Toward a Formal Model for Component Interfaces for Real-time Systems

Dang Van Hung
United Nations University
International Institute for Software Technology
P. O. Box 3058, Macau
dvh@iist.unu.edu

ABSTRACT

We give a model of component interface for real-time component based systems. We extend the specification of a method with a time constraint which is a relation between the resource availability and the amount of time spent to perform the method. We define a contract to include method specification, and define a component as an implementation of a contract. This implementation may require services from other components with some assumptions about the schedule for the use of shared methods and resources with the presence of concurrency. Our model supports the separation between functional and non-functional requirements, and the formal compositional verification of component-based real-time systems.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract, Formal methods*

General Terms

Theory

Keywords

Component-based systems, real-time constraints, extended duration calculus, unifying theory of programming

1. INTRODUCTION

Reusability is one of the advantages of component based development methods. However, when adding time features to the specification of a component, the reusability of the component is reduced if they are not flexible. This is typically true for real-time embedded systems, where components are based on specific hardware. If the timing specification of a component is fixed for that hardware, then the component cannot be used for different hardware. Furthermore, the real-time requirement of a component based system in general is achieved not only by the individual components but also by their interactions. In order to increase the flexibility for the timing specification of a component, we specify the timing of each of its methods as a relation of the time to carry out the method and the resources provided to the component. The implementation of a method may depend on services from other components which may be mutually exclusive with the presence of concurrency. Therefore, to guarantee its real-time services, a component needs an assumption about the real-time behaviour of the interaction of components in the system as well as the schedule for services of the system. To capture these kind of assumptions we introduce a schedule invariance to the specification of the component interface. Then, the component can provide correct service only if this invariance is satisfied. In the literature, there are a lot of work on the component interfaces, but not many of them take into account the timing specifications to our knowledge.

In this paper, we propose a model for component systems based on this idea using the notations from the Unifying Theory of Programming. With the flexible real-time specification for methods, with the assumption for the component interaction as schedule invariance interface, our model supports the formal compositional verification and facilitates the schedulability analysis of component-based real-time systems. The formal verification for industrial safety critical applications plays an important role, but is very difficult to perform even with the assistance from tools. Therefore, the compositionality will help to reduced the complexity for that hard works, and encourages to carry out the formal verification.

The paper is organised as follows. In the next section, we present our formal model for real-time component interfaces and components. After that, in Section 3 we propose a formal semantics of concurrent threads in the active components. The last section is the conclusion of our paper.

2. A FORMALISM FOR COMPONENT INTERFACE SPECIFICATIONS

A component provides services to its clients. The services could be either data or methods. To specify timing features of a method in a flexible way, we assume a fixed set of integer variables $RES = \{res_1, \ldots, res_n\}$. 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

FMICS'05, September 5–6, 2005, Lisbon, Portugal.

will have a resource specification to specify the resource requirements for its implementation, which will be a predicate over the integer variables in RES. A method will need some time to perform, and this amount of time depends on the type and number of available resources. 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 the variable ℓ, the resource variables and the input variables for the method.

DEFINITION 1 (INTERFACE).
An interface $I = \langle Fd, Md \rangle$ consists of

- *Fd - a feature declaration which is a set of variables,*
- *Md - a method declaration which is a set of methods; each method $m \in Md$ is of the form $op(in, out)$, where in and out are sets of variables.*

A method in an interface is specified by a so-called "timed design" $\langle \alpha, FP, FR \rangle$, where α denotes the set of (program) variables used by the method, FP denotes the functionality specification, and FR denotes the non-functionality specification of the method. We follow the style in [6] to represent FP and FR (as in the unifying theory of programming by He and Hoare [5]):

- FP 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 which is the assumption on the initial value of variables in $\alpha \setminus out$ that the method can rely on when activated, and R is the post condition relating the initial observations to the final observations (represented by the primed variables in the set $\{x' | x \in \alpha \setminus (in \cup out)\}$ and variables in out). The Boolean variable ok is a special variable denoting the termination of the method, i.e. ok is true iff the method starts, and ok' is true iff the method terminates. 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. We borrow the notation $\widehat=$ from the B method for the definition of a name.
- FR 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 variables in RES, 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 variables in RES, α and ℓ.

