Timing and Concurrency Specification in Component-based Real-Time Embedded Systems Development

Hung Ledang
UVHC-LAMIH UMR 8530 CNRS
Valenciennes 59313 Cedex 9, France
hung.ledang@univ-valenciennes.fr

Dang Van Hung
UNU-IIST, P.O. Box 3058
Macao SAR, China
dvh@iist.unu.edu

Abstract

In a development of real-time embedded systems, one needs to consider at very early stages the platform on which the systems are deployed. An explicit specification of system resources enables in fact the specification of non-functional properties. In this paper, as an attempt to define a formal component model for real-time embedded systems, we propose to use resources as the first-class citizen at the contract level of components. We show that this explicit specification of resources enables not only the flexibility of the component contract but also the specification of several non-functional properties of systems. We show a way that the worst case execution time (WCET) and the concurrency of component methods are expressed and validated in terms of resources that are available to the component in question.
Keywords: *component-based real-time embedded systems, concurrency, unifying theories of programming, resource specification.*

1 Introduction

Formal models for the component-based real-time embedded systems have been recently an attractive research topic [17, 7]. By formality, one refers to the formal specification and verification of work-products, not only for their functionality but also for their non-functional properties such as timing and performance in which the concurrency plays an important role.

A challenge for a component-based approach in real-time embedded systems development is the fact that this kind of systems strongly depend on the hardware they are implemented with. The factors like CPU power, network bandwidth and memory volume, really affect the performance of the real-time embedded systems. In order to attack this challenge, we have proposed to include an explicit specification of the resources of component in the component contract [5]. This specification describes the kind of the resources the component needs and their associated value when the component is instantiated.

By introducing the notion of the component resources, the timing constraints of a component method can be expressed as a relation between the worst case execution time (WCET) spent to carry out the method and the resources provided to the component [5]. This adds more flexibility to the existing approaches such as in PECOS model [7] which specifies the memory consumption and WCET of methods as constants.

The real-time requirement of a component based system in general is achieved not only by the individual components but also by their interactions. The implementation of a method may depend on services from other components which may be mutually exclusive in presence of concurrency. The component can provide correct services only if its *schedule invariant* is satisfied. Here the schedule invariant is the assumption of the component on how the schedule handles the concurrent accesses to one of methods. A question which has not been answered in [5] is how the component schedule invariant is specified?

In our opinion, any scheduler must take into account the ability of the concurrent use of methods in a component. So one part of *schedule invariant* should contain the concurrency specification of component methods. In different modelling and programming languages like UML, Java, the concurrency specification of component method is indicated at the signature level, i.e at the component interface. It seems that this specification approach is somehow explicit but not precise because it does not describe :

1. how two different methods can be concurrently executed, and

2. what is the maximum degree of concurrency of a given method (i.e. how many instances of a method can run

concurrently at a time).

In this paper, we propose a solution for the two problems mentioned above with an explicit and precise specification of component method concurrency at the contract level. The concurrency specification is composed of predicates indicating how methods can be executed concurrently with each other. This is in general related to the resource state. Furthermore, the predicates also indicate the maximum degree of concurrency of methods. Here, we also consider how the resource constraints of a component affect the concurrency degree. Two methods of a given component can be concurrently executed if their guards are guaranteed by the current state of the component resources. In other words, resource constraints of component can be used as a way to evaluate the concurrency of methods.

The paper is organized as follows. Section 2 introduces the concept of *timed design* to specify component methods. Section 3 introduces the *timed contract* concept and a way to explicitly specify the concurrency of methods of components. Section 4 presents two models of components, namely *passive component* and *active component* which link specification (contract) and code of components. An architecture and a development methodology which supports our component model are presented in Section 5. A discussion of related works is given in Section 6, and Section 7 concludes the paper.

2 Timed Design

A real-time embedded component when deployed is provided with resources (e.g. CPU, memory, and bandwidth). To specify the resource constraints, we assume a fixed set of integer variables $RES = \{res_1, ..., res_n\}$. The variable res_i indicates a resource type, and its value represents the amount of resources of the type assigned to a component. A method should have a resource specification to specify the resource requirements for its implementation which is a predicate over RES. We introduce a temporal variable ℓ to represent the amount of time spent performing a method. The value of ℓ for a method should satisfy some condition when the execution of the method terminates. This condition is represented as a predicate over ℓ, RES and even the input variables for the method. However, in this paper, for simplicity, we do not consider the input variables in specification of ℓ.

We call the functional and timed specification of a method *timed design* which is defined as follows.

Definition 1 (Timed Design) *Each component method $op(in, out)$ is specified by a so-called timed design $D(op)$ defined as $\langle \alpha, FP, FR \rangle$, where:*

- *α denotes the set of (program) variables used by the method op.*
- *FP denotes the functionality specification and is a predicate of the form*

$$p \vdash_f R \mathrel{\widehat{=}} ok \wedge p \Rightarrow ok' \wedge R$$

where p is the precondition of the method op that the method can rely on when activated, and R is the post condition relating the initial observations (undashed versions of variables) to the final observations (dashed versions of variables). The Boolean variable ok is a special variable denoting the termination of the method according to the unifying theories of programming [9], i.e. ok *has the value true when the method terminates.*

- *FR denotes the non-functionality specification of the method op and is a predicate of the form*

$$q \vdash_n S \mathrel{\widehat{=}} q \Rightarrow S$$

where q is the resource precondition for the method in the given interface which is the assumption on the resources used by the method, and is represented as a predicate on the resource variables, and S is the timed post condition for the method which relates the amount of time ℓ spent for performing the method and the resources used for the method. S is represented as a predicate on the resource variables in RES and ℓ.

