# Integration of Time Issues into Component-Based Applications

Sébastien Saudrais, Noël Plouzeau, Olivier Barais

# Integration of Time Issues into Component-Based Applications

Sébastien Saudrais, Noël Plouzeau, and Olivier Barais

IRISA France, Triskell Project*
{ssaudrai, barais, plouzeau}@irisa.fr

**Abstract.** In this paper we describe a technique for specifying time related properties on traditional software components. We apply the separation of concerns paradigm to allow independent specification of timing and to integrate time-checking specialized tool support into conventional software design processes. We aim at helping the designer to specify time contracts and at simplifying the introduction of time properties in the component behaviour description. We propose to handle timing issues in a separate and specific design activity, in order to provide means of formal computation of time properties for component assemblies without modifying in depth existing design processes.

## 1 Scope and Objectives

Component based design is now at the heart of many modern applications. A rather important category of these applications must manage time, for instance because they interact with users in a time controlled manner (e.g. media players, group cooperation environments, etc) or because they are highly distributed (e.g. applications based on a bunch of Web services from diverse origins). Yet mainstream design techniques often emphasize type centric interactions between components: the component models they use offer powerful notations and tools for defining, refining and checking data types. Time properties are not explicitely taken into account by these models. At the source code level, programming languages and their associated frameworks also include some time characteristics [8]. Again, time propeties such as the maximum duration of an operation execution are treated as second class concepts: there are no time type systems. To overcome this deficiency, timeliness and other quality of service properties are sometimes specified using meta-attributes of programming languages (e.g. C# or Java). From a static validation point of view, these attributes are often treated like structured comments. These comments may be used to generate runtime monitors but their semantics is usually too weak to allow reasoning about time properties.

At the design level, several research results have shown the usefulness of specific languages to describe component based software architectures. Thanks to the precise semantics of such languages, tools suites have been developed to analyze the consistency of a software architecture and to prototype it. For example, SOFA [17] provides a specific language that extends the OMG IDL to describe the architecture of component based software. It also provides a process algebra to specify the external behaviour

* This work was funded by ARTIST2, the Network of Excellence on Embedded Systems Design.

H.W. Schmidt et al. (Eds.): CBSE 2007, LNCS 4608, pp. 173–188, 2007.

of component. However, using SOFA the architect cannot describe the required and provided QoS of components. The AADL standard [26] is one of the first ADL that provides mechanism to specify the QoS into the component interface, also identified as the fourth level of contract [7]. However, AADL is a low abstraction model, strongly connected with the implementation. Besides, AADL is not yet connected with tools that use the QoS information to analyze the consistency of the architecture. In the domain of model driven engineering, modeling languages such as the UML use profiles to add time and performance dimensions [25]. Many profiles exist for designing real time systems: SPT-UML from OMG, MARTE [1]. These profiles define concepts for modeling real-time system but without precise semantics [15]. All these diverse time models are not formal enough to allow reasoning on time properties of software modules. Working with time properties of software components' assemblies is even more difficult, because loosely defined time notions do not compose well and they cannot be used to build quality of service contracts. On a more theoretical point of view, many formal systems exist to describe timed behaviours and reason about them. For instance, timed automata models support well-defined composition operations. Therefore they can help to specify precise component interfaces, which include types, logical conditions, behaviour and time specifications. Furthermore, tool chains provide automated means to check timed automata against time properties, e.g. timed logic formulas.

In this paper, we argue that time properties must be defined in a component interface. We propose a technique to manipulate time as a separate dimension of component-based software design in order to improve the modularity when the architect defines its architecture. It uses formal time conceptual tools based on temporal logic with quantitative timed automata. This time model can be used at design time to check a component's design against a specification and to compute the properties of component assemblies. The time model is also used to generate monitors that test and supervise component implementations.

The rest of this paper is organized as follows: Section 2 presents an overview of our meta-model for components. Section 3 details the time formalism and how to add timed information into components. Finally, Section 4 describes related work. Section 5 concludes and discusses future work.

## 2 Analysis and Design

Our approach extends a component based design process and relies on a set of artifacts. In this section we describe these artifacts together with a global overview of the process.

### 2.1 Artifacts of the Process

The component design process uses or produces the following artifacts:

*A service specification* describes what can be requested from a component, using type definitions and operations to request service execution. Operations can carry constraints such as type, pre and postconditions, behavioural and time related properties. Our interpretation of the notion of service bears some resemblance to the Web service notion: a service is defined as a public capability to perform a rather specialized set of

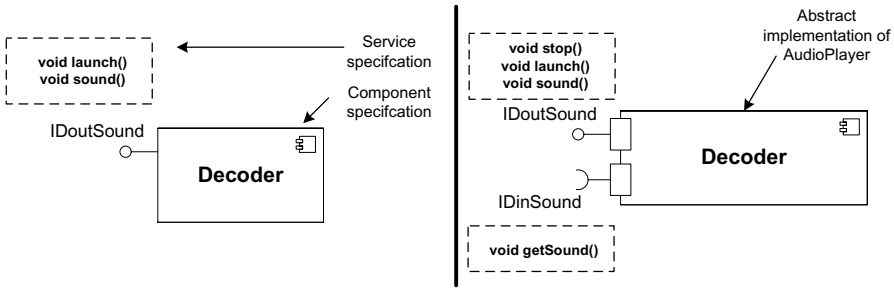


**Fig. 1.** Example of artifacts

tasks (e.g. hotel room booking service). A given service provide means to solve one application domain's precise concern.

