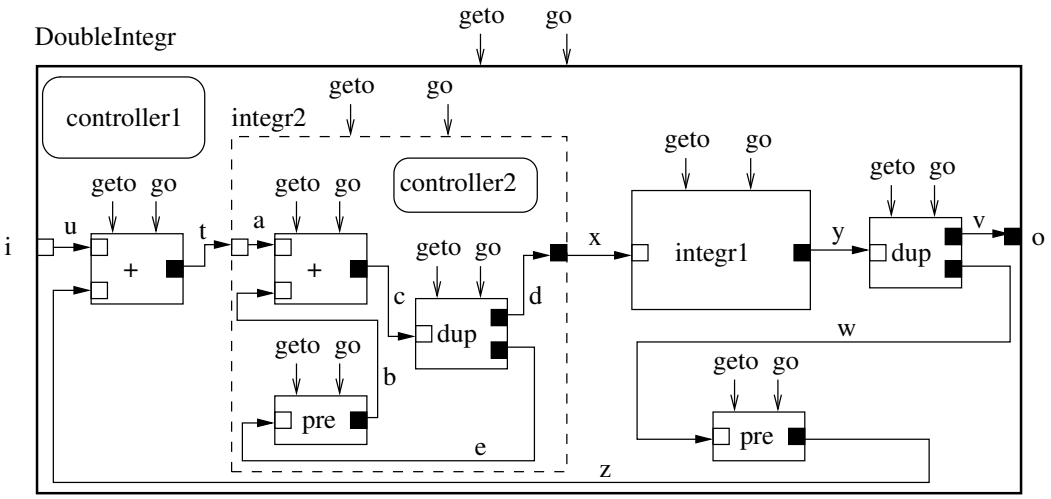

Figure 12: A component view

```
controller1 is { var u, v, w, x, y, z, t : FIFO(1,int)

 for geto do: {                              for go do: {
  pre.geto ; z.put ; z.get ;                        w.put ;w.get ;
  u.put // reads i                                  pre.go ;
  u.get ; plus.geto ;                               integr1.go ;
  t.put ; t.get ; integr2.geto ;                    dup.go ;
  x.put ; x.get ; integr1.geto ;                    integr2.go ;
  y.put ; y.get ; dup.geto ;                        plus.go ;
  v.put ; v.get ; } // defines o                    }}
```

Figure 13: The programs of the controller 'controller1'

```
controller2 is { var a, b, c, d, e : FIFO(1,int)

 for geto do: {                              for go do: {
  pre.geto ; b.put ; b.get ;                        e.put ;e.get ;
  a.put // reads t                                  pre.go ;
  a.get ; plus.geto ;                               plus.go;
  c.put ; c.get ; dup.geto ;                        dup.go ;
  d.put ; d.get ; } // defines x                    }}
```

Figure 14: The programs of the controller 'controller2'

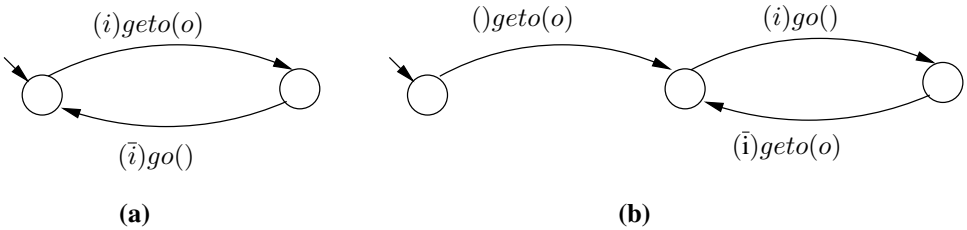

Figure 15: Protocols of the components for the synchronous MoCC

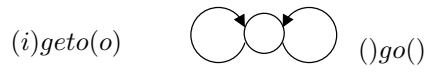

Figure 16: Protocols of the combinatorial components

```
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 17: Code of the PRE Component

```
initializations ;
while true {
  // read input into  i
  i.put (read()) ;
  // write the value produced
  DoubleIntegr.geto;
  write(o.get) ;
  DoubleIntegr.go;
}
```

Figure 18: Structure of a program using the main component `DoubleIntegr`

dependencies. For `controller1`, 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 `Dup`. 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 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. The `go` activations of all the subcomponents can be called in any order.

The programs of the controllers 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 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 11), it is sufficient to have all the cycles broken somewhere, but it could be inside the subcomponents.

3.2.4 The Protocols

In the simple case described above, the protocols for the components are those of Figure 15. **(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; the first `geto` needs no inputs because it delivers the default value hardwired in the component. $\bar{\text{i}}$ means that `go` has to be performed with the same value of `i` than the previous `geto` activation.

The protocols of the combinational components (+, `Dup`, ...) are very simple. See Figure 16. `go` is ignored, and `geto` is always accepted; it needs all inputs, and produces all outputs.

3.2.5 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 an output does 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 2.4 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.

3.2.6 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 of a main program using the component `DoubleIntegr` is given Figure 18.

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.

3.3 A Heterogeneous System

This example of Figure 19 is intended to demonstrate how heterogeneity is dealt with in 42. We consider a system made of two processors connected to a memory via a bus. When a processor is engaged in some READ or WRITE access to the memory, the second one may also wish to start a READ or WRITE. It requests access to the bus, but the bus will make it wait until the end of the ongoing transfer.

For one of the processors we describe the embedded software (it is a software component, whose interface complies with the model of section 3.2 above). For the other processor, we ignore the details of the embedded software, and only provide a very abstract and non-deterministic model of its behavior. This example covers various sources of heterogeneity mentioned in the introduction: software vs hardware, synchronous vs asynchronous, detailed vs abstract.

For sake of clarity in the rest of this section, each tuple of interconnected ports and the wire connecting them have the same name.

3.3.1 Example, highest level of hierarchy

The highest level of the model of Figure 19 is a simple modeling of a hardware architecture, very much in the spirit of so-called *transaction-level modeling* [19]. The connections between the hardware elements (the two processors, the memory, and the bus) are not given in full details as it would be the case at the register-transfer level (RTL). The exchanges between components are *transactions*, encapsulating the synchronizations that are necessary for one data exchange. We do not assume any synchronization between the two processors, which may access the memory at any time. The bus performs arbitration if necessary.