We use the index f in $\vdash_f$ to distinguish it with $\vdash_n$, where f stands for functional and n stands for non-functional.

Example 1

We give an example to illustrate the meanings of FR. Let $\alpha \mathrel{\widehat{=}} \{x, y\}, FP \mathrel{\widehat{=}} x{\geq}0 \vdash_f y'^2 = x$ and $FR \mathrel{\widehat{=}} P133 + P266 = 1 \vdash_n ((P133 = 1 \Rightarrow \ell{\leq}10) \wedge (P133 = 0 \Rightarrow \ell{\leq}5))$. Then $\langle \alpha, FP, FR \rangle$ represents a timed design to compute $y = \sqrt{x}$ for a non negative x in which it takes no more than 10 time units when performed by a 133 Mhz processor, and it takes no more than 5 time units when performed by a 266 Mhz processor. □

The predicate q in the non-functional specification FR of op represents not only the *resource precondition* but also the *guard* for the method op. The method op can only be executed if its resource requirements (represented by q) are satisfied by the dynamic environment. Therefore, two instances of op_1 and op_2 can be concurrently executed if their guards, namely q_{op_1} and q_{op_2} are concurrently satisfied by the running environment. In conclusion, the timed design

model is powerful enough to represent the concurrency of component methods.

In order to define the parallel or concurrent composition of timed designs, let's discuss the nature of resources. In general, we can classify resources into two kinds: the *consumable* resources and the *non-consumable* ones. Examples of consumable resources are the energy and device access time; examples of non-consumable resources are memory, processor or physical devices. In the concurrent composition of two methods, their consumable resources are added to each other to form the consumable resources for the composed method; however, to form the non-consumable resources for the composed method, we have to add the non-consumable resources of the two component methods when they are executed properly in parallel, and to take their max when the two methods are executed in the interleaving mode.

We use a resource predicate $truepar$ ranging over $\{true, false\}$ to denote the capacity of running methods in parallel. The value of $truepar$ is $true$ for two sets of resources (each set is for a method) at a resource state iff at the current resource state, there are enough resources for running two methods in parallel. In the case where there is at least one resource which is exclusively-usable non-consumable and needed by both the methods and the available amount of this resource is not enough for running two methods in parallel, $truepar$ is $false$. We assume that $truepar$ is totally defined on $RES \times RES \times RES$. This remark leads to the following definition of the *concurrent composition* of timed designs.

Definition 2 (Concurrent Composition of Timed Designs) *Let $D_1 = \langle \alpha_1, FP_1, FR_1 \rangle$ and $D_2 = \langle \alpha_2, FP_2, FR_2 \rangle$ be timed designs. Assume that $\alpha_1 \cap \alpha_2 = \emptyset$. Then*

$$D_1 || D_2 \mathrel{\widehat{=}} \langle \alpha, FP, FR \rangle,$$

where:

- $\alpha \mathrel{\widehat{=}} \alpha_1 \cup \alpha_2$,
- $FP \mathrel{\widehat{=}} FP_1 \wedge FP_2$,
- *Let* $FR_i = q_i(r) \vdash_n S_i(r, \ell)$, $i = 1, 2$. *Then,* $FR \mathrel{\widehat{=}} q(r) \vdash_n S(r, \ell)$, *where* $q(r) = \exists r_1, r_2 \bullet (q_1(r_1) \wedge q_2(r_2) \wedge r = r_1 \oplus_r r_2)$, *and* $S(r, \ell) = \forall r_1, r_2 \bullet (r = r_1 \oplus_r r_2 \wedge q_1(r_1) \wedge q_2(r_2) \Rightarrow \exists \ell_1, \ell_2 \bullet (\ell = \ell_1 \oplus_l \ell_2 \wedge S_1(r_1, \ell_1) \wedge S_2(r_2, \ell_2)))$.

  In our definition, $q(r)$ holds iff there exists a partition of r into r_1 and r_2 such that $q_1(r_1)$ and $q_2(r_2)$ hold. Similarly, $S(r, \ell)$ holds iff for any a partition of r into r_1 and r_2 such that if both $q_1(r_1)$, $q_2(r_2)$ hold then there is a "partition" of ℓ into ℓ_1 and ℓ_2 for which both $S_1(r_1, \ell_1)$ and $S_2(r_2, \ell_2)$ hold. We define $\oplus_l$ and $\oplus_r$ as follows:

  - $\ell_1 \oplus_l \ell_2 = \max(\ell_1, \ell_2) \triangleleft truepar(r_1, r_2, r) \triangleright (\ell_1 + \ell_2)$.
  - *assume that* $r_i \mathrel{\widehat{=}} (r_{i_1}, r_{i_2}, ..., r_{im})$ *where* r_{ij} *represents the amount of resource* R_j *required by* D_i, *then* $r_1 \oplus_r r_2 = (u_1, ..., u_m)$ *where* $u_j = r_{1_j} + r_{2_j} \triangleleft truepar(r_1, r_2, r) \triangleright (r_{1_j} + r_{2_j} \triangleleft consumable(R_j) \triangleright max(r_{1_j}, r_{2_j}))$.

