Architectural Techniques for the Description and Validation of Distributed Real-Time Systems

Laurent THOMAS[*]
Thomas LAMBOLAIS
Renaud LESIOUR
THOMSON-CSF Corporate Research Laboratory
Domaine de Corbeville, 91404 ORSAY Cedex, France
and
Adèle COVADIS[†]

Abstract

Our goal is to settle techniques for the validation of real-time distributed systems. We opted for executable models and simulation techniques. Our approach is to work at the architectural level of a system abstraction. The software architecture is understood as a set of structures comprising software components with external properties and relationships among them. We devised a language to describe those components and the logical structure of the architecture. It is a specialization and extension of object-oriented modeling languages designed to deal with concurrency, reactivity and architectural concerns. This language is based on a notion of components defined similarly to the class notion, extended for the timing requirements. Execution and simulation are driven to validate behaviors with respect to desired properties.

1. Introduction

A real-time computer system is a computer system which must produce its result within specified time intervals. To be acceptable, such systems must not only be functionally correct — that is produce correct results— but must also be temporally correct. The correct behavior of a real-time system may depend not only on the timing of its output, but also on time-varying functions of its inputs. Customers and developers need compelling evidence that these systems deliver results on time.

The early phases in any system development are by far the most critical. A very high percentage of the most serious errors are requirements errors. This is because, in the early phases, much is left to the designer's intention and common sense. As a result, early evaluation techniques such as rapid prototyping and *a priori* validation are needed to allow stakeholders to analyze an abstract system.

We are interested in reuse and *a priori* quality evaluation in order to know the most we can about a system before it is effectively built. Suppose a module, say a *Tracks Manager*, has been developed for an air traffic management system where one client's requirement was to handle 200 tracks[1] per second, then we want to be able to decide before integration if it is possible to reuse this *Tracks Manager* module in another system where the requirement has become 300 tracks per second. Obtaining the requirement may not only depend on the *Tracks Manager* itself, but also on communicating links performance, other modules and so on. The direction we chose is an architectural description approach. More evidence on why software architecture is a possible approach is given in section 2.

Taken for granted that the architecture level of description may help in answering the previous needs, our aim is thus the description and validation of software architectures in the domain of distributed real-time applications. We developed a software architectural specification language, *Ardeco*, for the descriptive part, and applied simulation techniques for the validation part. Simulation is a technique for analyzing the behavior of systems. It is a form of testing applied to a formal specification. To initiate simulation, the user provides a sequence of input events, generated either manually or automatically, to the simulator. The simulator then executes the system symbolically, using the system

[*] COVADIS Project Leader

[†] COVADIS partners are Charles André and Marie-Agnès Péraldi (I3S, University of Nice), Mireille Bayart and Chris Choukair (LAIL, University of Lile), Anne-Marie Deplanche and Yvon Trinquet (IRCyN, University of Nantes), and Françoise Simonot (LORIA, University of Nancy).

[1] Numbers have no real significance.

specification to determine the system's response to each input. By means of simulation, the user can investigate many different system executions, each derived from the (formal) specification, to check that the specifications capture the intended behavior.

The paper is organized as follows. Section 2 introduces the notion of software architecture and benefits expected of its use. Section 3 presents our software architecture description language. Section 4 explains how this description is used to validated stakeholder's requirements.

2. What is expected of software architecture?

We follow the definition of [2]. "*The software architecture of a program or a computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.*"

Architecture (from now on, architecture and software architecture will be used synonymously) is an abstraction of systems: it defines components and how they interact, and it ignores purely local information about components. It must be noted that systems have many structures and no single structure can be said to be the architecture: examples of common and useful software structures are module structure, for work assignments, logical structure, for abstractions of functional requirements, process structure, to deal with dynamic aspects, or physical structure to handle the mapping of software onto hardware. Properties of components (in the sense of [2]) are assumptions that one component can make about another: provided services, required services, performance characteristics, shared resource usage, fault handling, etc. Relationships among components may be runtime (often called connectors in the literature) or non runtime (*is a submodule of, inherits from*, etc.).

The architecture level provides a vehicle for communicating among stakeholders. It is also a transferable and reusable abstraction of a system. It is the manifestation of the earliest design decisions about a system. Because it influences qualities in known ways, it follows that architecture offers a good basis to analyze and predict how well some quality attributes will be achieved.