*A component specification* groups a set of services supported by the component implementations. A component specification is more than a bundle of services: a component specification gives additional constraints that pertain to the coordination of the services. This resembles the specification of a compound Web service (a choreography) built with an orchestration of other Web services.

*An abstract implementation of components* must adhere to a component specification in order to implement a set of services. This abstract implementation publishes additional information, such as the set of required services that the implementation relies upon in order to perform its tasks, and bounds to quantitative properties of the services that the component implementation provides. These bounds usually depend on quantitative properties of the environment. An abstract implementation hides all platform-specific details: it is a description suited to formal validation of composition, and to computation of the properties of a composition. In other words, the abstract implementation must contain all information needed to check for properties of individual component and component assemblies while hiding all other details not needed by these property checks.

*A concrete component implementation* is a code level entity that is runnable in a component runtime environment. A concrete implementation must provide the services of its associated abstract component implementation, together with the associated properties.

Each level (service specification, component specification and component implementation) conforms to languages or metamodels that define the fundamental constraints and properties of service and component models (for the sake of simplicity we merge the concept of metamodel and language here). Fig 1 illustrates the three first levels in UML2. In our tool chain implementation, we have selected languages and metamodels that (1) support simultaneously constraints ranging from traditional type compatibility up to real-time properties (e.g. timeliness), (2) are semantically sound, (3) are supported by tools for validation, (4) allow for COTS implementation.

In order to define a notation suitable for timed specifications and abstract implementation, we have extended a subset of UML 2.0. We base our subset on existing component based architecture concepts (components, ports, interfaces and connectors) of the UML and extend them with time related features. The resulting notation

(metamodel) is resembling those used by other approaches such as [23]. The design model is organized in two main parts:

1. the service specification describes services that components will implement, including time constraints;
2. the abstract implementation describes a component based architecture and provides a definition of the component behaviour.

Every component metamodel must deal with a clear, sound and complete definition of composition. In the UML 2.0, the composition semantics is not complete enough to support formal definitions of component interactions.

A common notion of component compatibility relies on type compatibility models derived from object-oriented programming: two ports can be connected when the port providing service exposes an interface subtyping the interface exposed on the port requiring service. However, this type model has already shown its limits [24]. The compatibility between two operation prototypes cannot guarantee their correct use. To overcome these limits, we rely on a "rich" black box model that includes timed behaviour descriptions. These specification enrichments are commonly used in Design by Contract software development techniques. They guarantee every component of a system lives up to its expectations. In our approach, according to [7], we identify four levels of contracts. The first level of contracts is based on classical type compatibility. The second level deals with behavioural contracts and it strengthen the level of confidence in a sequential context. A behavioural contract is a set of constraints defined as pre and post conditions on an operation. The third level deals with synchronization contracts. This level provides coordination rules in distributed or concurrency contexts, by explicit specification of the observable behaviours of a component. The fourth level deals with quantitative contracts, quantifies quality of service and the relevant contracts are usually parameterized through a negotiation.

The designer should be able to design each level independently and should be able to ensure at the design stage that the architecture obeys the component's contracts.

In the UML 2.0 world, a service is associated with the definition of a provided interface that specifies the operations that can be invoked. An abstract component specification is a set of services. It declares the interfaces provided by a component. These interfaces are enriched with four levels of contracts. Several approaches have worked on the first three levels. In section 3.2, we lay out a set of mechanisms to define and integrate the fourth level related to the quality of service.

### 2.2 Abstract Implementation of an Architecture

The next step in the software development life cycle is the abstract implementation of the component. By abstract implementation we mean the description of the component implementation where we omit all details that are not necessary to understand how a component interact with its environment along the time dimension axis. This step also defines either a component's internal structure (an assembly of other components) or its behaviour and temporal specification. This section presents the structural concepts for defining the architecture and the formalisms for the behavioural and the temporal properties of components.