The definition of FP in a timed design $\langle \alpha, FP, FR \rangle$ is exactly the same as in the Unifying Theory of Programming. We give an example to illustrate the meanings for FR. Let $\alpha \mathrel{\widehat=} \{x, y\}$, $FP \mathrel{\widehat=} x \geq 0 \vdash_f y^2 = x$ and $FR \mathrel{\widehat=} P133 + P266 = 1 \vdash_r ((P133 = 1 \Rightarrow \ell \leq 0.001) \wedge (P133 = 0 \Rightarrow \ell \leq 0.0005))$. 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 0.001 time units when performed by a $133Mhz$ processor, and it takes no more than 0.0005 time units when performed by a $266Mhz$ processor.

Refinement of timed designs

The definition of the refinement relation for the timed designs is just a small extension of the one for the designs as presented in UTP and also in [6]. 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 [6]. 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.

Sequential Composition

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 $FP \mathrel{\widehat=} \exists m \bullet FP_1(m) \wedge FP_2(m)$.
- $FR \mathrel{\widehat=} \exists \ell_1, \ell_2 \bullet (FR_1[\ell_1/\ell] \wedge FR_2[\ell_2/\ell] \wedge \ell = \ell_1 + \ell_2)$

Here and later, we use $F[x_1/x]$ to denote the expression resulting from the substitution of x_1 for x in the expression F. Note that we assume in this paper that all the resources are not consumable. Hence the same resources used for D_1 can be reused for D_2 when D_1 has terminated.

Disjoint Parallel Composition

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:

- $\alpha \mathrel{\widehat=} \alpha_1 \cup \alpha_2$, $FP \mathrel{\widehat=} FP_1 \wedge FP_2$
- $FR \mathrel{\widehat=} \exists \ell_1, \ell_2, r_1, r_2 \bullet (FR_1[\ell_1/\ell, r_1/r] \wedge FR_2[\ell_2/\ell, r_2/r] \wedge \ell = \max\{\ell_1, \ell_2\} \wedge r = r_1 + r_2$, where r_1 and r_2 are vectors of resource variables, and $r_1 + r_2$ are defined componentwise.

The condition $r = r_1 + r_2$ expresses that the number of resources are enough for performing D_1 and D_2 in parallel independently. The composed command terminates when both component commands terminate. To justify these two definitions, we can use the operational semantics for the programs defined as a labeled transition system $(\mathcal{S}, \longrightarrow, C)$, where each state $s \in \mathcal{S}$ is a tuple (v, r, t), v is a vector of values of program variables, r is a vector of values of resource variables, and t is a real number to indicate the real-time. C is the set of commands. Let the semantics of $c \in C$ be design $\langle \alpha, FP, FR \rangle$, where $FP = p \vdash_f R$ and $FR = p_r \vdash_n S$. Then, there is a transition $(v, r, t) \xrightarrow{c} (v', r', t')$ iff $p(v) \wedge R(v, v') \wedge r = r' \wedge p_r(r) \wedge \ell = t' - t \wedge S(\ell, r, v, v')$ according to the interpretation of designs. Defining the disjoint parallel composition and sequential composition in the obvious way in the label transition system coincides with the definition given above. It is obvious that like for untimed designs:

THEOREM 1. *The relation $\sqsubseteq$ is a partial order relation on the set of timed designs, and the disjoint parallel composition and the sequential composition are monotone according to this relation.*

DEFINITION 2 (TIMED CONTRACT). *A timed contract is a tuple $\langle I, Rd, MSpec, Init, Inv \rangle$, where*

- *$I = \langle Fd, Md \rangle$ is an interface*
- *Rd - a resource declaration, which is a subset of RES,*
- *$Init$ 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,*
- *$MSpec$ is method specification which associates each method $op(in, out)$ in Md with a timed design $\langle \alpha, FP, PR \rangle$, where $(\alpha \setminus (in \cup out)) \subseteq Fd$, and*
- *Inv is a predicate on the features in the contract (called contract invariance). Inv represents an invariant property of the value 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 $Init$.*

We want to emphasise here that the resource variables declared in Rd in a contract are internal (local) in the contract (and in the components - see below - that implement the contract). Inv in a contract expresses a property of the variables of the contract that it offers to the environment. In case the contract cannot guarantee any invariant property of its variables, Inv is *true*.