The 42 controller needed at this level is a *hardware transaction-level simulation* controller, i.e., a controller that simulates the potential physical parallelism between the hardware parts in a non-deterministic way. A hardware model at this level of abstraction concentrates on the synchronization points between components. A simulation controller should animate all the components and make data available for them. Since it does not know about the precise behavior and needs of the components involved (it is entirely generic), it relies on the components' protocols in order to know whether components are indeed ready to play.

Components and their protocols At this level of hierarchy, each component has a single control input `GO`, to be activated by the simulation controller. The idea is that, for each activation, a component representing a piece of hardware performs some internal computation, until the next access to the memory. Each component also has a `wsh` control output to report to the controller the kind of exchange operation it is engaged in (READ or WRITE). This assumption is part of the general guidelines on how to prepare components for their use in such a hardware simulation framework. Below we give more details for the processors, the memory and the bus, and we summarize these points in paragraph 3.3.1. The protocol of a particular component explains what data is needed for each simulation activation by the controller.

The component `M` models a memory which has two input data ports `a` and `wd` for the address and the data to be written. The `rw` data port is used to specify whether the memory will be engaged in a read or

write transaction. The output data ports `rd` and `res` are used to deliver the data read and the result of this memory operation (SUCCESS or ERROR).

Figure 21-(b) gives the protocol of the memory component. For the first activation, it needs an address `a` and the specification `rw` of the type of operation to be performed. Depending on the type of operation, the next activation either needs a data to be written, or produces a data read.

We assume that the memory component reproduces on its output control port `wsh` the value received on the data input port `rw`.

The components representing the processors have the same interface and the same protocol. The data inputs and outputs are as follows: `rd` is a data read, `a` is an address, `wd` is a data to be written, `req` is the request for the bus, `g` is the access-granted information from the bus, and `res` is the information from the memory, via the bus, signaling that the requested transfer is finished. Figure 21-(a) gives the protocol of the processor components. For the first activation, the processor performs some computation, and finally delivers a memory access `req1` request and an address `a1`. The next activation needs the grant input; depending on the type of operation stored in α, it either needs a data `rd1`, or delivers a data `wd1`. It cannot engage in another memory access before being activated with `res1` available.

Such a processor component reproduces on its output `wsh` the value produced on the data output port `req`.

The bus `B` manages concurrent accesses to the memory; it also delivers an output control port to define whether the current memory access is a read or write. Figure 21-(c) gives the protocol of the bus component. In each state, the bus may receive a request from any of the processors. It may either proceed with this request, if it is available (states 6, 7 for processor 1, and 2, 3 for processor 2), or remember this request for later treatment, if it is already engaged in a data exchange (in states 4, 5, processor 1 waits; in states 8, 9 processor 2 waits). Starting from the states where no processor is waiting, the bus terminates the data exchange, and goes to its initial state. If a processor is waiting, it will be taken into account just after the ongoing data exchange is finished.

Notice that, if they were more than 2 processors connected to the bus, the protocol could no longer be described with explicit states for the processes that are waiting. We should then use a protocol language allowing more easy-to-use data structures like FIFOs.

The bus component reproduces on its control output port `wsh` the value produced on the data output port `rw`.

The controller The *hardware transaction-level simulation* controller needed at this level activates one processor at a time, to simulate the asynchronous behavior of the two processors. The controller of Figure 24 is a possible description.

It is mainly an *interpreter* of the components' protocols. It maintains a memory of the protocols' states, initialized with the initial states of these protocols. For each `GO`, it chooses one of the components randomly. from the current state of this protocol, several transitions exist. The controller chooses (if possible) a transition for which the input data requirements are met. It provides the component with its required inputs, taking them in the components that produce them, via the FIFOs associated with the wires. `expliquer remove`.

Then it activates the component with `go`, stores its control outputs in its memory, and remembers that all its data outputs are now available. Finally, it changes states in the protocol, according to the transition chosen.

Guidelines for a hardware model The first level of hierarchy described above shows that for modeling hardware we need to have consistent components, protocols, and controller. The controller ignores the details of the components; it only knows about their protocols. The components expose in their protocols how they should be activated to produce their output control and data ports, and what input data they consume for that. The last important point in this modeling framework is a guideline for writing components: they should always reproduce on their `wsh` port the information about the type of data exchange they are engaged in, which is given either by a data input or by a data output.

3.3.2 Example, details of the processor components

The model of processor 2 is very abstract. We do not give more details than what is specified in its protocol. Consequently, it simulates the behavior of a processor that makes read/write transactions with the memory non-deterministically.

The model of processor 1 is further detailed (see Figure 20). The way transactions happen is determined by the embedded software. We consider a simple synchronous software component `synch`, which needs inputs and provides outputs. The component `wrap` deals with the memory accesses.

The protocol of the `synch` component is the same as in section 3.2, for the components whose output may depend on the input instantaneously.

The protocol of the `wrap` component (see Figure 22) describes the access to the bus both for the READ and for the WRITE operations (request, wait for grant, and then wait for the end of the data exchange).

At this level, the 42 controller is not a simulation controller (see figure 23). It is some abstraction of what happens in a real processor when it runs a piece of embedded software, taking its inputs from the memory, and writing its outputs to the memory. This MoCC shows how the processor writes to, and reads from the memory; it also transmit the data to and from the software.

The controller describes a cyclic behavior: first a read access, then a write access, and so on. Each access needs three activations, hence the controller has six states. State 0 describes the request for a memory read; in state 1 we wait for the grant access; in state 2 we wait for the end of the memory read; state 3 prepares the memory write; state 4 waits for the grant access, and computes the output by activating the software; in state 5 we wait for the end of the write access.

3.3.3 Comments

This example illustrates how we deal with heterogeneity in 42, with hierarchic levels, each level using a particular MoCC. The example could be extended by describing several processes on the same processor. This could be done in several ways: either with a level of the hierarchy where the controller describes a scheduler; or with an explicit component representing the operating system (in some abstract way), and a controller describing the interaction between an OS and the processes.

The example chosen is representative of a large class of embedded systems. This kind of applications is very similar to those used in automotive applications where different *ECU*s, or *Electronic Computing Units* are distributed and communicate over a particular communication medium. The *ECU*s may be hardware blocks or processors running several processes.