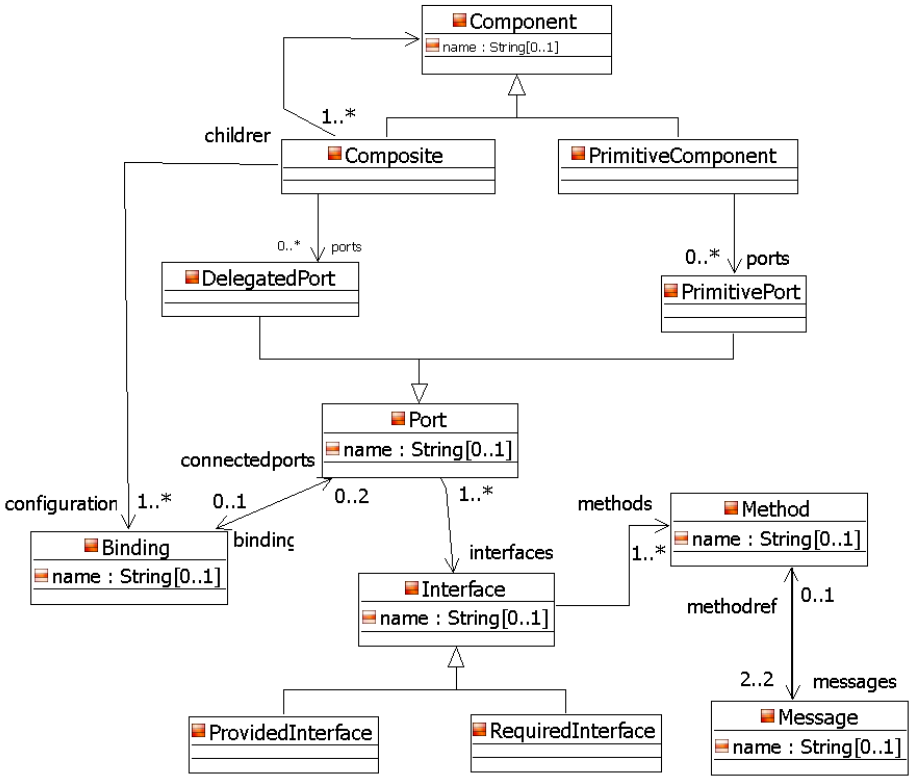


**Fig. 2.** Structural part of the component's Metamodel

**Structural Elements of the Component Model.** The structural part of our component model is largely derived from the UML 2.0 architecture metamodel concept. However, contrary to the UML 2.0, we define an abstract model with fewer concepts to limit the complexity of the language that the architect has to manipulate, and to remove all the semantic variation points existing in UML 2.0.

Consequently, in our component model, a *component* provides *methods* and may require some services from other components. Services can only be accessed through explicitly declared ports. A *port* is a binding point on a component that defines two sets of interfaces: *provided* and *required* ones.

Our component model distinguishes between two kinds of components: *primitives* and *composites*. *Primitives* contain executable code and are basic building blocks in component assemblies. *Composites* are used as a mechanism to deal with a group of components as a whole, while potentially hiding some of the features of the subcomponents. Our component model does not impose any limit on the levels of composition. There are, therefore, two ways to define the architecture of an application: using a *binding* between components ports or using a composite to encapsulate a group of components. A connector associates a component's port with a port located on another component. Two ports can be bound with each other only if the interfaces required by

one port are provided by the other, and vice versa. This constraint on binding is the classical type compatibility (level 1 contracts). The services provided and required by the child components of a composite component are accessible through *delegated ports* which are the only entry points of a composite component. A delegated port of a composite component is connected to exactly one child component port. The structural part of the component model is presented in Fig. 2.

**Behaviour Specification.** With the interface and method definitions, a component declares structural elements about provided and required services. The behaviour specification defines the component's interactions with its environment. This behaviour is declared by a process algebra with the In and Out Automaton model [21] to check the system.

*Process algebra.* To specify a component behaviour, we use a reduced process algebra inspired by FSP [22]. This process algebra is based on an expression describing a set of traces (sequences of events). When applied to components, an event is an abstraction of a method call or response to a call. For example, a call of $m1$ on the interface $i1$ of the port $p1$ is captured as $p1.i1.m1$, a response to the call as $p1.i1.m1\$$. Every event is emitted by a component and accepted by another component. Calling $m1$ via the interface $i1$ of the Port $p1$ is seen as the emission of $!p1.i1.m1$ by the component $C1$ (denoted by an event token of the form $!C1.p1.i1.m1$); at the same time the reception of $p3.i2.m1$ is accepted by $C2$ (denoted as $?C2.p3.i2.m1$ from the perspective of $C2$).

The operators employed in behaviour protocols are: $\rightarrow$ for sequencing, $|$ for alternative choice and $*$ for a finite repetition. This algebra is used to represent the behaviour of primitive components only.

*The I/O automaton model.* Besides a process algebra, we use an I/O automaton formalism to perform checking.

**Definition 1.** *(I/O automaton)*
*An Input/Output automaton is a tuple* $(S, L, T, s_0)$ *where:*

- $S$ *is a finite non empty set of states,*
- $L$ *is a finite non empty set of labels.* $L = I \bigcup O$ *where* $I$ *is a set of inputs and* $O$ *the outputs and* $I \bigcap O = \emptyset$*,*
- $T \subseteq S(L \bigcup \{\tau\})S$ *is the finite set of transitions where* $\tau$ *is an non observable internal action.*
- $s_0$ *is an initial state, an element of S.*

*Composition of I/O automata.* The composition of components in our system is based on the synchronization of an output of a component with the input of a connected component[10].

The composition of I/O Automata is associative and commutative. When the architect composes several components, the composition order is irrelevant.

This process algebra can be seen as a textual representation of a subset of the sequence diagram where the roles, identified in the diagram, are the port of the component.