DEFINITION 3 (REFINEMENT OF CONTRACTS). *Timed contract*

$$Ctr_1 = \langle \langle Fd_1, Md_1 \rangle, Rd_1, MSpec_1, Init_1, Inv_1 \rangle$$

is refined by timed contract

$$Ctr_2 = \langle \langle Fd_2, Md_2 \rangle, Rd_2, MSpec_2, Init_2, Inv_2 \rangle,$$

(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),*
- *$Md_1 \subseteq Md_2$,*
- *For all methods* op *declared in Md_1*

$$Mspec_1(op) \sqsubseteq Mspec_2(op), \text{ and } Inv_2 \Rightarrow Inv_1.$$

We justify this definition as follows. Ctr_2 provide all services that Ctr_1 does, but may provide more. Ctr_2 should have at least the same resources as Ctr_1 does. The condition $Inv_2 \Rightarrow Inv_1$ says that the property of variables guaranteed by Ctr_1 is ensured by Ctr_2. Hence we can use Ctr_2 to replace Ctr_1 without losing any services.

Let $Ctr_i = \langle Fd_i, Md_i, Rd_i, Mspec_i, Init_i \rangle$, $i = 1, 2$ be timed contracts which have the compatible sets of features and methods, i.e. $f \in Fd_1 \cap Fd_2$ implies $Init_1(f) = Init_2(f)$ and $op \in Md_1 \cap Md_2$ implies $MSpec_1(op) \Leftrightarrow MSpec_2(op)$. The combination $Ctr_1 \cup Ctr_2$ is defined as:

$$\begin{aligned} Ctr_1 \cup Ctr_2 \quad = \quad & \langle (Fd_1 \cup Fd_2, Md_1 \cup Md_2), Rd_1 \cup Rd_2, \\ & Mspec_1 \cup Mspec_2, \\ & Init_1 \uplus Init_2, Inv_1 \wedge Inv_2 \rangle, \end{aligned}$$

where $(Init_1 \uplus Init_2)(x)$ is defined to be

$$\begin{cases} \max\{Init_1(x), Init_2(x)\} \\ \quad \text{if } x \in dom(Init_1) \cap dom(Init_2) \\ Init_1(x) \quad \text{if } x \in dom(Init_1) \setminus dom(Init_2) \\ Init_2(x) \quad \text{if } x \in dom(Init_2) \setminus dom(Init_1) \end{cases}$$

When $Ctr_1 \cup Ctr_2$ is defined, we say that Ctr_1 and Ctr_2 are composable. Note that when combining two contracts, the amount of resources available for the combined one is defined as the maximal of the component contracts. This definition reflects our view that a method in the combined contract have at least the same time performance as it has in the component contracts, provided the following well-formedness is satisfied. The well-formedness means that a better timed performance is achieved if more resources are provided, and is formalised as:

A timed design $\langle \alpha, FP, FR \rangle$ is said to be well-formed iff FR satisfies

$$\forall r, r_1 \bullet (r \geq r_1 \Rightarrow (FR[r/RES] \Rightarrow FR[r_1/RES])),$$

where r and r_1 are vectors of values of resource variables (recall that RES is the vector of resource variables, and FR is a relation on RES, ℓ and α). For the definition of the refinement of timed contracts to be meaningful, we assume that all the timed designs for the specification of contracts are well-formed.

THEOREM 2. *Let Ctr_1, Ctr_2 be composable timed contracts in which the specification of all methods are well-formed. Then $Ctr_i \sqsubseteq Ctr_1 \cup Ctr_2$ for $i = 1, 2$.*

PROOF. By direct check from the well-formedness of the specifications for methods and the definition of the timed design refinement. □

Now we want to formalise the concept of component. Intuitively, a passive component is an implementation of a contract using services from other passive components via their contract. For the simplicity of presentation, we do not introduce the concept of private methods and private features, and use the simple architectural style with the client/server initiative, and synchronous communication. Our model can be extended to the general case easily. Recall that we are dealing with real-time methods. The implementation of a method may invoke other methods in other components. The invocation of methods in other component may need some extra time for handling the concurrent use of the methods. This is because that when there are concurrent calls to a method, the system needs a scheduler to schedule the uses of the method, which may force a call to wait. We assume that there is a scheduler in the system. This scheduler may be centralized or distributed. We try to incoporate only the needed information about the scheduler into components by using a schedule invariance $Sinv$. As with the set of resource names, we fix a set of global variables Π that are used by the scheduler. Each $v \in \Pi$ may correspond to a call from a component C for a service from a component Q. The scheduler uses the variables in Π to schedule for calls from components based on the schedule invariant $Sinv$. We also introduce a set Dep of component names in the declaration of a component $Comp$. Dep is a finite set of components that $Comp$ depends on. The idea is that when the implementation of a method op in $Comp$ has a call to a method in a component C then this call should be sent to the scheduler for scheduling. The scheduler bases on the current requests to resolve any conflict and may force some calls to wait a certain amount of time.