Here we use the predicate $truepar$ on the set of involved resources to check if two corresponding methods can be executed in parallel or not. We use the predicate $consumable(R)$ to check if the resource R is consumable or not. If $truepar$ is $true$ the two methods are executed in parallel and in this case the amount of each kind of resource used in the composition is the sum of the amount of the corresponding resource used by two methods; the time spent for running two methods in parallel is the maximal time spent for both methods. Otherwise, the two corresponding methods are executed in the interleaving mode. The time spent for running the composition in this case is the sum of the amount of time spent for running the two methods. The amount of each kind of resource is the sum of the corresponding amounts used by each method if the resource is consumable (the predicate $consumable$ takes value $true$). The amount of non-consumable resources is summed-up according to the law of maximum. We use non lazy assumption which means that a task is executable then it should be executed with no delay.

The disjoint parallel composition defined in our previous work [5] is just a special case of Definition 2. The *sequential composition* of timed designs is slightly different from [5] when taking into account consumable resources together with non-consumable resources.

Definition 3 (Sequential Composition of Timed Designs) *Let $D_1 = \langle \alpha, FP_1, FR_1 \rangle$ and $D_2 = \langle \alpha, FP_2, FR_2 \rangle$ be timed designs. Then*

$D_1; D_2 \mathrel{\widehat{=}} \langle \alpha, FP, FR \rangle$,
where:

- *Let* $FP_1 = FP_1(v')$ *and* $FP_2 = FP_2(v)$. *Then* $FP \mathrel{\widehat{=}} \exists m \bullet FP_1(m) \wedge FP_2(m)$.
- $FR \mathrel{\widehat{=}} \exists \ell_1, r_1, \ell_2, r_2 \bullet (FR_1[\ell_1/\ell, r_1/r] \wedge FR_2[\ell_2/\ell, r_2/r] \wedge \ell = \ell_1 + \ell_2) \wedge r = r_1 \oplus_c r_2$, *where* $r_1 \oplus_c r_2 = (u_1, ..., u_m)$, *and* $u_j = (r_{1_j} + r_{2_j} \triangleleft consumable(R_j) \triangleright max(r_{1_j}, r_{2_j}))$.

Here, we use $F[x_1/x]$ to denote the expression resulting from the substitution of x_1 for x in the expression F.

The definition of the refinement relation for timed designs is just a small extension of the one for the designs as presented in UTP and also in Jifeng et al's work [12].

Definition 4 (Timed Design Refinement) *A timed design* $D_1 = \langle \alpha, FP_1, FR_1 \rangle$ *is refined by a design* $D_2 = \langle \alpha, FP_2, FR_2 \rangle$ *(denoted by* $D_1 \sqsubseteq D_2$*) iff*

$$(\forall ok, ok', v, v' \bullet FP_2 \Rightarrow FP_1) \wedge (\forall r, \ell \bullet FR_2 \Rightarrow FR_1)$$

where v, v' *are vectors of the program variables, and* r *denotes a vector of the resource variables,* $r = (res_1, \ldots, res_n)$*. The first part of the conjunction is to say that the functional part of* D_2 *is a refinement of the the functional part of* D_1 *as in Jifeng et al's work [12]. The second part of the conjunction simply says that if the non-functional requirement of* D_2 *is satisfied then the non-functional requirement of* D_1 *is also satisfied. Hence,* D_2 *can implement* D_1*.*

It is obvious that, like for the untimed designs, we have:

Theorem 1 *The relation* $\sqsubseteq$ *is a partial order relation on the set of timed designs, and the concurrent composition and the sequential composition are monotonic according to this relation.*

3 Timed Contract

A component can provide and require several interfaces, however in a formal definition we can regroup them into one interface which declares the services that the component provides as well as the services that the component needs from other components. The definition of a component interface is as follows.

Definition 5 *An interface is a tuple* $I \mathrel{\widehat{=}} \langle Fd, Md_p, Md_r \rangle$ *where:*

- Fd *is a finite set of variables (a feature declaration).*
- Md_p *and* Md_r *are set of methods declaration; each method in* Md_p *or* Md_r *is of the form* $op(in, out)$*, where* in *and* out *are sets of variables.* Md_p *is a set of provide methods (intended to be the ones that the associated component provides to the environment), and* Md_r *is a set of required methods (intended to be the ones that the associated component requires from other components). Naturally, we assume that* Md_p *and* Md_r *are disjoint sets.*

Contract of a real-time embedded component specifies its interfaces. Namely, to each method declared in the interface, is associated a timed design. The set of timed designs for the component interface is represented by the field $MSpec$ in the component contract. We need also a resource declaration for the component, this constitutes the field Rd inside the contract. The resource variables and interface variables should be initialized by an $Init$ clause. In order for the component to provide services as expected, certain conditions should be imposed on consumable resources. So the contract needs a field representing consumable *resource invariant*; we denote it as $InvR$. We suppose that $InvR$ will be guaranteed when the component is initialized and during the lifetime of the component, by an automatic resource self-updated mechanism. We propose to add into the timed contract an explicit concurrency specification of methods. This specification is denoted by the field $ConCall$. $ConCall$ is a set of predicates $Para$ with name of component provide methods as their arguments. The predicates $Para$ indicate the way that the component methods can be concurrently executed.

The timed contract of a component is therefore defined as follows.