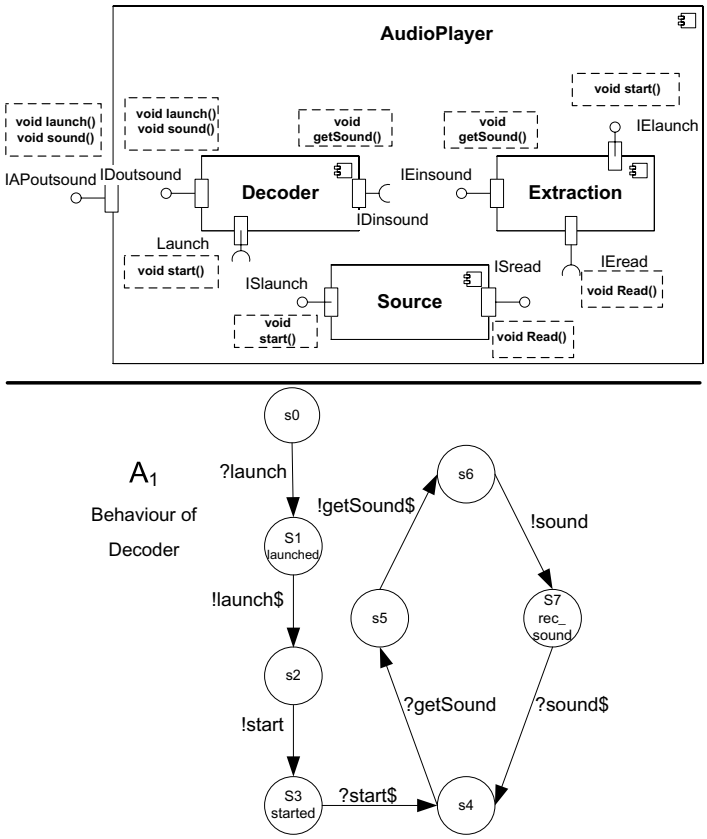


**Fig. 3.** Example of an audio player component

**Example.** Figure 3 illustrates the model with an example of component `AudioPlayer`. The `AudioPlayer` component provides an `IAPoutsound` interface that contains methods `launch` and `sound`. It is composed of 3 components: `Decoder`, `Extraction` and `Source`. The top side shows the structural representation of the component in UML 2.0. The bottom of Figure 3 shows an automaton $A_1$ describing all possible behaviours of the `Decoder`.

## 3 Adding Time Properties into Components

After defining all functional properties in the component, the designer may add some extra-functional properties into it. These extra-functional properties include dense time properties. In order to add time properties to the components we will modify add time information in two different places: on behaviour specifications and on contracts attached to required interfaces. These two places represent what the component provide and require and are used during the composition of components. In order to add time to component behaviour, we use the Timed Automata theory [5]. Furthermore, we define

time patterns to help the designer with the definition of time contracts. Our formalism for such contracts is based on a timed temporal logic (Timed Computation Tree Logic [4]).

### 3.1 Adding Time into Component's Behaviour

While time logic is used to specify contracts, one also needs a means to specify the time properties of abstract implementation of components. Since automata are already used to describe component behaviours, we rely on timed automata (TA) to add precise timing constraints on these behaviours.

**Timed Automata.** A timed automaton is an automaton extended with clocks, which are a set of variables increasing uniformly with time. Formally, a timed automaton is defined as follows:

**Definition 2.** *(Timed Automaton) A timed automaton is a tuple* $A =< S, X, L, T, \iota, P >$ *where:*

- $S$ *is a finite set of locations,*
- $X$ *is a finite set of clocks. To each clock, we assign a valuation* $v \in V$, $v(x) \in R^+$ *for each* $x \in X$.
- $L$ *is a finite set of labels,*
- $T$ *is a finite set of edges. Each edge* $t$ *is a tuple* $< s, l, \psi, , s' >$ *where* $s, s' \in S$, $l \in L$, $\psi \in \Psi_X$ *is the enabling condition.* $\Psi_X$ *is the set or predicates on* $X$ *defined as* $x \sim c$ *or* $x - y \sim c$ *where* $x, y \in X$ *and* $\sim \in \{<, \leq, =\}$ *and* $c \, in N$.
- $\iota$ *is the invariant of* $A$. $\iota \in \Phi_X$ *where* $\Phi_X$ *is the set of functions* $\phi : S \rightarrow \Psi_X$ *mapping each location* $s$ *to a predicate* $\psi$,
- $P$ *associates a set of atomic propositions to each location.*

A state of an automaton is a location and a valuation of clocks that satisfies the invariant of that location. Two different types of state transition exist: discrete transitions and timed transitions.

**Timed Patterns.** In order to ease the addition of time constraints to behaviour, we have defined a set of time patterns based on those partially defined in [14]: response time, delay, execution time, period of service call, duration, etc. We explain hereafter two of these timed patterns: response time and execution time.