Definition 4 (Passive Components). *A real-time passive component is a tuple* $Comp = \langle Ctr, Dep, SDep, Mcode, SInv\rangle$, *where Comp is identified with the name of the component, consisting of*

- *a contract* $Ctr = \langle\langle Fd, Md\rangle, Rd, Mspec, Init, Inv\rangle$.
- *a set Dep of component names, each element of Dep is the name of other components that Comp depends on.*
- *SDep is the set of variables in* Π *(representing the interaction with the scheduler).*
- *SInv is a predicate on the variables* $v \in SDep$ *(to express the assumption about information that the scheduler can rely on when a method in Comp is called).*
- *Mcode assigns to each method op in Md a design built from basic operators (as well understood or defined in [7] with a suitable time consumption assumption as time and resource specification) and the method calls of the form* $call(Comp, C, op_1)$*, where* op_1 *is a method in a component C in Dep (see below). Note that method names, resource variables and local variables used in the specification and implementation of a method* $op_1(in, out)$ *in a passive component C (with the name C) are local in C, and are prefixed by "*$C\centerdot$*" to avoid the confusion with the variables used in other passive components. Let Env denote the predicate*

$$\wedge_{U\in Dep}(Inv(Ctr(U)) \wedge SInv(U))$$

(here and below we use $Ctr(U)$ *to denote the contract of component U,* $Inv(Ctr(U))$ *to denote the invariant of the contract of component U,* $Dep(U)$ *to denote the set of component names that U depends on, and* $SInv(U)$ *to denote the system schedule invariant of component U). The following condition should be satisfied by Mcode:* $Env \models (Mspec(op) \sqsubseteq Mcode(op))$*, and Inv is preserved by any operation used in Mcode.*

Let $C \in Dep$*, and* $op \in C$*. Then* $call(Comp, C, op)$ *is defined as* $Schedule(Comp, C)||C\centerdot op$*, where* $Schedule(Comp, C)$ *is a design using variables in* $SDep(C)$ *(the value of these variables represent the current calls to a method in C; we expect that the precondition of* $Schedule(Comp, C)$ *is implied by* $SInv(C)$*). From the disjoint parallel rule,* $Schedule(Comp, C)||C\centerdot op$ *implies the functional specification of* $C\centerdot op$*, but may need more time to perform.*

Contract Ctr is said to be implemented by Comp.

In the definition of component $Comp$, it requires that $Mspec(op) \sqsubseteq Mcode(op)$ for every method op in the contract of $Comp$ under the assumption $\wedge_{U\in Dep}(Inv(Ctr(U))\wedge SInv(U))$. In words, this means that provided that all the components that $Comp$ depends on ensure their invariants, any method of component $Comp$ is implemented correctly. Also, we require that any operation in $Comp$ should ensure the invariants of $Comp$. Therefore, op can be used as a proper service with the specification $Mspec(op)$. How to make sure that $\wedge_{U\in Dep}(Inv(Ctr(U)) \wedge SInv(U))$ is guaranteed? The implementation of op relies on the methods in the components with names in Dep. But the implementation of those methods may eventually rely on op. This situation may cause circular reasoning, and may cause op to be implemented incorrectly. This situation will not happen if we have the well-implementedness for the methods defined as follows.

Definition 5. *Well-implemented methods are defined recursively as*

1. *if op is a method in a component with the code* $Mcode(op)$ *composed from the basic commands, then op is well-implemented*
2. *if* op *is a method in a component with the code* $Mcode(op)$ *composed from the basic commands and method-calls for a well-implemented method, then op is well-implemented.*

So, well-implemented methods do not contain recursive method calls, although methods which contain recursive method calls may always terminate and have well-defined semantics.

Let $Comp = \langle Ctr, Dep, SDep, Mcode, SInv\rangle$. Let **Dep** be a binary relation defined as

$$\mathbf{Dep} \mathrel{\widehat{=}} \{(C_1, C_2)|C_2 \in Dep(C_1)\}$$

(i.e. $C_1\,\mathbf{Dep}\,C_2$ iff the implementation of a method in C_1 contains a call to a method in C_2). Let $\mathbf{Dep}^+$ and $\mathbf{Dep}^*$ be the transitive closure and the reflexive and transitive closure of **Dep** respectively.