3. An Architecture Description Language

Ardeco focuses on the logical communication structure of a software architecture. It provides abstractions for the functional and time related behavior of the application by means of interacting components. The chosen architectural description may be unrelated to the way abstract entities will eventually be implemented.

A component is an instantiation of a component type, in the same way that objects are instantiations of classes. Architects describe or reuse component types and instantiate them to built an architecture. Components interact by emitting and receiving requests. The basic relationship between components is a perfect link: it never loses nor alter messages and transmission times are null. It should be seen like a "logical link" to indicate the target component of emitted requests. Moreover, it is a point-to-point link. Thus, to model time consumption, transmission failures, message dispatching or collecting, specifically devised components must be added.

We will first focus on the component type description and then the software architecture description.

3.1. What is a component?

We will use interchangeably component and component type when context makes it clear which one is intended. The main paradigms governing a component are the following:

Reactivity: A component maintains a continuous interaction with its environment, i.e. with the other components of the architecture. As defined by Pnueli [7], reactivity also implies bounded response times.

Homogeneity: As for Milner's CCS language [6], there is no distinction between active and passive components. Passive components, like buffers, must be described like other components.

A component type consists of two main fields which are:

- The component interface: the interface of a component defines provided and required services, with their signatures and causality relations. It also contains a list of available parameterized versions. At any time, a component is in one of its version. In each version the component may react differently to incoming requests and may create different spontaneous activities.
- The component behavior: it may be described, either by an architecture description of other components, or, at the lowest level, by a reactive behavior. This latter behavioral description is based on *activities* and *events*.

The behavior is given by a combination of *declarative* and *operational* descriptions. The declarative part of a component's behavior is in charge of handling requests to this component's services. It is a list a rules governing the events. For instance, a rule may defer the beginning of an activity or its termination[2].

The operational part describes how a component answers requests, mainly with respect to time. An activity may be

[2]When accepted, the request has created an activity to execute the operational part, but one rule in the declarative part may prevent the activity to begin until some condition is met.

created to execute the operational part for each accepted request. The operational part lists services either externally invocable or spontaneously activated by the component itself, and for each of them describes how it behaves wrt. sending and receiving requests and time spent in execution.

Figure 1 shows a component with input and output points, events (dashed lines) and activities. Note that *Ardeco* component descriptions are given in a textual way; figure 1 being only an illustration. Spontaneous activities can also be triggered, by time deadlines (as in [9]) or state conditions. On figure 1, ‘on a’ is the name of an activity associated with the provided service ‘a’, and ‘each δt’ is a self triggered activity. We see that there is a time lap between the acceptance event and the actual beginning of the activity. Although not shown by the figure for the sake of clarity, events managed by the activity manager are broadcasted to all activities.

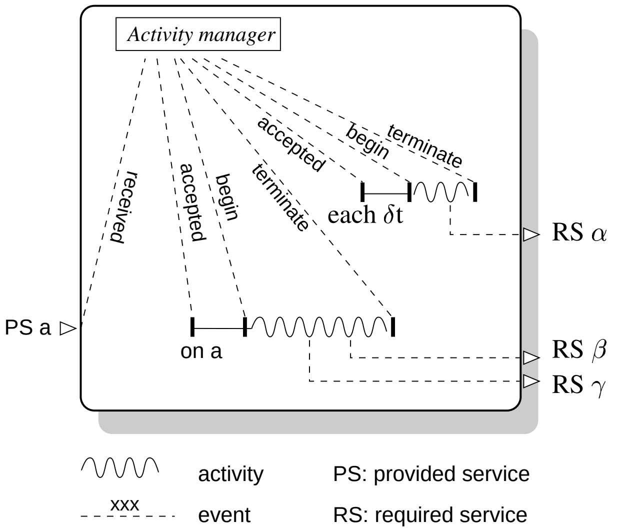

Figure 1. Activities and events generated inside a component.

Figure 2 presents the automaton following a request received by a component. It explains which internal events are generated by the outside request. Transitions are labeled by a 3-tuple formatted as follows: ‘[condition] received event/emitted event’ where any of the fields are optional, provided it remains at least one. A request is first *received* by a component (the received event is generated), it may then be *accepted* or not, it may then create an activity which will *begin* and finally *terminate* (either graciously or not, but it will always *terminate*).