*Response time.* The *response time* pattern enables the expression of a response time with a timed automaton. The response time is the delay between a service call and its acknowledgment. For example, to express a response time on the getSound service, one needs to initialize a clock when calling getSound; to receive the acknowledgment, one checks if the clock value is correct with respect to a defined value. This pattern requires three parameters: the service call $service$, the operator $\sim \in \{<, \leq, =, \geq, >\}$ and the value $c$. The $RT$ automaton on Figure 4 represents the generic response time pattern. The pattern consists in three locations, two transitions and one clock. The clock is initialized on the first transition with the service call and checked in the second one with the acknowledge of the service. The second location has the property

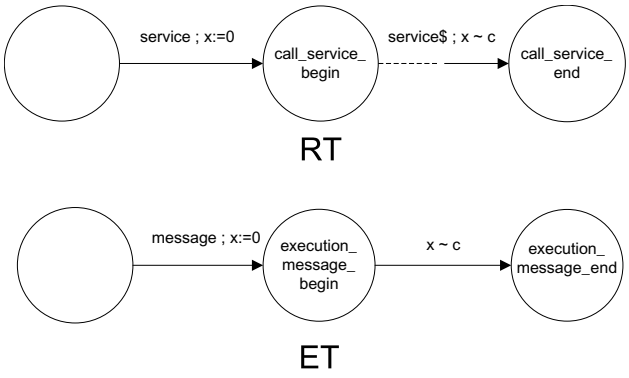


**Fig. 4.** Response time and execution time patterns

$call\_service\_begin$ and the third one $call\_service\_end$. These properties will be used for checking contracts. When the pattern will be added to the component's behaviour, the two transitions need be not consecutive, other transitions can be inserted between them. This is represented on $RT$ by the dotted line between the second location and the second transition.

*Execution time.* The execution time pattern is used to represent an execution with a timed automaton. The execution time is the time used to do a processing. For example, after receiving the response to a service call, the component requires some processing time. The pattern has three parameters: message $message$, operator $\sim$ and value $c$. The automaton $ET$ on Figure 4 represents the generic execution time pattern. The pattern consists in three locations, two transitions and one clock. The clock is initialized on the first transition with the message to be processed and checked in the second transition without any message. The second location has the property $execution\_message\_begin$ and the third one $execution\_message\_end$. These properties will be used to check TCTL formulas. In contrast with the response time pattern, the two transitions must be consecutive because the component is used by the processing and cannot compute something else. This is why the second transition and the third location do not exist in the component's behaviour; they will be created when the pattern will be applied. This way of adding the pattern is not the only one, we can define an execution time pattern where the clock check is added to every outgoing transition of the second location.

**Timed Behaviour.** After defining a set of patterns, we will add them to the component's behaviour. The designer selects the different patterns with their parameters. They will be automatically integrated to the component's behaviour. We will illustrate this design process by adding two timed patterns to the component's behaviour of the example. First, we select the timed pattern response time with the $getSound$ service call, the $<$ operator and the value $4$. For this pattern we add a clock $x1$ to an automaton. This clock is initialized on the $?getSound$ transition from the location $s5$. The $call\_getSound\_begin$ property is added to the targeted location of this transition, in location $s5$. Then we select the transition $!getSound\$$, add the guard $x1 < 4$ and add

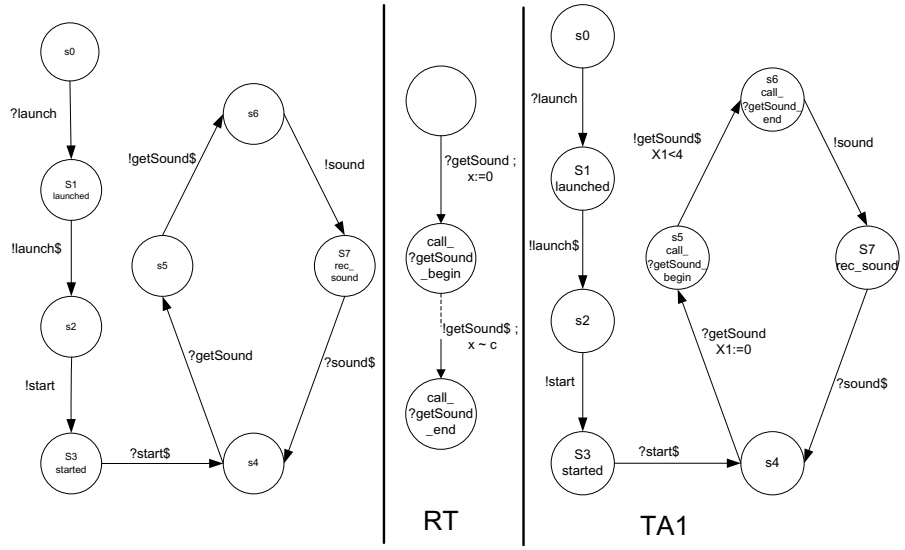


**Fig. 5.** Adding response time patterns to component's behaviour

the property $call\_getSound\_begin$ to the target of the transition. The result is shown on the $TA1$ of figure 5.

Second, we add the pattern execution time with message !$getSound$\$, operator $<$ and value 2. A second clock $x2$ is added to the automaton and it is initialized on the transition !$getSound$\$. We add a new location $s6\_exec$ and a transition between $s6$ and $s6\_exec$ with the guard $x2 < 2$. The outcoming transitions of $s6$ of $TA1$ become the outgoing transitions of $s6\_exec$. The properties $execution\_!getSound\$\_begin$ and $execution\_!getSound\$\_end$ are respectively added to $s6$ and $s6\_exec$. The new automaton of the component is shown on the $TA2$ of figure 6. The new behaviour of the component does not change with respect to the original one : you can obtain $A_1$ from $TA_2$ by removing the clock and the transition without a label.