By repeatedly replacing a method name by its implementation, we have:

Theorem 3. *Let* $Comp = \langle Ctr, Dep, SDep, Mcode, SInv\rangle$*, and let op be a well-implemented method of Comp. Then, there is a program text P without occurrences of method calls such that* $\bigwedge_{C\in \mathbf{Dep}^+(Comp)} Inv(C) \models Mspec(op) \sqsubseteq P$.

Combination of Components

Let $C_i = \langle Ctr_i, Dep_i, SDep_i, Mcode_i, Inv_i\rangle$, $i = 1, 2$ be passive components which have the composable contracts, and satisfy that $Mcode_1(op) \equiv Mcode_2(op)$ for all $op \in Md_1 \cap Md_2$. The combination $C_1 \cup C_2$ is defined as $\langle Ctr_1 \cup Ctr_2, Dep_1 \cup Dep_2, SDep_1 \cup SDep_2, Mcode_1 \cup Mcode_2, SInv_1 \wedge SInv_2\rangle$.

Let $\mathcal{U}$ be a finite set of passive components such that $\bigcup_{U\in\mathcal{U}} U\centerdot Dep \subseteq \mathcal{U}$ (recall that $U.Dep$ is the set of components that component U depends on). Let dependency graph of $\mathcal{U}$ be defined as the directed graph $D(\mathcal{U}) \mathrel{\widehat{=}} (\mathcal{U}, \mathcal{A})$, where $(U, V) \in \mathcal{A}$ iff $V \in U\centerdot Dep$. $\mathcal{U}$ is *well structured* iff its dependency graph has no cycle. A passive component U is said to be *self-contained* iff $U\centerdot Dep = \emptyset$.

THEOREM 4. *If $\mathcal{U}$ is well-structured, any method in a component $U \in \mathcal{U}$ is well-implemented.*

Remark

- The methods in components are defined as designs with preconditions, post conditions and relations on the amount of time to execute the methods and the resource availability. This is suitable for specifying the termination systems, but is not powerful enough to express the behaviour of nonterminating programs or reactive systems.

- The definition of a component $Comp$ requires that $Mspec(op) \sqsubseteq Mcode(op)$ under the assumption

$$\bigwedge_{U \in Dep} (Inv(Ctr(U)) \wedge SInv(U)).$$

  The condition $Inv(Ctr(U))$ is on the variables used to implement the functionality specification for the method op, and is guaranteed by all components U. The condition $SInv(U)$ is on the variables in $SDep(U)$ used by the scheduler only, and is used to implement the non-functional specification of the method. Therefore if $SInv(U)$ is verified as a global invariant for the corresponding untimed system (which has more untimed behaviours than the timed system), it must be an invariant of the timed system as well. The verification of the invariant $SInv(U)$ for the corresponding untimed system can be done with classical techniques. For example, when scheduling is unnecessary (e.g. the parallel usages of services are allowed, or services are called by only one component at a time, $SDep(U) = \emptyset$ for all U), then, and we can have $Schedule(Comp, C) = \langle \emptyset, skip, \ell = 0 \rangle$ (later we will assume that computations always take time, hence, the time specification for the scheduler in this case should be changed to $\ell > 0 \wedge \ell \leq d$ where d is the smallest amount of time needed to perform a command under the assumption about resources in the system). The precondition for $Schedule(Comp, C)$ is $true$, and hence $SInv(C)$ can be $true$, which is a trivial invariant. As another example, assume that the scheduler uses the 'first in first service' ($FIFO$) policy, and the maximal amount of time that a component uses a service of component $Comp$ each time is a, and that at most n other components may use services of $Comp$. Then we can have $Schedule(Comp) = \langle SDep(U), FP, \ell \leq n \times a \rangle$. We leave FP unspecified here. Whether there are concurrent calls to a component or not depends on if there are concurrent active methods in the system. The latter depends on if the language allows a method to be implemented with parallel commands or if there are more than one thread running in parallel in the system. We will discuss more about this aspect later.

From the discussion in the remark, it is reasonable to define that a component $comp_1$ is refined by a component $Comp_2$ if and only if $Comp_2$ is better than $Comp_1$ in the sense that $Comp_2$ provides more services than $Comp_1$, but needs less services than $Comp_1$, and the schedule condition needed in $Comp_2$ is looser than in $Comp_1$ (i.e. $Comp_2$ has stronger invariants for the scheduler, hence the scheduler working for $comp_1$ should work for $Comp_2$).