The type of system-level model we describe in 42 for this kind of embedded system has one major use. It allows to observe the interleavings of the software activities that are produced via their execution on a particular hardware platform. This is something that cannot be captured by very abstract semantic models like pure synchronous or pure asynchronous process compositions; for instance the precise way in which the software components access the memory, and the order in which it happens, depend a lot on the behavior of the hardware platform. Notice that the details we gave in our model of the bus places it in a more precise category than the so-called TLM-programmer's view advocated for SoC design (see [14] for a discussion on these levels).

4 Semantics of the Controllers

In this section, we formalize the components, the architectures and the controllers. Formalizing the protocols and the compatibility notions of section 2.5 is given in next section.

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. This formal semantics allows a precise definition of the controller actions, independently of any concrete programming style. In particular, it defines the life span of all the variables involved.

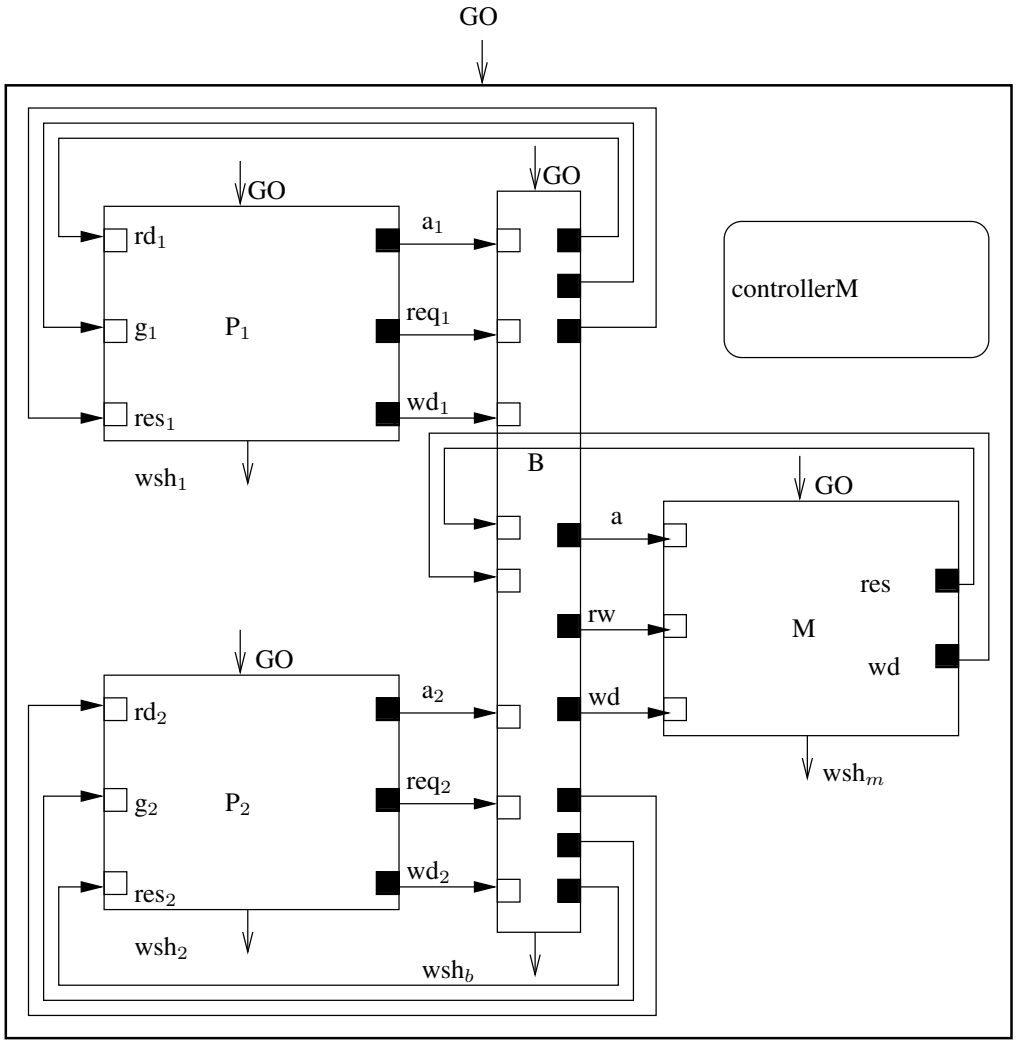

Figure 19: A multiprocessor system

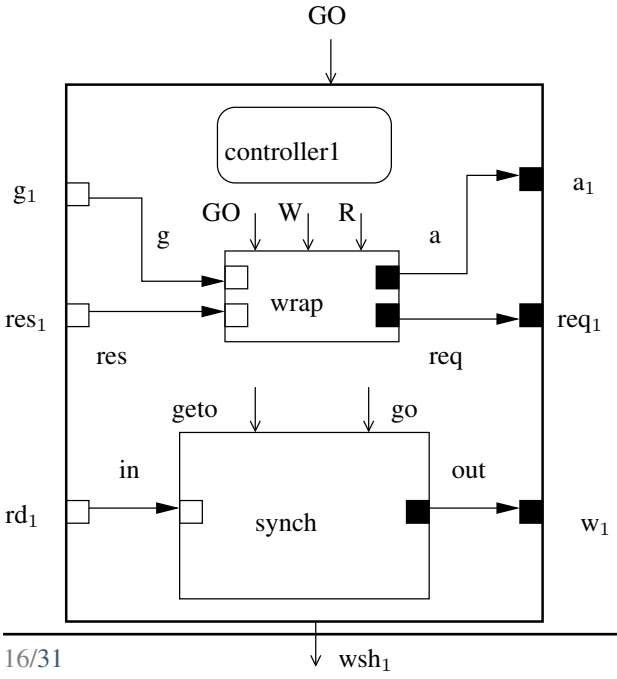

Figure 20: Details of component P1

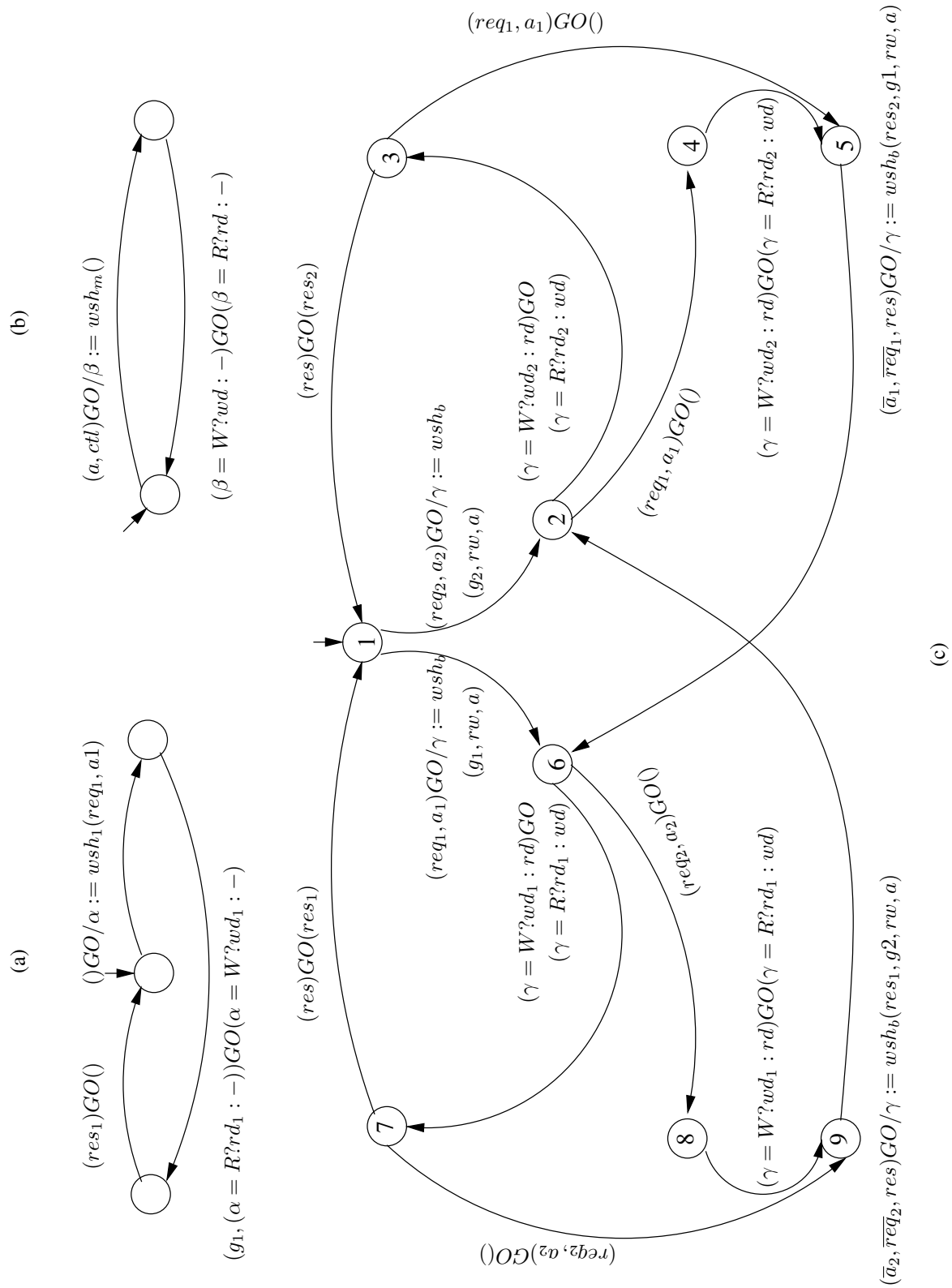

Figure 21: Protocols of the components at the highest level of hierarchy. (a) processor components; (b) memory; (c) bus.

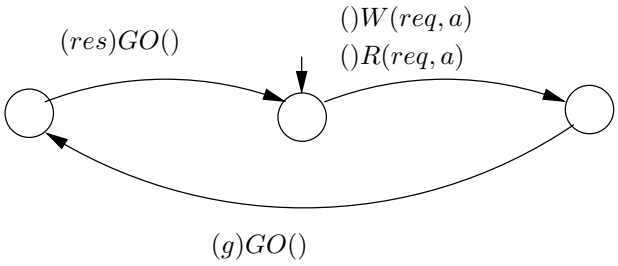

Figure 22: The wrapper protocol

```
Controller1 is
   var state : int;
state := 0;
for GO do : {
 a, in, out, res, req, g : FIFO(1,int);
 switch (state){
  case 0 :
     wrap.R; wh1:="R"
     a.put; a.get; req.put;
     req.get; state :=1;
  case 1 :
     g.put; g.get ;
     wrap.go; state := 2;
  case 2 :
     res.put; res.get;
     Wrap.go; state := 3;
  case 3 :
     wrap.W; wh1:="W"
     a.put; a.get; req.put;
     req.get; state :=4;
  case 4 :
     g.put; g.get ; wrap.go;
     in.put; in.get;
     synch.geto; synch.go;
     out.put; out.get;
     state := 5;
  case 5 :
     res.put; res.get;
     Wrap.go; state := 0;
}}
```