The temporal ordering of activities is managed by declarative constraints. Activities never communicate between each other but instead generate events. These events are broadcasted within all the component. External stimuli also generate events.

It is also possible to attach a watchdog to an activity. As soon as a request is accepted, the watchdog is set and initialized to the value of a **delay**. If 0 is reached before any event given by **disarmed by** has been seen, the code associated to the **watchdog** is executed by the activity created to execute the request. This code may kill the current activity.

An activity is described by two kinds of primitives: temporal and non-temporal primitives. Only the temporal primitives can take some time, others being instantaneously performed. This approach, as well as the broadcast of events throughout a component, is borrowed from the synchronous approach [3]. There are two kinds of temporal primitives: ‘work during ⟨delay expression⟩’ or ‘wait for ⟨event expression⟩’.

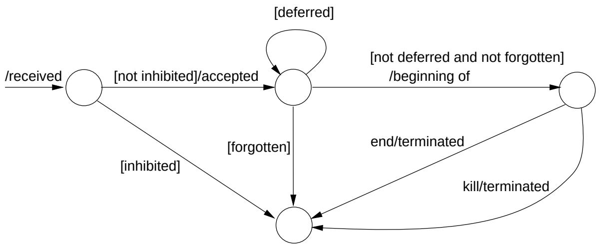

Figure 2. Automaton and generated events by a request reception.

3.2. Example of a component

Figure 3 illustrates most of the concepts used in *Ardeco*. An hypothetical coffee machine delivers coffee cups. Coffee may be strong, normal or light. They are freely delivered.

In the component code (fig. 3), circled numbers refer to explanation given below. Like the leading points, they are not part of the *Ardeco* syntax.

① This (strange) coffee machine offers 4 services (**input**) and has 2 requests (**output**). At any time it is in one of three versions (**versions**). When in `Distributing_Coffee` version, its service `ask_for_coffee` may send a request `deliver_coffee` on its required service.

② Let’s examine the behavior of this machine depending on its current version:

③ Version `Doing_Coffee` models *initialization* of the machine. It corresponds to the preparation of

```
component Coffee_Machine
① interface
. . input // also known as provided services
. . . ask_for_coffee(integer), ask_for_sugar, do_coffee(integer), temperature(real)
. . output // also known as required services
. . . deliver_coffee, verify_temperature
. versions
. . Doing_Coffee(t1,t2: integer); Cleaning;
. . Distributing_Coffee(n,t: integer) dependency ask_for_coffee requires deliver_coffee;
② behavior
③ . version Doing_Coffee(t1,t2)
❶ . . features // of Coffee_Machine in version Doing_Coffee
. . . . coordination
. . . . . exclusive do_coffee
. . . . operation
. . . . . inhibit do_coffee(n) when n ≤ 0
❷ . . triggers // of Coffee_Machine in version Doing_Coffee
. . . . on do_coffee(n) delay 10 disarmed by end of myself else e
. . . . . work during t1*n; become Distributing_Coffee(n,t2);
. . . . . watchdog e { kill; }
. . . . endon
④ . version Distributing_Coffee(n,t)
❸ . . data // of Coffee_Machine in version Distributing_Coffee
. . . . m : integer ← n; δt : integer = t;
. . . . min : integer = ...; max : integer = ...;
❹ . . exception // of Coffee_Machine in version Distributing_Coffee
. . . . f(){...}
❺ . . features // of Coffee_Machine in version Distributing_Coffee
. . . . coordination
. . . . . exclusive ask_for_coffee
. . . . . temperature excludes ask_for_coffee
. . . . operation
. . . . . forget ask_for_coffee(p) when p > m ∨ p ≤ 0 ∨ p ≥ 4
❻ . . triggers // of Coffee_Machine in version Distributing_Coffee
❼ . . . on ask_for_coffee(p) delay 5 disarmed by end of myself else e
. . . . . wait for acceptance of ask_for_sugar {/*pour sugar in coffee cup*/}
. . . . .                                                  during 2 at most;
. . . . . work during ...; m ← m - p; deliver_coffee;
. . . . . watchdog e { ... }
. . . . endon
. . . . on ask_for_sugar
. . . . . // do nothing
. . . . endon
❽ . . . verify : each δt
. . . . . verify_temperature; wait for beginning of temperature(_) { ... }
. . . . .                               during 2 at most or else raise f;
. . . . endeach
❾ . . . on temperature(t)
. . . . . if t ≤ min ∨ t ≥ max then work during ...; fi;
. . . . endon
❿ . . . empty : when m = 0
. . . . . become Cleaning;
. . . . endwhen
⑤ . version Cleaning
. . . data // of Coffee_Machine in version Cleaning
. . . . b: boolean ← true;
. . . triggers // of Coffee_Machine in version Cleaning
. . . . once : when b
. . . . . b ← false; work during ...;
. . . . endwhen
end Coffee_Machine
```