Definition 6 (Timed Contract) *A timed contract is a tuple* $\langle I, Rd, MSpec, Init, Inv, InvR, ConCall \rangle$*, where*

- $I = \langle Fd, Md_p, Md_r \rangle$ *is an interface.*
- Rd *- a resource declaration, which is a subset of* RES*.*
- $MSpec$ *is method specification which associates each method* $op(in, out) \in Md_p \cup Md_r$ *with a timed design* $\langle \alpha, FP, FR \rangle$*, where* $(\alpha \setminus (in \cup out)) \subseteq Fd$*.*
- $Init$ *is an initialization, which associates each variable in* Fd *and each local variable with a value of the same type, a variable in* Rd *with an integer.*
- Inv *represents an invariant property of the variables in the feature declaration* Fd *that can be relied on at any time that it is accessible from outside. Hence,* Inv *is satisfied particularly by* $Init$*.*
- $InvR$ *indicates the resource condition under which the services of the component can be executed.*
- $ConCall$ *is a set of predicates; each predicate is of the form* $Para(op_1, ..., op_k)$ *indicating the fact that the methods* $op_1, ..., op_k$ *can be concurrently executed under the current state of resources.* $Para(op_1, ..., op_k)$ *implies* $truepar$ *for the current state of the resources of every pair of methods in the list of its arguments.*

Example 2

Let's consider a component $Decoder$ which provides a method $decode$ for decoding video images. The component $Decoder$ can support two threads decoding images. One thread provides 25 images per second using a rapid algorithm and the another provides 10 images per second using a slow algorithm. With the rapid algorithm, the decoder needs 100KB of memory; with the slow algorithm, the decoder needs 150KB of memory. The time spent for

executing $decode$ method using the rapid algorithm is not more than 40 milli-seconds. The time spent for executing $decode$ using the slow algorithm is is not more than 100 milli-seconds.

The interface of the component $Decoder$ is defined as $I(Decoder) = \langle \emptyset, \{decode\}, \emptyset \rangle$.

The resource declaration of $Decoder$ component is defined as $Rd(Decoder) = \{MEM, CPU\}$. We initialize the resource variables as follows : $Init(Decoder) = \{MEM \mapsto 400, CPU \mapsto 60\}$. The specification of $decode$ method is
$MSpec(decode) = \langle \{source, dest\}, source \neq null \vdash_f dest' = dest \cup nextImage(source),$
$(CPU \geq 20 \wedge MEM \geq 150) \vee (CPU \geq 40 \wedge MEM \geq 100) \vdash_n (((CPU \geq 40 \wedge MEM \geq 100) \Rightarrow \ell \leq 40) \wedge ((CPU \geq 20 \wedge MEM \geq 150) \Rightarrow \ell \leq 100))\rangle$

Since all the resources are non-consumable, $InvR = true$, and since there is no feature specification, $Inv = true$. With the current state of resource, at most two instances of $decode$ method can be executed. Therefore the concurrency specification of $Decoder$ component is $ConCall = \{Para(decode, decode)\}$.

The $Decoder$ component is specified with the following contract:
$\langle \langle \emptyset, \{decode\}, \emptyset \rangle, \{MEM, CPU\},$
$\{MSpec(decode)\},$
$\{MEM = 400, CPU = 60\},$
$true,$
$true,$
$\{Para(decode, decode)\}\rangle$ □

In the definition of timed contract, we have given a "minimal" specification for the required methods. In order to offer correct services, the environment has to ensure that the required services are at least as good as specified according to the contract.

The resource initialization should establish the resource invariant, i.e we need the following proof obligation:

$$(Init|_{Rd})|_{ConsumableRES} \Rightarrow InvR$$

where $(Init|_{Rd})|_{ConsumableRES}$ denotes the restriction of $Init$ on the consumable resource variables in Rd.

The predicates in $ConCall$ have to be validated by the resource invariant and by the current state of non-consumable resources, i.e given $Para(op_1, ..., op_k) \in ConCall$, we need the following proof obligation:

$$InvR \wedge (Init|_{Rd})|_{non-ConsumableRES} \Rightarrow q_{op_1} \otimes ... \otimes q_{op_k}$$

where q_{op_i} is the resource precondition of the method op_i, and the operation $\otimes$ is defined below. Let us denote r_{op_i} the vector of resource variables for the method op_i. We define the operator $\otimes$ as follows:

$$q_{op_1} \otimes ... \otimes q_{op_k}(r) = \exists r_{op_1}, ..., r_{op_k} \bullet (r = r_{op_1} \oplus_r ... \oplus_r r_{op_k} \wedge q_{op_1}(r_{op_1}) \wedge ... \wedge q_{op_k}(r_{op_k}))$$

In case where $InvR$ is $true$, we have only one proof obligation saying that the initialization should validate the resource precondition of methods in a $Para$ predicate.

We give below some more explanations about the concurrency specification of methods.

Methods of a component can be concurrently executed if the resources allocated to the component allow. The introduction of the set $ConCall$ at the component contract level provides an explicit and precise way for the component designers to specify the concurrency of component methods. From QoS viewpoint, by inspecting the concurrency predicates $Para$ in $ConCall$ of a component, one can expect to be deadlock-free and conflict-free on the resource access of the component. The predicates $Para$ are indeed taken into account for scheduling. Any schedule plan has to preserve these constraints. The conjunction of predicates in $ConCall$ takes part of a so-called *schedule invariant* of components. The other part of schedule invariant is derived from the implementation of threads.

As it can be seen later, components may be assembled resulting in a combined component whose contract has $ConCall$ calculable from $ConCall$ of the contract of operand components. In order to facilitate the computation of combined $ConCall$ we assume that the predicates $Para(op)$ for all $op \in Md_p$ are implicitly included in $ConCall$. A specification for $ConCall$ of the value $\emptyset$ (empty set) means that it contains only $Para(op)$ and no-concurrency of methods is possible in the corresponding component.

Notice also that the $Para$ predicates are not transitive, i.e: $Para(op_1, op_2) \wedge Para(op_1, op_3)$ does not imply $Para(op_2, op_3)$. In addition, $ConCall$ should be preserved by the refinement of timed contracts as shown in the following definition.