Figure 23: Controller for the processor component P_1

```
ControllerM is {
tab  ={ (P1,p1), (P2,p2), (B,b), (M,m)} // set of (Component,Protocol)
available = {}    // The set of wires whose producers have available
                  // new values in their output ports
T: Transition ;   // complete transition of a protocol
memory : some_type ;  // variables necessary to store the
                      // control outputs of the components

for GO do : {
      i := random(1,4);              // a random value in 1..4
      (cpt,pro) := tab[i];           // the component i and its protocol
      // try to find a fireable transition sourced
      // in the  current state of the protocol:
      T := pro.Transition () where(for each id in T.required,
                                   id in available);
      // if such a transition exists:
      if (T <> NULL) {
             for each id in T.required {
                    id.put; id.get;
                    available.remove (id);
             }
             cpt.go ;
             // update memory according to the control output:
             memory.update (oc) ;
             for each od in T.provided { available.add (od);}
             // take the transition of the protocol :
             pro.move_to_next_state(T.target);
}}     }
```

Figure 24: Controller for the highest level of hierarchy

4.1 Individual Components and Architectures

A 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 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: $L \subseteq (\bigcup_I OD_i) \times (\bigcup_I ID_i) \cup ID_g \times (\bigcup_I ID_i) \cup (\bigcup_I OD_i) \times OD_g$. Note that $(x, y) \in L \wedge (x, z) \in L \Longrightarrow y = z$ because links are point-to-point. Similarly $(y, x) \in L \wedge (z, x) \in L \Longrightarrow y = z$. The input and output control ports are implicitly linked to the controller. □

4.2 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. □

4.3 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 $(\Sigma_C, \Sigma_C^{\text{init}} \subseteq \Sigma_C, \Sigma_c^{\text{final}} \subseteq \Sigma_C, IC_g \longrightarrow \text{Progs}, S)$ 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 $(T_{\text{put}}, T_{\text{get}}, T_{\text{act}}, F)$.