Figure 3. An *Ardeco* component modeling a coffee machine.

the machine by an operator, filling it with ground coffee and water for instance. In this version only the `do_coffee` service is defined (see ❷). Any other request will not be serviced.

❶ The **features** keyword introduces the declarative part of the current version. Here it says first that no two `do_coffee` services may be handled concurrently. Thus, an incoming `do_coffee` request while one is executing will be deferred until the first finishes, it will not be directly rejected. Secondly, no request for a `do_coffee` with a negative parameter will be accepted.

❷ The **triggers** section introduces the operational part of the component's behavior in this version. Here we ask the time between the acceptance of `do_coffee` and the end of the answering activity not to exceed 10 time units otherwise the activity is killed, effectively preventing the become instruction. Time taken to initialize the machine is modeled by the work during instruction[3]. The become instruction asks the component to change version.

④ Version `Distributing_Coffee` models the delivering of coffee.

❸ The **data** part defines local variables initialized when entering the version. The variable 'm' is the current amount of coffee remaining in the machine while 'δt' is the (constant, hence = and not ←) frequency with which a spontaneous activity will be launched (see ❽). Constants 'min' and 'max' are the liquid's temperature interval outside which action has to be taken (trigger ❾).

❹ An exception code is defined. This exception may be raised by some activity either unconditionally or upon a missed deadline (see the wait for instruction at ❽).

❺ The declarative part says that an `ask_for_coffee` activity will be deferred if another `ask_for_coffee` or a `temperature` is active, and until this other activity terminates. Its **operation** part says that an accepted `ask_for_coffee` will be forgotten if 'p' is out of bounds. In fact, an `ask_for_coffee` may have been accepted and deferred for some reason, and during the time it was deferred the 'm' variable has been decremented such that the

[3] Instruction end is a semi-colon.

new value is now less than the 'p' parameter of the deferred `ask_for_coffee`.

❻ The operational part defines 5 activity codes, two of them being spontaneous (an activity created without external request).

❼ When a request to deliver a coffee is accepted, a watchdog is created and its delay is initialized to 5. It first waits for a request for sugar for 2 time units and then delivers the coffee with or without sugar. Time taken to deliver the coffee is modeled by the work for instruction. If `ask_for_sugar` is accepted during the wait for instruction, the local code {/*pour sugar in coffee cup*/} is executed (independently of the activity created to execute the code of the `ask_for_sugar` request itself). If no `ask_for_sugar` is accepted within this time bound, no local code is executed and no sugar is poured, which is the intended action. If a request `ask_for_sugar` is accepted and begins outside the waiting instruction of `ask_for_coffee`, an activity is created which does nothing. To prevent this, a rule may have been added to the **coordination** clause[4].

❽ This spontaneous activity is created every δt time units. Its rôle is to verify the temperature of the coffee. It asks it via a required service (a **dependency requires** `verify_temperature` may have been added to the declaration of the version `Distributing_Coffee` in the interface part) and then waits for an answer. If the answer comes back (in the form of the `temperature` request), a local code is executed (the dots between the brackets) and then finishes. Note that the `temperature` request has also created an activity whose code is at ❾. If no `temperature` request has been received within 2 time units the exception *f* is raised. This trigger is given a name (`verify`) for reference in a **features** clause.

❾ A `temperature` request is in fact a notification of the coffee temperature. If this temperature is outside some bounds (definition of these bounds not shown), the service does something that takes time, as expressed by the work during instruction.

❿ When the coffee machine becomes empty, it needs to be cleaned, which is modeled by the changing of version (see ⑤).