Definition 7 (Timed Contract Refinement)
Timed contract

$$Ctr_1 = \langle \langle Fd_1, Md_{p_1}, Md_{r_1} \rangle, Rd_1, MSpec_1, Init_1, Inv_1, InvR_1, ConCall_1 \rangle$$

is refined by timed contract

$$Ctr_2 = \langle \langle Fd_2, Md_{p_2}, Md_{r_2} \rangle, Rd_2, MSpec_2, Init_2, Inv_2, InvR_2, ConCall_2 \rangle,$$

(denoted $Ctr_1 \sqsubseteq Ctr_2$) iff:

- $Fd_1 \subseteq Fd_2$, $Rd_1 \subseteq Rd_2$, *and* $Init_2|_{Fd_1} = Init_1|_{Fd_1}$, $Init_2|_{Rd_1} \leq Init_1|_{Rd_1}$ *(where for functions f, f_1, f_2 and a set A, $f|_A$ denotes the restriction*

of f on A, and $f_1 \leq f_2$ denotes that f_1 and f_2 have the same domain and $f_1(x) \leq f_2(x)$ for all x in their domain),

- $Md_{p_1} \subseteq Md_{p_2}$, $Md_{r_2} \subseteq Md_{r_1}$
- *For all methods* op *declared in* Md_{p_1} $MSpec_1(op) \sqsubseteq MSpec_2(op)$
- *For all methods* op *declared in* Md_{r_2} $MSpec_2(op) \sqsubseteq MSpec_1(op)$
- $Inv_2 \Rightarrow Inv_1$.
- *The concurrency constraints declared in* Ctr_1 *are preserved in* Ctr_2*:* $ConCall_2|_{Md_{p_1}} \supseteq ConCall_1$.
- *The resource invariant is weakened under the refinement, i.e:* $InvR_1 \Rightarrow InvR_2|_{Rd_1}$.

Roughly speaking, in the timed contract Ctr_2, more services with better quality and functionality are provided, and less with lower quality and functionality are needed from the environment.

There is a difficulty in defining $ConCall$ of the composition of timed contracts. We cannot compute the set $ConCall$ for the compound one without knowing the implementation of provided methods declared in the operand timed contracts. Hence the composition of timed contracts will be defined at the component level (see Section 4). Here we give a definition for the condition for combining components.

Definition 8 (Compatibility of Timed Contracts) *Let* $Ctr_i = \langle\langle Fd_i, Md_{p_i}, Md_{r_i}\rangle, Rd_i, MSpec_i, Init_i, InvR_i, ConCall_i\rangle$, $i = 1, 2$ *be timed contracts. The contracts* Ctr_1 *and* Ctr_2 *are compatible iff:*

- *Their sets of features and provided methods are compatible, i.e.* $f \in Fd_1 \cap Fd_2$ *implies* $Init_1(f) = Init_2(f)$ *and* $op \in Md_{p_1} \cap Md_{p_2}$ *implies* $MSpec_1(op) \Leftrightarrow MSpec_2(op)$.
- *Their sets of provided methods and required methods are compatible (for connecting): For all* $op \in Md_{r_1} \cap Md_{p_2}$*, it holds that* $MSpec_1(op) \sqsubseteq MSpec_3(op)$*, and for all* $op \in Md_{r_2} \cap Md_{p_1}$*, it holds that* $MSpec_2(op) \sqsubseteq MSpec_1(op)$.

The compatibility is not preserved by refinement without a certain condition. We have:

Theorem 2 *Let* Ctr_1*,* Ctr_2 *and* Ctr_3 *be timed contracts such that* Ctr_1 *and* Ctr_2 *are compatible, and contract* Ctr_2 *is refined by contract* Ctr_3*. Let for all* $op \in Md_{r_1} \cap (Md_{p_3} \setminus Md_{p_2})$ *it holds that* $MSpec_2(op) \sqsubseteq MSpec_1(op)$*, and for all* $f \in (Fd_3 \setminus Fd_2) \cap Fd_1$ *it holds that* $Init_3(f) = Init_1(f)$*. Then* Ctr_1 *and* Ctr_3 *are compatible*

Proof. By direct check for the conditions in Definition 8. □

4 Timed Component

Our component model is used to link the specification (contract) to code of components. We distinguish between *passive components* which provide services and *active components* which are responsible to carry out threads with the occurrence of triggering events.

Definition 9 *A real-time passive component is a tuple* $Comp = \langle Ctr, MCode\rangle$*, where* $Comp$ *is identified with the name of the component, consisting of*

- *a timed contract* $Ctr = \langle\langle Fd, Md_p, Md_r\rangle, Rd, MSpec, Init, Inv, InvR, ConCall\rangle$*. Contract* Ctr *is said to be implemented by* $Comp$*.*
- *a mapping* $MCode$ *assigns to each method* op *in* Md_p *a design build from basic operators (as well understood or defined in by Jifeng et al. [13] with a suitable time consumption assumption as time and resource specification) and the calls to methods* m *in* Md_r*.* $MCode$ *satisfies that* $MSpec(op) \sqsubseteq MCode(op)$ *for all* $op \in Md_p$*.*

We give some explanations here about this model. When implementing its contract, a component can use the services provided by the environment via Md_r. If there is no non-functionality requirement for a method, then a protocol which specifies the temporal order between the calls and is consistent with the environment, is enough for implementing the functionality requirement of the method. However, when taking WCET of a method into account, we need to know how long a call has to wait before being serviced. In this model, we leave this for the environment to handle, and assume that all methods in Md_r have the specification guaranteed by its environment that should include also the time needed to handle the calls. In turn, if some process wants to use a service from a component, it has to go through a "Scheduler" that will provide the service with some modification of the timing specification when taking into account the behavior of the scheduler. So, this model offers the black box view for components.