4.3.1 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)`. Similarly, the method `get` will be represented by a function get : $M \longrightarrow \mathcal{D} \times M$.

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

4.3.2 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.

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

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, σ_P and σ_L: the consumer port P_2 of link ℓ receives the value taken from the link.

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

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.

$$\frac{\begin{array}{c}(\sigma_C, ic_\gamma,\ \sigma'_C) \in T_{\text{act}},\\ \exists vod, voc, vid, \sigma'_\gamma \text{ such that } (\sigma_\gamma, vid, vod, voc, \sigma'_\gamma) \in \mathcal{B}(ic_\gamma)\\ \text{and } \forall x \in \mathit{dom}(vid).vid(x) = \sigma_P(x)\\ (\sigma'_c, voc, \sigma''_C) \in S\\ \sigma'_P = \sigma_P[vod(x)/x][voc(y)/y], \forall x \in \mathit{dom}(vod), \forall y \in \mathit{dom}(voc)\end{array}}{\begin{array}{c}(\sigma_C, \sigma_I = (\sigma_1, \sigma_2, ...\sigma_\gamma, ..., \sigma_n), \sigma_P, \sigma_L) \longrightarrow\\ (\sigma''_C, \sigma'_I = (\sigma_1, \sigma_2, ...\sigma'_\gamma, ..., \sigma_n), \sigma'_P, \sigma_L)\end{array}} \quad \text{[act]}$$

The transitions in $\mathcal{B}(ic_\gamma)$ that can be taken are those whose input valuation $vid : ID \longrightarrow \mathcal{D}$ corresponds to what's available in the ports. σ''_C is the modification of σ'_C by storing the values of oc; In σ'_P the outputs ports in $dom(voc) \cup dom(vod)$ are modified according to the valuations vod and voc of the transition; there are the ports on which the component writes during the transition.

4.3.3 Macro-steps

The global component is of the form $(\Sigma_g, \Sigma_g^{\text{init}}, IC_g, OC_g, ID_g, OD_g, \mathcal{B}_g)$.

$\Sigma_g = \Sigma_C \times \Sigma_I \times \Sigma_P$, where Σ_I represents the states of the components, and Σ_P the state of the ports (see section 4.3.1). Notice that the state of the links does not appear here, because the links' values are not persistent across global activations.

$\Sigma_g^{\text{init}} = \Sigma_C^{\text{init}} \times \prod_I \Sigma_I^{\text{init}} \times \sigma_P^{\text{init}}$. The initial configuration is made of the initial state of the controller, the initial states of the components, and some initial state for the ports, that we may leave undefined. Indeed, 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. Hence $\forall p.\sigma_P^{\text{init}}(p) = ??$.

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 σ_C^n of the controller, via the function F of the program associated with ic_g.

The states of 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).

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

5 Semantics of the Protocols

5.1 Formal Definition of the Protocols

The protocols used in the examples are finite-state automata, whose transitions are labeled with various elements, including conditional data dependencies. The variables denoted by Greek letters are used to store the values of output control information, and may be used in conditions later on in the protocol; let us note $\mathcal{V}$ the set of such variables, and use $\mathcal{D}$ as their domain. See figure 25-(a) for an example.

For a component $C = (\Sigma, \Sigma^{\text{init}}, IC, OC, ID, OD, \mathcal{B})$, the protocol is an automaton $P = (S, S^{\text{init}} \subseteq S, \mathcal{V}, IC, OC, ID, OD, T \subseteq S \times \text{LAB} \times S)$. LAB is the set of transition labels, of the form: $([C](CI)ic/AOC(CO))$ where:

- $[C]$ is a condition on the variables of $\mathcal{V}$, i.e., a function $(\mathcal{V} \longrightarrow \mathcal{D}) \longrightarrow B$
- (CI) is the description of conditional input data dependencies; since we need to distinguish between: the inputs we need, for which we can accept a fresh value, and the inputs we need, but without being able to accept a fresh value, the set of input variables is $ID \cup \overline{ID}$. For each i among those variables, and for each valuation of the variables in $\mathcal{V}$, (CI) tells whether i is needed. Hence (CI) is a function $(\mathcal{V} \longrightarrow \mathcal{D}) \longrightarrow (ID \cup \overline{ID}) \longrightarrow B$.
- ic is a control input in IC
- AOC describes the control outputs and the way they are stored in the variables of $\mathcal{V}$. It is a set of tuples (α, oc) where $\alpha \in \mathcal{V}$ and $oc \in OC$.
- (CO) describes the conditional outputs; it is a function $(\mathcal{V} \longrightarrow \mathcal{D}) \longrightarrow OD \longrightarrow B$.

Using the variables in $\mathcal{V}$ is a convenient way to write protocols, but if the types of the output control variables are finite, this does not add to the expressivity of the protocol language. In order to express the compatibility relation between the protocols and the controller, we first *expand* the protocols so that the

variables in $\mathcal{V}$ disappear. The simplified protocol is also a state machine where the valuations of $\mathcal{V}$ are added to the state: $P' = (SV = S \times (\mathcal{V} \longrightarrow \mathcal{D}), S^i \subseteq SV, IC, OC, ID, OD, TV \subseteq SV \times \text{LAB'} \times SV)$, but now the labels LAB' are of the simpler form $(I)ic(O)$ where $ic \in IC$, $I \subseteq ID \cup \overline{ID}$ and $O \subseteq OD$.

The rule [exp] describes the expansion of a general protocol P into a simpler protocol P'. It gives the transitions of P' in terms of the transitions of P. See Figure 25-(b). Notice that it generates one transition per value in the type of each control output, leading to a combined state in which these values are stored.

The initial states are defined by: $S^i = S^{\text{init}} \times (\mathcal{V} \longrightarrow \mathcal{D})$. Any valuation of the variables in $\mathcal{V}$ may be considered, since there should be no read operation on such a variable before it has been assigned a value.