⑤ In this version, only one activity is ever created which represents the time needed to clean the machine.

[4]Synchronization counters of [8] may help in writing such a rule: inhibit `ask_for_sugar` when $\neg(\sharp t($`ask_for_coffee`$) - \sharp c($`ask_for_coffee`$) = 1)$. See also [10].

Change of version flushes pending requests. It is up to the component to notify others that it has changed of version. It is up to requesting activities to be sure that their request has been taken into account either by asking for an explicit acknowledgment (wait for ...) or by using a service with an appropriate watchdog (**on/each/when** ... **delay** ...).

Ardeco features not shown in the previous example include :

- in the **features operation** clause: defer a_service as long as some_condition: the beginning of an activity is delayed until the condition is falsified. It also applies to watchdog instructions.

- properties (see section 4) are logical expressions hooked on components. They will be examined by the simulation.

- (F-bounded) genericity ([4]): rather than defining a coffee machine component type, a more general beverage machine component type may be described as in:

```
component Beverage_Machine[T <: Beverage]
. interface
. . input ask_for_beverage(integer),
. .         ask_for_sugar, do_beverage(integer), temperature(real)
. . output deliver_beverage, verify_temperature
...
end Beverage_Machine
```

A coffee machine may then be created by instantiating `Beverage_Machine[Coffee]` or a chocolate machine may be created by instantiating `Beverage_Machine[Chocolate]`.

3.3. Constructing a Software Architecture

The architecture we are interested in is a logical view. Components are linked via communication channels. Thus, the considered relationship is what is usually called a connector [1] in the literature.

Only static architectures are considered. We do not deal with dynamic creation or destruction of components. However, the use of versions can most of the time supply dynamic mechanisms.

Instances of components types are connected via typed channels. Because independently developed components may define a required service with one signature and the corresponding provided service with another signature, it is possible to explicit a cast on a channel and connect services with different names.

As an example let us consider a `Vending_Machine` distributing coffee, chocolate and orange juices. We generalize the specification of a `Beverage_Machine` to build a machine which can distribute coffee, chocolate or orange juices.

<u>type</u> Sensor_Temperature = <u>record</u>(min : **integer**, max : **integer**);
<u>type</u> Beverage = <u>record</u>(t : Sensor_Temperature, ...);

Each beverage is (partly) defined by the temperature interval outside which a machine is not allowed to distribute it.

The `Vending_Machine` will use three instances of `Beverage_Machine`, one for each delivered beverage. Using the previous types, we can now generalize the previous `Coffee_Machine` to `Beverage_Machine` mainly by renaming input and output points as follows :

<u>component</u> Beverage_Machine
. **<u>interface</u>**
. . **<u>input</u>**
. . . ask_for_beverage(**integer**), ask_for_sugar, do_beverage(**integer**),
. . . temperature(**real**)
. . **<u>output</u>**
. . . deliver_beverage, verify_temperature
...
. . **<u>version</u>** Distributing_Beverage(n,t : **integer**; b : Beverage)
. . . **<u>data</u>**
. . . . min : **integer** = b.t.min; max : **integer** = b.t.max;
...

Figure 4 presents a graphical view of a subset of this architecture. As before we put circled numbers for references in this text. They are not part of the architecture.

Filters are introduced to model general communication pattern (not only peer to peer). A filter receives a message and in no time redirects it to targets. It can be viewed as a specialized component.

① This filter breaks the `do_beverages` request and redirects to each internal component a modified request with the only parameters each component really needs.

② This filter broadcasts the `ask_for_sugar` request to the two components 'a' and 'b' (respectively the "coffee" and "chocolate" `Beverage_Machine`). As expected, this filter does not broadcast (④) the request to component 'c' (the "orange" `Beverage_Machine`).

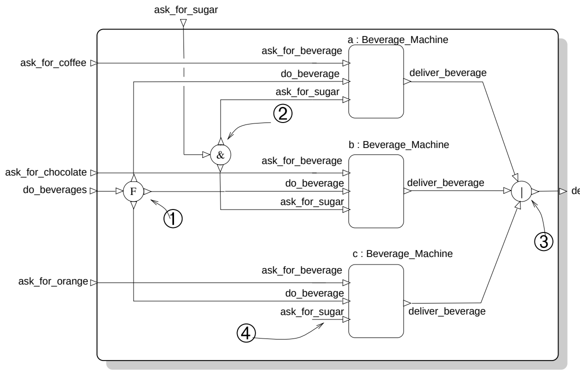

Figure 4. Vending machine architecture (1/2)

③ This filter simply transmits any delivering request from one component to the outside.