Now we have to model how components can be connected. If a component C_1 has a provided service $op1$, and a component C_2 has a required service op_2 for which $op_1 = op_2$. Assume that $MSpec_2(op_2) \sqsubseteq MSpec_1(op_1)$ we can combine C_1 and C_2. The compound component will not require op_2 any more, however, the use of the provided service op_1 by the environment now may be more restricted than before: those methods op whose implementation uses op_2 cannot be used in parallel with op_1. So, the set $ConCall$ for the compound component in general should include all the elements of $ConCall_1$ for C_1 and $ConCall_2$ for C_2, and those $Para(op_{1_1}, \ldots, op_{1_k}, op_{2_1}, \ldots, op_{2_j})$

where $Para(op_{1_1}, \ldots, op_{1_k}) \in ConCall_1$ and $Para(op_{2_1}, \ldots, op_{2_j}) \in ConCall_2$ such that $MCode_1(op_{2_i})$ contains no calls to op_{1_h} and $MCode_2(op_{1_h})$ contains no calls to op_{2_i} for all $h \leq k$ and $i \leq j$.

Recall that from the intended meaning of $ConCall$, in this paper we use the convention that for any timed contract, for any op in the contract, $Para(op) \in ConCall$.

To avoid the complex behavior for the compound component, we should rule out the case that methods are implemented using themselves, i.e. to avoid circular calls. So, we define:

Definition 10 (Composability of Components) *Two components C_1 and C_2 are said to be composable iff their contracts are compatible, and the unfolding of the calls to the methods in Md_{p_j} in $MCode_i(op)$, $i, j = 1, 2$, $i \neq j$ is not circular.*

It is trivial to formalize what is meant by "the unfolding of the calls to the methods in Md_{p_j} in $MCode_i(op)$, $i, j = i, 2$ is not circular", and we skip this.

Definition 11 (Combination of Components) *Let $C_i = \langle Ctr_i, MCode_i \rangle$, $i = 1, 2$ be passive components which are composable. The combination $C_1 \cup C_2$ is defined as the component $C = \langle Ctr, MCode \rangle$, where*

- *Ctr is defined as*

$$\begin{array}{rcl} Ctr & = & \langle \langle Fd_1 \cup Fd_2, (Md_{p_1} \cup Md_{p_2}), \\ & & \quad ((Md_{r_1} \cup Md_{r_2}) \setminus (Md_{p_1} \cup Md_{p_2})) \rangle, \\ & & Rd_1 \cup Rd_2, \\ & & MSpec_1 \cup MSpec_2, \\ & & Init_1 \uplus Init_2, \\ & & Inv_1 \wedge Inv_2, \\ & & InvR_1 \otimes InvR_2 \\ & & ConCall \rangle, \end{array}$$

  where $ConCall$ is defined as above, and $(Init_1 \uplus Init_2)(x)$ is defined to be

$$\begin{cases} Init_1(x) + Init_2(x) & \text{if } x \in RES \\ Init_1(x) & \text{if } x \in Fd_1 \cap Fd_2 \\ Init_1(x) & \text{if } x \in Fd_1 \setminus Fd_2 \\ Init_2(x) & \text{if } x \in Fd_2 \setminus Fd_1 \end{cases}$$

- *$MCode(op) = MCode_1(op)$ if $op \in Md_{p_1}$, and $MCode(op) = MCode_2(op)$ if $op \in Md_{p_2} \setminus Md_{p_1}$.*

In Definition 11, for a method that appears in the two components (they should have the same specification as from the compatibility condition), we always use the code given by the first component just for the purpose of determinism because the code given in any of the two components can be used. In case one wants to make use of both codes assigned to the methods with the same (equivalent) specification and same name from two components to be combined for increasing the concurrency (specified in $ConCall$), the only thing one has to do is to rename the methods to make their names different.

Definition 12 (Refinement of Components) *Let $Comp_i = \langle Ctr_i, MCode_i \rangle$, $i = 1, 2$ be passive components. $Comp_1$ is said to be refined by $Comp_2$ (denoted by $Comp_1 \sqsubseteq Comp_2$) iff $Ctr_1 \sqsubseteq Ctr_2$.*

At this point, we can have a series of "obvious" properties for our model that are useful for component-based software development. Namely, the combination of two components, if it becomes close, will be a refinement of each of them, and the combination of components is monotonic for the operands with respect to the refinement relation if the compatibility is not violated.

Theorem 3 *The refinement relation is transitive.*

Theorem 4 *Let C_1, C_2, C_1' and C_2' be components such that C_1 and C_2 are composable, C_1' and C_2' are composable. Then $C_1 \sqsubseteq C_1'$ and $C_2 \sqsubseteq C_2'$ imply $(C_1 \cup C_2) \sqsubseteq (C_1' \cup C_2')$.*

Theorem 5 *Let C_1 and C_2 be composable timed components. Then, $C_1 \sqsubseteq C_1 \cup C_2$ iff $Md_{r_2} \subseteq Md_{p_1} \cap Md_{r_1}$.*

Theorem 6 *Let C_1, C_2 and C_3 be timed components that are pairwise composable. Then, $C_1 \cup C_2$ and C_3 are also composable. Similarly, C_1 and $C_2 \cup C_3$ are also composable, and $(C_1 \cup C_2) \cup C_3$ and $(C_1 \cup (C_2 \cup C_3)$ are refinement of each other (i.e. they are equivalent)*