$$\frac{\begin{array}{l}(s, [C](CI)ic/AOC(CO), s') \in T \\ C(\sigma) \text{ is true} \\ I = \{id \mid CI(\sigma)(id)\}, \quad O = \{od \mid co(\sigma)(od)\} \\ \sigma' = \sigma[d_1/\alpha_1, d_2/\alpha_2, ...d_n/\alpha_n] \text{ where} \\ AOC = \{(\alpha_k, oc_k)\}_{k=1..n}, \text{ and } \forall k \in 1..n, d_k \in \textit{type\_of}(oc_k)\end{array}}{((s, \sigma), (I)ic(O), (s', \sigma')) \in TV} \quad \text{[exp]}$$

5.2 Compatibility Semantics

The compatibility is a relation between:

- A set of components $\{C_i = (\Sigma_i, \Sigma_i^{\text{init}}, IC_i, OC_i, ID_i, OD_i, \mathcal{B}_i)\}_I$
- The set $\{P_i\}_I$ of their simplified protocols
- An architecture $A = (IC_g, OC_g, ID_g, OD_g, L)$
- A controller $(\Sigma_C, \Sigma_C^{\text{init}} \subseteq \Sigma_C, \Sigma_c^{\text{final}} \subseteq \Sigma_C, IC_g \longrightarrow \text{Progs}, F, S)$

It expresses that, given the way the components are assembled with the architecture, and given their protocols, the controller is *correct*, i.e., it uses the components according to their protocols. Since the controller activity happens only in response to the activations of the global component (the big box around the C_is), we may define two compatibility relations: either we require that the controller respect the protocols P_is *for any sequence of global activations*, or we require that it respect the protocols *only for those sequences of global activations that are indeed allowed by the protocol of the global component*. The second one is weaker than the first one. We define the strongest one below.

The definition of the compatibility relation can be split into two sub-properties, that should be both true. Let us consider a sequence of macro-steps S_M, and the sequence S_m of the micro-steps and FIFO management operations that are produced by the controller as a response to S_M.

- first, each protocol P_i can be seen as a recognizer of the language of correct activation sequences of C_i; we can just erase anything in the labels of P_i but the control input, and we get a *language recognizer* in the form of a non-deterministic finite automaton A_i, on which all states have to be considered as accepting states. The words to be tested are obtained by keeping only the activations of C_i, in the sequence S_m. For each possible S_M, the corresponding S_m projected on the activations of C_i has to be accepted by A_i, for all C_i.
- second, the protocol P_i can be seen as a *transducer*, taking a sequence S_m, and producing a sequence of input port assignments and required inputs. The operation `get` on the FIFO associated with the wire connected to the input port `a` is considered as an assignment to this port (we note it `geta` in figure 26). The required inputs are taken from the transitions of P_i, in the order in which they are fired by S_m.
  The controller respects the protocol of C_i is, for any S_M, the result of the transducer is accepted by the language recognizer of figure 26. This automaton expresses the temporal property: an occurrence of `geta` cannot be followed immediately by an occurrence of $\overline{\texttt{a}}$; there should be an occurrence of `a` in between.

6 Related Work and Main Choices for 42

We already mentioned some related work in the introduction. A lot of approaches have been proposed for heterogeneous modeling frameworks based on the notion of model of computation and communication.

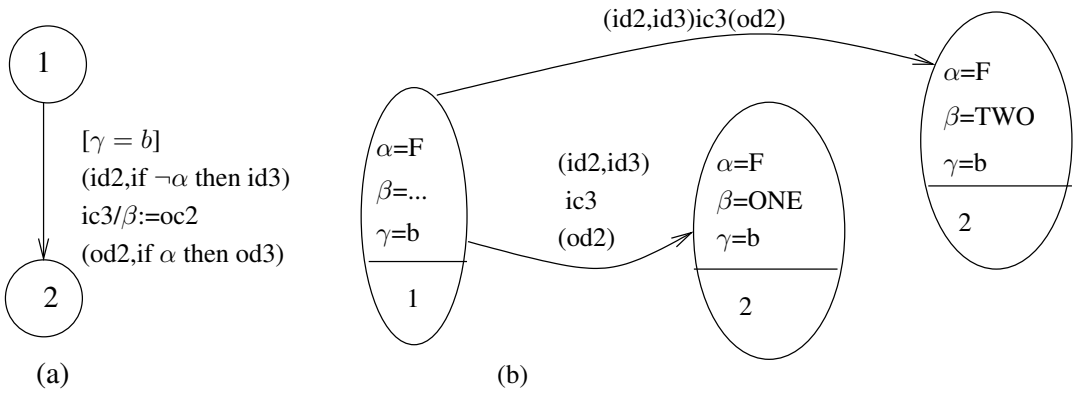

Figure 25: An example protocol: (a) simple form; (b) expanded form, if the type of `oc2` is { `ONE, TWO` }; α is a Boolean, and γ any type.

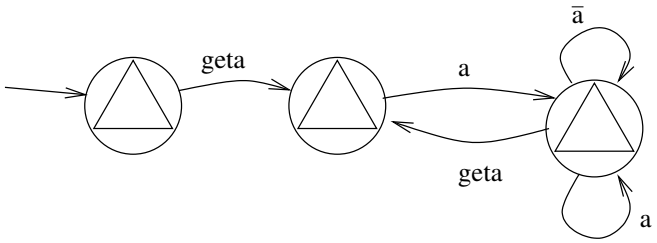

Figure 26: Recognizing correct sequences of input port assignments and required inputs

Moreover, related work includes component-based approaches in other domains than embedded systems, contract-based specification of all kinds, etc. We first give a (non-exhaustive) list of such works, and then we insist on some of the choices that have been made for the definition of 42.

6.1 Related Approaches

6.1.1 Component-Based Modeling or Programming Frameworks

ForSyDe (Formal System Design) [43] uses various MoCCs for the various modeling and design phases of embedded systems; the way MoCCs interact is not formalized; the initial specification model uses a synchronous model of computation. In 42, the choice of the MoCC, depending on the modeling phase, is not decided a priori.

Other proposals, like the reactive modules [3], 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.

There are a lot of approaches for software components, not dedicated to embedded systems. Some models are hierarchical models such as Fractal [18] and SOFA [38]. CORBA is a software architecture where heterogeneous (programming language and execution platforms) are deployed. Models such as Enterprise Java Beens(EJB) [17] and CORBA Component model (CCM) [12] do not allow for hierarchic organization of components.

6.1.2 Contracts

Component-Based approaches for embedded systems often classify contracts into four types: basic, behavioral, synchronization, and quality-of-service contracts. Basic contracts are concerned with the types of

the ports; behavioral contracts may specify sequences of activations; synchronization contracts are not always clear; they depend on the synchronization primitives of the underlying component language. Finally, quality-of-service contracts may equip components with a specification of what resources they need, and what resources they consume.