Figure 5 presents the graphical part of the architecture concerned with handling temperatures.

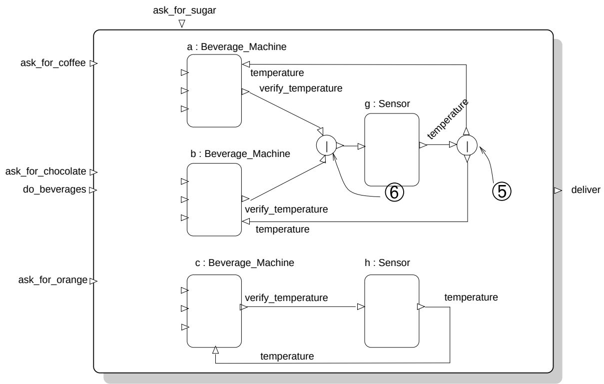

Figure 5. Vending machine architecture (2/2)

In this figure we can see that 'a' and 'b' components share sensor 'g', whereas the 'c' ("orange") component has its own sensor 'h'. Filter (pointed by) ⑥ simply forwards any `verify_temperature` request from 'a' or 'b' to 'g'. Filter ⑤ sends the reply. However how can filter ⑤ determine the correct receiver ('a' or else 'b') ? The answer lies in a new **<u>features</u>** sub-clause named **<u>activity</u>**. This feature links a calling activity to the corresponding answer.

The previous architecture can then be seen as a component itself. **<u>features</u>** rules may (and usually must) be given to drive the sub-components. At any time the architecture is in one version, with data, parameters, and features. An architecture version depends on its constituents' version, which means that version change in one component is notified to the enclosing component. The *Ardeco* description is given in figures 6 and 7.

```
component Vending_Machine
① interface
. . input
. . . ask_for_coffee(integer), ask_for_chocolate(integer),
. . . ask_for_orange(integer), ask_for_sugar,
. . . do_beverages(integer)
. . output
. . . deliver
② architecture
⑤ . components
⑧ . . component Sensor
. . . . interface
. . . . . input
. . . . . . verify_temperature
. . . . . output
. . . . . . temperature(real)
. . . . version
. . . . . Init(...), Working(...), Out_Of_Order(...)
. . . . behavior
. . . . . version Init(...)
. . . . . . triggers
. . . . . . . first : when true
. . . . . . . . ...; become Working(...);
. . . . . . . endwhen
. . . . . version Working(...)
. . . . . . features
⑨ . . . . . . activity
. . . . . . . . receiving_of(temperature)
. . . . . . . .                 ← sender_of(verify_temperature)
. . . . . . triggers
. . . . . . . on verify_temperature
. . . . . . . . ...; temperature(...); ...
⑩ . . . . . properties
. . . . . . . □(acceptance of verify_temperature
. . . . . . .                → ◇_{≤5} emission of temperature)
. . . . . version Out_Of_Order(...)
. . . . . . ...
. . . end Sensor
. . . component F
. . . . ...
. . . end F
⑥ . create
. . . t_1 : Sensor_Temperature ← (50,70);
. . . t_2 : Sensor_Temperature ← (0,10);
. . . coffee : Beverage ← (t_1,...); chocolate : ← (t_1,...);
. . . orange : ← (t_2,...);
. . . a: Beverage_Machine ← {Doing_Beverage(coffee,...)},
. . . b: Beverage_Machine ← {Doing_Beverage(chocolate,...)},
. . . c: Beverage_Machine ← {Doing_Beverage(orange,...)},
. . . d : &_{1,2} ← {...}; // ② on figure 4
. . . e_1 : |_{3,1} ← {...}; // ③ on figure 4
. . . e_2 : |_{1,2} ← {...}; // ⑤ on figure 5
. . . e_3 : |_{2,1} ← {...}; // ⑥ on figure 5
. . . f: F{...};
. . . g : Sensor ← {Init(...)};
. . . h : Sensor ← {Init(...)};
⑦ . connect
. . . self.ask_for_coffee to a.ask_for_beverage;
. . . self.ask_for_chocolate to b.ask_for_beverage; ...
. . . self.do_beverage to f.in; f.out1 to a.do_beverage; ...
. . . . ...
```

(To be continued on figure 7)

Figure 6. Outlines of a vending machine in *Ardeco* (part 1/2)

① The interface of the architecture `Vending_Machine` (which is a also component) defines five input points (one for each beverage, one for sugar and one for initializing internal components) and one output point for delivering the beverage.