Definition 13 (Active Component) *Active components are defined in the same way as passive components, except that the active components should have concurrent thread declarations and event declarations. Active components are either driven by events from the environment or by their internal clocks. A thread is defined as*

$$\textbf{\textit{always}}\ D\ \textbf{\textit{follows}}\ e$$

*where e is an event which is a Boolean expression, and D is a method (timed design). The meaning of the notation "D **follows** e" is $e \Rightarrow ok \wedge D$. Roughly speaking, thread T is listening for the occurrences of event e; whenever event e occurs, method D should be executed. The formal meaning of the operator **always** is given using Extended Duration Calculus [2], and is not in the scope of this paper. We refer the reader elsewhere [5, 10] for the details of the formal semantics of threads and examples illustrating the concepts of active and passive components.*

Figure 1. An Architecture for Component Systems

5 An Architecture and a Development Methodology for Real-time Component Systems

Now we propose a component-based architecture for systems in our model. With autonomous components, via their interface, we can put them together by connectors to form bigger and bigger components with more and more services until it becomes closed (i.e. its required interface is empty), and that the services it provides are enough for us to satisfy our requirements specified as processes (use-cases). So, our system built in this way has two parts: the passive part that is a close component composed from a set of components, and the active part which are a set of reactive processes which are triggered by external events and using services from the passive part to satisfy the requests from the external actors of the system.

This system architecture is depicted in Figure 1.

From this architecture, we propose a development methodology for component-based systems as follows. Suppose that we want to develop a real-time embedded system. We do it with the following steps:

- Develop an active component that include all the threads from the requirements of the system. This active component will have a set of required methods that come from the domain analysis and our development knowledge.

- Try to develop new passive components and search the suitable available passive components that can provide services to the active component. The development of a component starts with its contract having an empty set of required methods, and will be repeatedly updated when a service is implemented.

- Put the passive components together in an incremental way using the composition operation on components until we can get a "big enough" passive component that does not have required methods and that have all provide methods that are required by the active component.

- From the predicate $Concall$ of the resulting "big enough" passive component, and from the design of the threads (processes) in the active component, we can analyze the schedulability for the desired system.

6 Related Work

The component model PECOS for field devices [7] shares the explicit specification of resources with our model, however our model is more general because we allow different kinds of resources while PECOS is interested only in the memory consumption. Furthermore, there is no specification on the relation between WCET and resources in PECOS as it is in our model. Each component in PECOS represents indeed a task or a thread while in our model, each component may allow a number of threads. Furthermore, our model is more denotational, especially when the concurrency specification is concerned.

The Metropolis meta-model (MMM) [1] provides building blocks for specifying and designing heterogeneous embedded systems; MMM deals separately with various aspects of system design: computation of a component, communication and coordination between components, performance goals and constraints, and refinement that model design steps. The action automata are used to define execution semantics. Compared with our high level timed designs based on UTP, such a formalism is really at low level. Metropolis uses the linear temporal language (LTL) to specify the coordination OCL constraints on events. A so-called logic of constraint (LOC) is proposed in order to specify the annotated behaviors of events (mainly time constraints). We can notice that such a LOC is quite similar to our EDC language. In the future work, we are planning to add to our model the concept of protocol [11]. Such kind of protocol together with the concurrency constraints proposed in this paper will be a comparable solution with respect to the coordination constraints and concurrency mechanisms in Metropolis.

Tesanovic [17] defined an aspect component model for real-time systems has been defined. The idea is to weave into a core component model different aspects representing QoS properties such as concurrency, security, portability. The core component model in this approach is very similar to our model. However, the significant differences of this model with regard to our model is the capacity of the dynamic reconfiguration of system. However, the composabilities and a formal semantics for the proposed component model remain an open question.

Our approach shares the idea of machine independence proposed by Hayes [8]. However, in Hayes approach, the refinement is limited only for the transition from real-time specification to real-time program with deadlines. A timing analysis is necessary before such a program can be used on a particular target in order to ensure that all deadlines will be reached on time. We note that the timing analysis was done by another mechanism than refinement techniques. In contrast, in our approach, the refinement techniques are used for the transition from specification to final implementation.

Roubtsova et al [18] propose a temporal logic to specify real-time properties in specification classes. Extended class diagrams and extended state-chart diagrams are used together with classical UML diagrams. They also suggest to use XTG to describe the behaviour of real-time systems and propose a technique to convert real-time UML with clock variables into XTG. Shendall et al. [14] propose to specify timing properties as guards for transition, so state-charts can specify real-time behavior. They propose the stereotype "SIP view" to specify the temporal order of the interaction for different customers to simplify the interactions (multiple views). This approach is similar to our specification of concurrent threads except that SIP views do not carry timing information. Dino et al [6] propose a temporal logic is introduced for specifying dynamic and static properties of object systems. A map to convert a large fragment of OCL to the logic is also proposed.

Sifakis et al [15] propose a method to build timed models of real-time systems by adding time constraints to their application software. The applied constraints take into account execution times of atomic statements, the behavior of the system's external environment and scheduling policy. Their model can be analyzed by using time analysis techniques to check relevant real-time properties. In comparison with their work, our approach is similar, but we work at the component level as well as the system level. Also, in our work, in order to increase the re-usability of a component, we specify time as a relation between resource and time constraints.