### 3.2 Adding Time into a Component Contract

Component contracts are part of a component specification; they are bound to ports to describe type, state and behaviour properties that must be enforced by component implementations. In this section we show the addition of time contracts expressed with a timed temporal logic named TCTL [4]. These new contracts will be checked during the composition phase against the timed automata to validate the compatibility between two components.

**TCTL.** TCTL is an extension of CTL [13] with quantitative temporal operators.

In CTL, a formula $\exists\Diamond p$ is satisfied if and only if predicate $p$ can become true along some computation path, without any information about the instant $p$ evaluates to true. The TCTL extension is able to handle quantitative constraints: for example formula

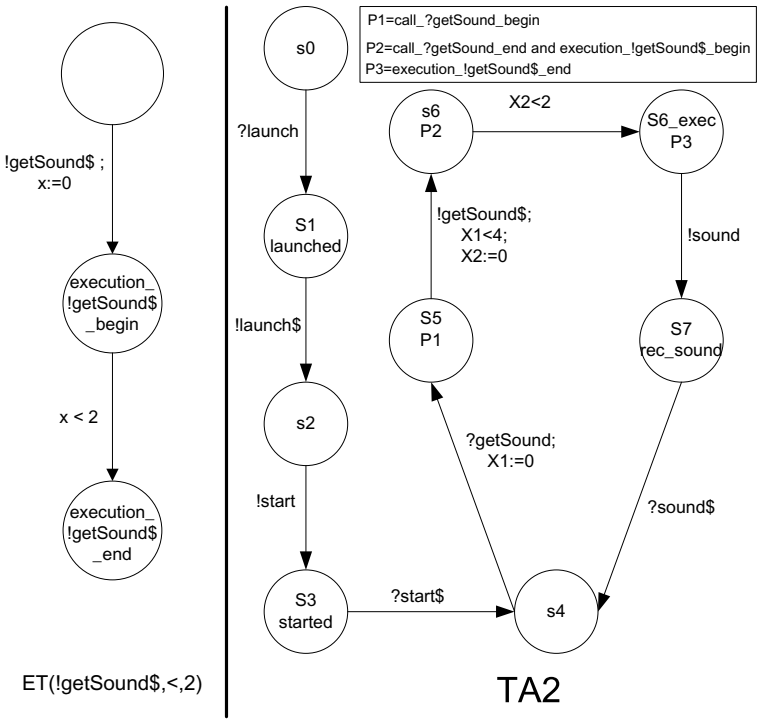


**Fig. 6.** Adding execution time patterns to component's behaviour

$\exists\Diamond_{<5}p$ is true if and only if along some computation path property $p$ becomes true within 5 time units.

Let $P$ be a set of properties and $N$ be the set of natural numbers:

**Definition 3.** *(Syntax) The formulas $\psi$ of TCTL are defined as follows:*
$\psi := p|false|\psi_1 \rightarrow \psi_2|\exists\psi_1 U_{\sim c}\psi_2|\forall\psi_1 U_{\sim c}\psi_2$
*where $p \in P$, $c \in N$, and $\sim \in \{<, \leq, =, \geq, >\}$.*

Abbreviations are defined by:

- $\exists\Diamond_{\sim c}\psi$ for $true\exists$ (possibility),
- $\forall\Diamond_{\sim c}\psi$ for $true\forall U_{\sim c}\psi$ (all locations along all computations),
- $\exists\Box_{\sim c}\psi$ for $\neg\forall\Diamond_{\sim c}\neg\psi$,
- $\forall\Box_{\sim c}\psi$ for $\neg\exists\Diamond_{\sim c}\neg\psi$ (some locations along all computations).

We prohibit the use of more than one clock in a given expression in order to avoid the forward analysis problem[9].

**Timed Contract.** The timed contracts are attached to the required interfaces of a component, like the three other types of contract. To use the definition of these timed contracts, we define a set of patterns based on [19]. These contract patterns are skeletons, which must be completed by the designer. A designer may also write contracts directly

in TCLT. To create a new timed contract, the designer selects the appropriate pattern and provides parameter values. Some examples of patterns are:

- time response of $c$ of service call $foo$ : $call\_foo\_begin \rightarrow \forall\Box(\forall\Diamond_{\sim c} call\_foo\_end)$
- period of $c$ of the property $p$ : $\forall\Box(\forall\Diamond_{\sim c} p)$
- time of $c$ between two property $p1$ and $p2$ : $p1 \rightarrow \forall\Box(\forall\Diamond_{\sim c} p2)$

Other contracts are automatically created when timed patterns are added by the designer. For example, if the response time pattern is chosen with an external service call, (e.g. service $?getSound$), the contract is implicitly included in this pattern. The formula in TCTL is created with the parameters of the timed pattern.