A contract-based component model for embedded systems is presented in [25]. It targets specification of embedded software using contracts, including all of the four kinds of contracts, but has limited MoCCs. 42 deals with the first three types only.

Protocols, usually given as state machines, are used to specify components with a behavioral description. In SOFA, components come with a behavioral specification [39]. Each transition is labeled by a message suffixed by a symbol defining whether the message is passed or received.

Other approaches (see, for instance [15]) 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.

The PACC (Predictable Assembly from Certifiable Components) initiative [35] has the same objectives of predicting the behavior of a component-based system, based on known properties of components.

Session types [46] express sequential (or behavioral) contracts as types.

6.1.3 Expressing the Semantics of, or Programming MoCCs

[23] 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.

The most relevant and recent work in this category is the family of TAG semantics [6]. 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 precise comparison between 42 and the TAG semantics is still to be done.

Coordination languages are used to describe systems of parallel software entities. We can distinguish between data-oriented ones such as Linda [16], and control-oriented ones such as the very first definition of the Architecture Description Language Darwin [34]. They can be compared to the controller in 42. They are not associated with a clear hierarchical component-based framework, however. A more recent definition of Darwin [29] is a configuration language that allows for hierarchical description of component-based distributed systems.

BPEL (Business Process Execution Language) [8] is used to define business process behaviors based on Web Services; it is executable. BPEL describes asynchronous processes, and insists on the specification of Web services so that they can be used as components.

SML-sys [31] (not to be confused with SysML) is a framework based on the functional programming language standardML, to model heterogeneous models of computation. It is very close to 42 controllers.

6.2 Comments on specific points

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.

6.2.1 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) or SystemC-AMS concern the modeling of mixed digital-analog systems, but they do not address the component aspects. Moreover, they concentrate on the collaboration between a numerical solver and a discrete simulation engine, from a quite operational point of view, without trying to define the *semantics* of this heterogeneous combination. Similarly, Matlab/Simulink designs can

mix continuous and discrete parts, but the notion of a component is not dealt with specifically. In both cases, if the collaboration between a numerical solver and a discrete simulation engine involves a fixed-step sampling of the continuous part, the result can be expressed easily in a discrete framework, where one of the components is a discretized version of a continuous object; this is what we provide with 42. Nevertheless, there is a need for mixed discrete/continuous models, but the problem is the semantics, not the implementation.

6.2.2 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 [9], and to some extent SystemC-TLM [19] (used for the description of systems-on-a-chip at the so-called transactional level), but not for other component models like BIP [5], 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. In some formalisms developed in the architecture description languages community, there is also a clear distinction between the set of elementary components, and the object obtained by combining them with an ADL. Sometimes there is no way to consider that we can *"close the box"* around this assemblage, obtaining a new 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. Moreover, the hierarchy is essential for the modeling of heterogeneity, since we do not allow to use several MoCCs at the same level.

6.2.3 Oriented connections vs non-oriented ones

42 adopts a dataflow style architecture description language, with oriented connections. In Ptolemy, in the modeling tool Spice [44] for electronic circuits, or in the bond graph formalism [26], this is not necessarily the case, allowing the modeling of various physical 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 [40]). 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. In particular, *time* is essentially an input (the system has no influence on it) and is naturally modeled as an input. Hence we choose oriented wires.

6.2.4 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 3.2, which corresponds to what electricity does in synchronous circuits). 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.

6.2.5 Do connections express some behavior?

In 42, the connections only express that some information may flow from one component to another. There is no synchronization nor 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. [23] 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 [19], 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. Notice, however, that in SystemC, the meaning of the connection between ports is built in the definition of the SystemC execution engine, while in 42 it is programmed freely in the controller.

6.2.6 Architecture Description Languages and MoCCs

In 42, the ADL 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 1.4, Ptolemy is more liberal. For 42, we chose only one style of architecture description language, because we want a simple formal semantics.

6.2.7 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 that we should be able to consider its activations as 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.

Atomicity is also related to the notion of "fresh" inputs. In 42, we are able to express that a component needs several steps to finish dealing with a set of inputs, before being able to accept new ones.

6.2.8 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 called *multi-form time* first introduced by G. Berry for Esterel [7]: physical time is nothing special but a sequence of events (seconds, milliseconds, ...), and any sequence of events (meters, or beacons on a track) can be considered as a time scale for the reactive system that perceives these events. Using timed automata [2] 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, ...).

In 42, time is discrete and logical; if "real time" is needed, it is represented by a particular clock input. This allows descriptions of distributed systems in which the various clocks of the processors may be unrelated (or partially constrained) clock inputs. It is very important to be able to de-correlate the clocks of the computing units in a distributed system, to represent systems faithfully. A formalism where time is a single dedicated event, like timed automata, makes it quite difficult to describe systems with *several* time events.

6.2.9 Specifying components, and the notion of a contract

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 [1]), 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.

42 could be equipped with data contracts, following the ideas of [30]. But the most interesting extension is towards quality-of-service contracts (see conclusion).

7 Conclusion and Further Work

7.1 Summary on 42

We have defined the general framework 42 for component-based modeling of heterogeneous systems, and shown its use for Kahn Networks, pure synchronous systems, or heterogeneous systems. Heterogeneity is dealt with as in Ptolemy, by using various MoCCs at the various levels of hierarchy. The synchronous example shows that we can include hardware descriptions in a 42 model. The heterogeneous example shows that 42 may be used to show how an quite abstract model of a hardware architecture (very much in the spirit of "transaction-level-modeling) can be mixed with a detailed view on several pieces of synchronous software that run on separate processors.

42 is meant to be used as an abstract description level for describing the concurrent and timed behavior of heterogeneous embedded systems components. It concentrates on a precise modeling of logical 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. It does not deal with quality-of-service, or performance evaluation, of embedded systems.

42 is not meant to become another language for the programming of embedded systems. We concentrate on system-level descriptions. We are developing a proof-of-concept tool for allowing graphical descriptions of 42 architectures, specification of controllers, and the use of existing code as a 42 components (for instance, the C code produced by the Lustre compiler, or the code of a thread in java, with appropriate wrappers). The tool will allow the simulation of a 42 model by interpreting the controller code, and it will propose a number of verifications based on the definition of the protocols.