② Internal components and their connections are then described :

⑤ Component–types whose instances will be used in the architecture are defined here : for instance a `Sensor` component–type (⑧) is described. We may notice the **activity** clause (⑨). It dynamically "links" the activity that called `verify_temperature` so that the `temperature` answer is received by only this "linked" activity (❽ on figure 3). A property (⑩) to be verified on every instance of this component–type during execution of the architecture is also given (see § 4). Different instances of the same component–type may be given different properties (not shown here).

⑥ Component–types are instantiated ('var : component–type ← initial_version'). Variables are created ('var : type ← value'). Figures 4 and 5 should make it clear.

⑦ This clause introduces the connections between input and output points of the inner components. Again, have a look at the previously mentionned figures.

③ The component/architecture `Vending_Machine` can be in one of three versions, each of which is computed by the versions in which the inner components are.

④ This behavior's component mainly defines condition under which its incoming requests are (or are not) forwarded to its inner components. For instance, a request for `do_beverage` will be discarded when in `Distributing` version. The exclusive a_1, a_2, ..., a_n is a macro for $\forall 1 \leq i, j \leq n$, a_i excludes a_j.

4. Validating a software architecture: its simulation

Simulation is intended to detect discrpencies between a stakeholder's notion of the required system behavior and the system behavior captured by the architecture description in *Ardeco*. The simulation symbolically executes the *Ardeco* description to observe that it captures the intent.

The simulation aims at checking component properties. These properties are part of the component description. They are logical expressions about the state and

(see figure 6)

③ **versions**
. . Init(. . .), Distributing(. . .), Cleaning(. . .)
. . **computed by**
. . . become Distributing when a.Distributing ∧ b.Distributing
. . . ∧ c.Distributing ∧ g.Working ∧ h.Working;
. . . become Cleaning when a.Cleaning ∧ b.Cleaning ∧ c.Cleaning
④ **behavior**
. . **version** Init(. . .)
. . . **features**
. . . . **coordination**
. . . .
. . . . **operation**
. . . . . inhibit ask_for_coffee, ask_for_chocolate,
. . . . . ask_for_orange, ask_for_sugar when true;
. . . .
. . **version** Distributing(. . .)
. . . **features**
. . . . **coordination**
. . . . . exclusive ask_for_coffee, ask_for_chocolate, ask_for_orange
. . . . **operation**
. . . . . inhibit do_beverage when true;
. . .
end Vending_Machine

Figure 7. Outlines of a vending machine in *Ardeco* (part 2/2)

the events of a component in a given version. They are described in a real-time temporal logic [5]. For instance the coffee machine vendor may be interested in $\Diamond(m = 0)$: "will the machine be eventually empty?". A client may be interested in $\Box(\texttt{ask\_for\_coffee}\ \text{received} \rightarrow \Diamond_{\leq 5}\texttt{deliver\_coffee}\ \text{emitted})$: "will I obtain my coffee within 5 seconds?". Because an architecture is also a component, properties can be observed on the architecture itself.

The validation of the architecture may be done in two steps. The first one is simulation with infinite resources without taking into account the physical resources which can ensure that the logical architecture does not content logical inconsistencies. The second step is a mapping on hardware and middleware resources. For this, one must describe for each component on which resource it will be (a component containing activities cannot be distributed on more than one resource), which type of scheduling the resource uses, and the type of communication between components (middleware or network characteristics).

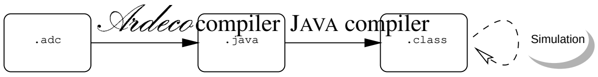

Figure 8. *Ardeco* analysis chain tool.

The simulator is written in JAVA and includes two parts. The first one is a compiler producing a set of JAVA classes representing components from their *Ardeco* description.