7 Conclusion

We have presented an approach to formally model the component-based real-time embedded systems. The model is an extension of the one defined in our previous work [5] to explicitly address the schedule of the concurrent use of its services as well as timing and resource constraints. The main purpose of our model is to support the specification and refinement of components, and possibly the verification of the thread deadline constraints with respect to real-time requirements imposed on the occurrences of triggering events.

Our model also supports the separation between the functionality specification from the non-functionality specification of components, which can simplify the verification of the functionality requirements, and in many cases can simplify the verification of non-functional requirements as well, particularly when the real-time requirements are in the form of deadline constraints.

With UML, one can derive a component based design and implementation. But since UML is just semi-formal, it does not support the formal verification of the system. Furthermore, even real-time UML does not support the timed design for components. Our technique is used as a nice complement to UML to support the timed design and the formal verification of the safety critical systems. With the separation of non-functionality and functionality during the system development, we first use UML to design an untimed system that satisfies the functionality requirements. Then, resource and time constraints are added to the untimed design of methods based on the timed sequence diagrams. After that, the specification of the scheduling for the concurrent use of services is introduced as global invariants distributed over the components. The final timed design is then verified formally against the non-functionality requirements.

We are planning to add in our model the concept of interaction protocol proposed by Jifeng et al [11] in order to give a more complete model for schedule invariant. Also, there is a question if our verification and analysis techniques can be supported by theorem provers or model-checkers. For this purpose, we are planning to define explicitly the EDC timed behaviour for timed designs. Then a study on integration of our model with Duration-Calculus-embedded B [3] can lead to the use of B support tools such as AtelierB [16] or BRILLANT [4] for the validation and verification purpose.

Acknowledgment

The authors are grateful to Vincent Poirriez and anonymous reviewers for their helpful comments.

References

[1] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. Modeling and designing heterogeneous systems. *Concurrency and Hardware Design — Advances in Petri Nets, Volume 2549 of Lecture Notes in Computer Science / J. Cortadella, A. Yakovlev, G. Rozenberg (Eds.)*, pages 228–273, Nov. 2002.

[2] Z. Chaochen and M. R. Hansen. *Duration Calculus*. Springer-Verlag, 2004.

[3] S. Colin, G. Mariano, and V. Poirriez. Duration calculus: A real-time semantic for b. In *ICTAC*, pages 431–446, 2004.

[4] S. Colin, D. Petit, V. Poirriez, J. Rocheteau, R. Marcano, and G. Mariano. Brillant : An open source and xml-based platform for rigourous software development. *sefm*, 0:373–382, 2005.

[5] V.-H. Dang. Toward a formal model for component interfaces for real-time systems. In *10th ACM Intl. Workshop on Formal Methods for Industrial Critical Systems*, pages 106–114, Lisbon, Portugal, September 5-6 2005. UNU-IIST Technical report 296.

[6] D. Distefano, J.-P. Katoen, and A. Rensink. On a Temporal Logic for Object-based Systems. In S. F. Smith and C. L.

Talcot, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 305–326. Klumer Acadeim Publisher, 2000.

[7] T. Genßler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. Müller, and C. Stich. Components for embedded software: the pecos approach. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 19–26, New York, NY, USA, 2002. ACM Press.

[8] I. J. Hayes. The real-time refinement calculus: A foundation for machine-independent real-time programming. In *Lecture Notes in Computer Science: 23rd International Conference on Applications and Theory of Petri Nets, Adelaide, Australia, June 24-30, 2002 / J. Esparza, C. Lakos (Eds.)*, volume 2360, pages 44–58pp. Springer Verlag, 2002.

[9] C. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice Hall, 1998.

[10] H. L. D. V. Hung. Concurrency and Schedulability Analysis in Component-based Real-Time System Development. Technical Report 341, UNU-IIST, P.O.Box 3058, Macau, July 2006.

[11] H. Jifeng, X. Li, and Z. Liu. Component-Based Software Engineering. Technical Report 330, UNU-IIST, P.O.Box 3058, Macau, October 2005. Presented at and published in the proceedings of ICTAC05, LNCS 3722, Dang Van Hung and Martin Wirsing (eds), Spinger 2005, pp. 70 - 95.

[12] H. Jifeng, Z. Liu, and L. Xiaoshan. Contract-Oriented Component Software Development. Technical Report 276, UNU-IIST, P.O.Box 3058, Macau, April 2003.

[13] H. Jifeng, L. Zhiming, and L. Xiaoshan. Modelling Object-oriented Programming with Reference Type and Dynamic Binding. Technical Report 280, UNU-IIST, P.O.Box 3058, Macau, May 2003.

[14] S. Sendall and A. Strohmeier. Specifying concurrent system behavior and timing constraints using OCL and UML. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 391–405. Springer, 2001.

[15] J. Sifakis, S. Tripakis, and S. Yovine. Building models of real-time systems from application software. In *Special issue on modeling and design of embedded systems*, volume 91(1) of *Proceedings of the IEEE*, pages 100–111, January 2003.

[16] STERIA - Technologies de l'Information, Aix-en-Provence (F). *Atelier B, Manuel Utilisateur*, 1998. Version 3.5.

[17] A. Tesanovic. *Developing Reusable and Reconfigurable Real-Time Software Using Aspects and Components*. PhD thesis, Linkopings University, February 2006. Dissertation No. 1005.

[18] H. Toetenel, E. Roubtsova, and J. van Katwijk. A Timed Automata Semantics for Real-Time UML Specifications. In *IEEE Symposia on Human-Centric Computing Languages and Environments (HCC'01), Visual Languages and Formal Methods (VLFM'01)*, pages 88–95, Stresa, Italy, September 5-7 2001. IEEE Computer Society.