### 3.3 Checking Time Properties When Composing Components

As described in the previous sections, we use timed logic for specification of components and timed automata to describe abstract implementations of these components. Since we use formally defined notations, we are able to use software tools for validation of implementations against specifications. To check the time properties during the composition process, we use the Kronos tool [11], which is able to evaluate TCTL formulas on timed automata. The behaviour of each primitive component is modeled by timed automata and the timed contracts are expressed in the real-time temporal logic TCTL. When a timed automaton does not satisfy a formula, Kronos identifies the locations where the formula does not hold. For instance, if the environment's contracts are :

- receive $sound$ periodically with 7 units of time : $\forall\Box(\forall\Diamond_{<7} rec\_sound)$
- receive $sound$ at least 5 units of time after sending $launch$ :
  $launched \Rightarrow \forall\Box(\forall\Diamond_{<5} rec\_sound)$

Kronos answers $true$ when provided with the timed automaton and the first formula. When we provide Kronos the second formula, the tool answers that the formula does not hold and gives the previous locations of where $rec\_sound$ is true.

## 4 Related Work

Architecture level timing analysis will not come as a replacement for lower-level timing analysis that can be performed once all the detailed design step is achieved. It aims at validating the system early in the development process. To perform such an analysis, an abstract model of the internal behaviour of the components must be known (including estimation of computing times, which can be obtained from a WCET analysis for pre-existing components and by a first evaluation for other ones). In this section, we discuss the different existing models that can be used to describe time properties in a the component behaviour. Next, this section comes back on the issue of the separation of concerns between time properties and functional properties in software modeling.

### UML Profiles, ADLs and Component Models

There are several component based models dedicated to the design of real-time applications. For example, in the UML community several profiles have been proposed to add time information at the modeling level. CQML [3] is a lexical language designed for QoS specification. It can be integrated with UML and can be used at different levels of abstraction. However, CQML is poorly tooled. Consequently, it can not be efficiently used in a software development process. The OMEGA project [2] provides formal methods to check the consistency of UML 2.0 models. The OMEGA approach deals with the specification level only but without link to component-based applications. In the domain of component-based software architecture, the AADL is a new international standard for predictable model-based engineering of real-time and embedded software [26]. Mainly inspired by MetaH [29], its fields of application are automotive, avionics, space and industrial control systems. AADL is a lower-level modeling language than UML or other component models used at the modeling level. Main concepts manipulated by this language are components, ports, threads, communication bus, *etc*. AADL models describe software topologies bound to execution platform topologies. AADL is interesting for two reasons. Far-off the concern of this article, it provides a mechanism of *mode* to model the reconfiguration of statically-known systems. Secondly, more relevant for this paper, it was one of the first ADL to model the quality of service in a component based software architecture. It can model times properties or latency. Nevertheless, AADL is a low-level ADL, directly connected to the implementation. Besides, there is currently no way to help the designer to integrate time properties in an existing component based software architecture.

Several toolboxes for real-time modeling exist. Uppaal [20] is an integrated tool environment for the modeling, the validation and the verification of real-time systems defined as networks of timed automata. Uppaal is able to evaluate CTL formulas on timed automata but can not check TCTL formulas. Consequently, it cannot be used to evaluate QoS contracts expressed on the component interfaces but can be used for the third level.

All the models presented in this section support the description of the architecture and aim at validating the architecture of a system with respect to its timing requirements (e.g. basic and end-to-end deadlines, throughput, etc.). However, two main problems limits their use in a concrete system development process. First, most of these models are not connected to a concrete component platform. Consequently, analysis performed at design time are lost to the implementation. Secondly, due to a lack of separation of concerns in the software development process, time properties and functional properties has to be managed at the same time.

To solve the issue of the gap between the modeling stage and the development stage, BIP [6] provides a framework to model heterogeneous components. The BIP component model is the superposition of three layers: the lower layer describes the *behaviour* of a component as a set of *transitions*; the intermediate layer includes *connectors* describing the *interactions* between transitions of the layer underneath; the upper layer consists of a set of *priority* rules used to describe scheduling policies for interactions. BIP components can be extended with clock variables, but the time model is then a

discrete and simulated one. For instance, BIP can not embed time contracts such as TCTL contracts. Besides, BIP does not provide any mechanism to handle contract violations.

**Separation of Concerns**

Improving the separation of concerns in a component based software architecture comes from a very natural analogy: Just like in an house architecture we have distinct *view/-plan/blueprints* describing distinct concerns of the same house (walls and spaces, electrical wiring, water conducts), it seems reasonable to conceive a software architecture description as the composition of several concerns specifications reflecting several perspectives of the same software system. With this kind of analogy, it seems natural to view time as a separated concern that must be integrated with the rest of the architecture.

In this trend, the Accord methodology proposed in [28] defines a technique based on aspect oriented design to support separation of concerns in real-time component based architectures. The associated component model is tuned to allow the computation of worst case execution time of woven parts rather than general analysis techniques on abstract components.

Klein et al. propose a semantic-based weaving of scenarios [18], where the weaving is based on the dynamic semantics of the models used. This work relies on Message Sequence Charts (MSC) as a language of scenarios, but MSC and I/O automata used to specify the behaviour are similar languages for the weaving operator point of view. Nevertheless, the weaving operator can help the designer to integrate aspect behavioural specification but it does not support timed automata and the integration of QoS in the component specification. Our approach can be seen as a first step to support the weaving of time, although currently we do not provide any pointcut language to specify integration of our time based patterns.