7.2 Further Work

Concerning *semantical issues*, we will compare our quite operational semantics with the family of TAG semantics [6]. 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. Expressing the semantics of MoCCs by operational means has been proposed by several teams. It is related to the idea of "abstract semantics", in which the semantics of MoCCs could be described without taking concrete language details into account, as the abstract syntax is used to forget about concrete syntax details.

In 42, the language of the controller plays a central role, with respect to the definition of (abstract) semantics. We have to study how to characterize the expressive power of the controller. For the moment, we have tried to characterize the type of memory it needs, but we should also look at its general expressive power.

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.

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 [41] that has been defined for the generation of test scenarios or the simulation of non-deterministic systems, could be an idea. It could also answer the above question about expressing parallelism in the controller: potential parallelism is easier to describe in a constraint-based framework than in some imperative style.

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 [42]:

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. We will first look at MPA (Modular Performance Analysis) [13]. Some recent work [37] has shown that the traditional models of performance analysis should be enriched with behavioral information; extending 42 with models of this kind could be a way of finding a compromise between very detailed behavioral models on which performances can be assessed by simulation methods only, and quite abstract analytical models, on which performances can be given by analytical solutions, but which are often overly approximate.

References

[1] 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. 6.2.9

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. 6.2.8

[3] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999. 6.1.1

[4] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto L. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003. 1.2

[5] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12. IEEE Computer Society, 2006. 6.2.2

[6] A. Benveniste, B. Caillaud, L. P. Carloni, and A. L. Sangiovanni-Vincentelli. Tag machines. In Wayne Wolf, editor, *5th ACM International Conference On Embedded Software (EMSOFT)*, pages 255–263, Jersey City, NJ, USA, September 2005. ACM. 6.1.3, 7.2

[7] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992. 6.2.8

[8] Business Process Execution Language for Web Services Version 1.1. `download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel`, May 2003. 6.1.3

[9] 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. 6.2.2

[10] 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. 1.4

[11] 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. 1.5

[12] Object Management Group: CORBA Components, v. 3.0, OMG document formal/02-06-65. 6.1.1

[13] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Design Automation and Test in Europe (DATE)*, pages 190–195, Munich, Germany, March 2003. IEEE Press. 7.2

[14] J. Cornet, F. Maraninchi, and L. Maillet-Contoz. A method for the efficient development of timed and untimed transaction-level models of systems-on-chip. In *Design Automation and Test in Europe (DATE)*, Munich, Germany, March 2008. 3.3.3

[15] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001. 6.1.2

[16] D.Gelernter and L.Zuck. On what linda is: Formal description of linda as a reactive system. In *Second International Conference on Coordination, Languages and Models (Coordination'97),*, Berlin, Germany, September 1997. 6.1.3

[17] Enterprise Java Beans specifcation, version 2.1. Sun Microsystems, November 2003. 6.1.1

[18] Fractal Component model. `fractal.objectweb.org/`. 6.1.1

[19] F. Ghenassia. *Transaction Level Modeling With SystemC: TLM Concepts And Applications for Embedded Systems*. Springer-Verlag, 2005. 1.1, 3.3.1, 6.2.2, 6.2.5

[20] 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. 1.3, 1.5

[21] N. Halbwachs, E. Jahier, P. Raymond, X. Nicollin, and D. Lesens. Virtual execution of AADL models via a translation into synchronous programs. In *Seventh International Conference on Embedded Software (EMSOFT 2007)*, Salzburg, Austria, September 2007. 1.5

[22] 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. 1.5

[23] 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. 6.1.3, 6.2.5

[24] C. Hewitt. A universal, modular actor formalism for artificial intelligence. In *Proc.International Joint Conference on Artificial Intelligence*, 1973. 1.4

[25] Shuyu Lin; Jian Wu; Zhengguo Hu. A contract-based component model for embedded systems. *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 232–239, 8-9 Sept. 2004. 6.1.2

[26] J. Thoma. *Introduction to Bond-Graphs and their applications*. Pergamon Press, 1975. 6.2.3

[27] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974. 1.8

[28] 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. 1.3, 1.5, 2.4.1

[29] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin. 6.1.3

[30] 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. 1.5, 6.2.9

[31] A. Mathaikutty, D, D. Patel, H, and K. Shukla, S. A functional programming framework of heterogeneous model of computation for system design. Technical report, Virginia Tech, FERMAT Lab, 2004. 6.1.3

[32] B. Meyer. *Eiffel: An Introduction*. Interactive Software Eng., 1988. 1.1, 1.7, 2.5.1

[33] R. Milner. A calculus of communication systems. In *LNCS 92*. Springer Verlag, 1980. 1.5

[34] N. Minsky and A. Borgida. The darwin software-evolution environment. *SIGPLAN Not.*, 19(5):89–95, 1984. 6.1.3

[35] Predictable Assembly from Certifiable Components, PACC Initiative. `www.sei.cmu.edu/pacc`. 6.1.2

[36] Pierre G. Paulin. Automatic mapping of parallel applications onto multi-processor platforms: A multimedia application. In *DSD*, pages 2–4. IEEE Computer Society, 2004. 1.7

[37] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael Gonzlez Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *ACM & IEEE International Conference on Embedded Software (EMSOFT)*, pages 193–202, Salzburg, Austria, October 2007. ACM Press. 7.2

[38] F. Plásil, D. Bálek, and R. Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 43, Washington, DC, USA, 1998. IEEE Computer Society. 6.1.1

[39] S. Plasil, F.; Visnovsky. Behavior protocols for software components. *Software Engineering, IEEE Transactions on*, 28:1056– 1076, 11 Nov 2002. 6.1.2

[40] 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. 6.2.3

[41] 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. 7.2

[42] Ludovic Samper, Florence Maraninchi, Laurent Mounier, and Louis 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. 7.2

[43] 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. 6.1.1

[44] Spice. `bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/`. 6.2.3

[45] Systems modeling language (SysML) specifications. `www.sysml.org`, nov 2005. 1.2

[46] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of software components using session types. *Fundam. Inf.*, 73(4):583–598, 2006. 6.1.2

[47] Jan van den Bos. PROCOL: A protocol-constrained concurrent object-oriented language. *Information Processing Letters*, 32(5):221–227, 1989. 2.4.1