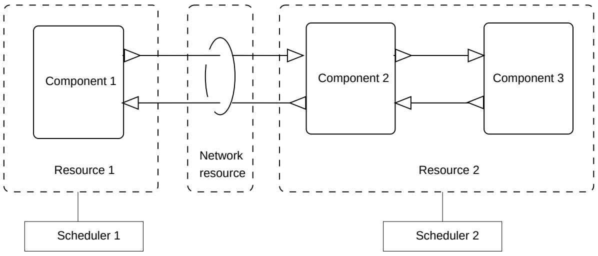

Figure 9. Mapping on resources (dashed boxes).

The second part is the kernel of the simulator which drives the set of produced classes. During the simulation, all components use the same logical clock. The time is incremented by one unit by the kernel when all components finish what they have to do during this tick. As previously seen, there are only two sorts of instructions with time consumption: 'wait for ⟨ event expression ⟩' spends logical time; and 'work during ⟨ δt ⟩' spends CPU time. Logical and CPU times are managed differently by the simulator. An activity can not execute 'work during ⟨ δt ⟩' if it has not access to the processor. Hence, the logical time passed is greater than δt and it depends on the scheduling and the processor's load. Note that those two notions of time are equivalent in infinite resources.

In the course of the simulation, the simulator may point out exception occurrences if a property is not respected, and/or can create a trace of events for each component. This trace can be analyzed to find some time properties (for example: the maximum of time between the asking for coffee and the delivery of the coffee) and to size the amount of data.

5. Conclusion and further work

The component notion is the basis of our approach. We deal with coarse-grained components rather than fine-grained objects, with the only restriction that a component cannot be distributed on several physical resources. Examples of application domains are air traffic management or simulation and training technologies. Such systems are also composed of physical equipments, but we assume that they always are software intensive systems.

Components are our chosen direction to handle reusability and maintainability. Thus, they are mainly built on object-oriented concepts such as encapsulation. As opposed to object-oriented models, component interfaces are much more precise, indicating required resources and offered services, as well as available versions and visible properties.

Internal descriptions are conceived to take into account every possible time consumptions, in order to check if upper time-deadlines are kept. Thus, internal events structure the treatment of services, by distinguishing between reception, acceptance or treatment of external requests with the opportunity to introduce many time consumption points. Coordination characteristics and component properties are given in a declarative style, independently of activity codes. This approach follows [10, 11].

Many works are ongoing on the use of architectural descriptions and support in the context of product line applications, where many variants of the same products must be conceived. These architectures must forecast many genericity points and specify variation points as well as main commonalities of a product family. The way to model commonalities and variation points is under study.

Acknowledgments: We are grateful to the members of the CoVADIs team. We appreciate their insights on the *Ardeco* language definition. This research is partly funded by the French Ministry of Defense under grant 96.34.448.00.470.75.65 and the French Ministry of Research under grant 96C0076.

References

[1] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[3] G. Berry and G. Gonthier. The ESTEREL Synchronous Programming Language : Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, Sept. 1989.

[5] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative Temporal Reasoning. In *Lecture Notes in Computer Science. Springer-Verlag*. 1990.

[6] R. Milner. *Communication and Concurrency*. HOARE, C.A.R., Prentice Hall International, 1989. Series in Computer Science.

[7] A. Pnueli and Z. Manna. *The Temporal Logic of Reactive and Concurrent Systems—Specification*, volume 1. Springer-Verlag, 1992.

[8] P. Robert and J. Verjus. Toward Autonomous Descriptions of Synchronization Modules. In B. Gilchrist, editor, *Proc. IFIP Congress*, pages 981–986, 1977.

[9] E. Shokri, P. Crane, and K. Kim. An Implementation Model for Time-Triggered Message-Triggered Object Support Mechanism in Corba-Compliant COTS platforms. In *Object-Oriented Real-Time Distributed Computing*, pages 12–21, 1998.

[10] L. Thomas. Inheritance anomaly in true concurrent object oriented languages: A proposal. In *IEEE TENCON'94*, pages 541–545, August 1994.

[11] L. Thomas. An Object Oriented Concurrent Language for Extensibility and Reuse of Synchonization Components. *Computers and Artificial Intelligence*, 15(5):437–457, December 1996.