## 5 Conclusion and Perspectives

The separation of concerns make the design easier, improve the testability and the software maintainability. The separation of concerns is often used to modularize in separated units some technical concerns like security, persistence or traceability. This paper addresses time as a concern and proposes mechanisms to help the designer to integrate time QoS information during the specification and the design of a component based software. For example, this approach highlights patterns for the behaviour and the contracts definition.

The work presented in this paper is a part of a global approach that aims to decrease the gap between the specification model and the implementation [27]. It proposes a unified approach to the design and implementation of component based systems. This approach aims at assisting architects in the design and in the implementation of real-time systems by providing a set of tools that check the consistency of the artifacts used to create these systems. This approach is based on an extension of the UML 2.0 standard used to design the services provided by components, to specify components and to give

a first abstract implementation of the systems. Using a Model Driven Engineering style, the approach provides code generation capabilities that clearly separate functional part based on the Fractal Component Model [12] and QoS part based on the Giotto framework [16]. The patterns proposed in this paper are mainly useful at the design stage. They allow to design the software without QoS information and add these information in a second stage.

We are currently working on implementing these patterns as an aspect at the model level. The goal is to design a new primary artifact at the model level to be able to reuse QoS models. Besides, we want to define a expressive pointcut language to simplify the integration of the same QoS model into several component based software architectures. It will also allow the QoS layer to be composed with other aspects of the architecture.

## References


1. MARTE UML profile RFP. voted at OMG. `http://www.omg.org/cgi-bin/doc?realtime/2005-02-06`
2. Webpage of the OMEGA IST project. `http://www-omega.imag.fr/`
3. Aagedal, J.O.: Quality of Service Support in Development of Distributed Systems. PhD thesis, Department for Informatics, University of Oslo (June 2001)
4. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time. Information and Computation 104(1), 2–34 (1993)
5. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
6. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in bip. In: SEFM '06. Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods, Washington, DC, pp. 3–12. IEEE Computer Society Press, Los Alamitos (2006)
7. Beugnard, A., Jézéquel, J.-M., Plouzeau, N., Watkins, D.: Making components contract aware. Computer 32(7), 38–45 (1999)
8. Bollella, G., Gosling, J.: The real-time specification for java. Computer 33(6), 47–54 (2000)
9. Bouyer, P.: Untameable timed automata! In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 620–631. Springer, Heidelberg (2003)
10. Bouyer, P., Petit, A.: Decomposition and composition of timed automata. In: Wiedermann, J., van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 210–219. Springer, Heidelberg (1999)
11. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool for real-time systems. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, Springer, Heidelberg (1998)
12. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B.: An open component model and its support in java. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 7–22. Springer, Heidelberg (2004)
13. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986)
14. Graf, S., Ober, I.: A real-time profile for UML and how to adapt it to SDL. In: Reed, R., Reed, J. (eds.) SDL 2003. LNCS, vol. 2708, Springer, Heidelberg (2003)
15. Graf, S., Ober, I.: How useful is the UML real-time profile SPT without semantics. In: SIVOES 2004, associated with RTAS 2004, Toronto Canada (submitted for publication) (April 2004)

16. Henzinger, T.A., Kirsch, C.M., Horowitz, B.: Giotto: A time-triggered language for embedded programming. Proceedings of the IEEE 91(1), 84–99 (2003)
17. Kalibera, T., Tuma, P.: Distributed component system based on architecture description: The sofa experience. In: Meersman, R., Tari, Z. et al. (eds.) CoopIS 2002, DOA 2002, and ODBASE 2002. LNCS, vol. 2519, pp. 981–994. Springer, Heidelberg (2002)
18. Klein, J., Hélouët, L., Jézéquel, J.M.: Semantic-based weaving of scenarios. In: Proceedings of the 5th international conference on Aspect-oriented software development, pp. 27–38 (2006)
19. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, pp. 372–381. Springer, Heidelberg (2006)
20. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a Nutshell. Int. Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
21. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)
22. Magee, J.: Behavioral analysis of software architectures using ltsa. In: Proceedings of the 21st international conference on Software engineering, pp. 634–637. IEEE Computer Society Press, Los Alamitos (1999)
23. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26, 23 (2000)
24. Meyer, B.: Applying design by contract. Computer 25(10) (October 1992)
25. Object Management Group OMG. UML Profile for Schedulability, Performance, and Time Specification, Version 1.1. (January 2005)
26. As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards n$^o$ AS5506 (November 2004)
27. Saudrais, S., Barais, O., Duchien, L.: Using model-driven engineering to generate qos monitors from a formal specification. edocw 0, 45 (2006)
28. Tesanovic, A.: Aspects and components in real-time system development: Towards reconfigurable and reusable software. Journal of Embedded Computing 1(1), 17–37 (2005)
29. Vestal, S.: Fixed-priority sensitivity analysis for linear compute time models. IEEE Transactions on Software Engineering 20(4) (1994)