DEFINITION 6 (REFINEMENT OF COMPONENTS). *Let $Comp_i = \langle Ctr_i, Dep_i, SDep_i, Mcode_i, SInv_i \rangle$, $i = 1, 2$ be passive components. $Comp_1$ is said to be refined by $Comp_2$ (denoted by $Comp_1 \sqsubseteq Comp_2$) iff*

- *$Ctr_1 \sqsubseteq Ctr_2$ ($Comp_2$ provides more services than $Comp_1$)*

- *$Dep_2 \subseteq Dep_1$, $SDep_2 \subseteq SDep_1$ and $SInv_1 \Rightarrow SInv_2$ ($Comp_2$ does not need more services from the system than $Comp_1$, and need weaker assumption about the system scheduling)*

Active Components

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 driven by either events from the environment or by their internal clocks. A thread T is defined as **always** D **follows** e, where e is an event which is a boolean expression, and D is a method. 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 invoked. The formal meaning of the operator **always** will be given in the next section using a real-time temporal logic.

DEFINITION 7. *A component based system is a set S of components such that for any active component $U \in S$, for any V such that U **Dep*** V, $V \in S$ holds.*

In a component based system, we can replace a passive component by a better component without any violation of the requirements.

THEOREM 5. *Let S be a component based system. Let $Comp_1$ and $Comp_2$ be passive components such that $Comp_1 \sqsubseteq Comp_2$, and let $Comp_1 \in S$. Let S_1 be obtained from S by replacing $Comp_1$ by $Comp_2$ and replacing each occurrence of the name $Comp_1$ in components in S by an occurrence of the name $Comp_2$. Then S_1 is also a component based system, and provides more services than S.*

PROOF. The only thing we need to prove is that after the replacement of the occurrences of the name $Comp_1$ by the occurrences of the name $Comp_2$, the resulting system is also a set of components, i.e. we have to show that for any method op in a contract of a resulting component C,

$$Mspec(op) \sqsubseteq Mcode(op)$$

under the assumption

$$\wedge_{U \in Dep(C)} (Inv(Ctr(U)) \wedge SInv(U)).$$

From Definition 6, it follows that

$$\begin{array}{c} Schedule(Comp, Comp_1) || Comp_1.op \sqsubseteq \\ Schedule(Comp, Comp_2) || Comp_2.op \end{array}$$

for any method op in $Comp_1$. Hence, from the monotonicity of operations in the used programming language according to the refinement relation, and from the fact that

$$SInv(Comp_2) \Rightarrow SInv(Comp_1)$$

we have that

$$Mspec(op) \sqsubseteq Mcode(op)$$

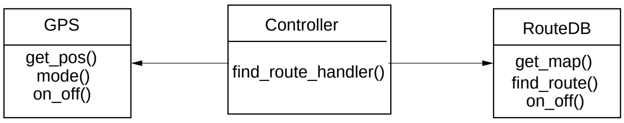

Figure 1: A Component Diagram for CNS.

holds under the assumption

$$\bigwedge_{U \in Dep(C)} (Inv(Ctr(U)) \wedge SInv(U))$$

for any method *op* in the contract of C in the system S_1.

$\square$

Example:

A Car Navigation System (CNS) [4] assists the driver of a car to navigate through an area. To interact with the driver, it consists of a display to show a map of the area around the car location, a keypad to enter commands (e.g. "display <map>", "zoom in/out" and "find a route to <destination>").

A component based design for CNS which is shown in Fig 1, consists of the following main components in which the **Dep** relation between components is represented by arrows in the figure).

1. Component *GPS*: This component has one method $get\_pos(out : src)$ with the specification

   $$\langle \{src\}, true \vdash_f src' = current\_position, 0 < \ell \leq 1 \rangle.$$

   We leave the code of this method unspecified here, but assume that the code does not contain any call to a method from other components. The only other component that may use this component is $Controller$. (We leave the resource unspecified here, and assume that the resource-precondition for $get\_pos(out : src)$ is $true$.

2. Component *RouteDB*: The resource declaration of this component consists of resource variables *memory* (initiated to 4 (Mb)) and *75MHz_processor* (initiated to 1). The component has two methods

   $get\_map(in : src, in : dstn, out : map)$, and
   $find\_route(in : src, in : dstn, out : route)$.

   The specifications of these methods are given respectively as

   $$\langle \{src, dstn, map\}, true \vdash_f map' = map\_for\_the\_area, 0 < \ell \leq 1 \rangle, \text{ and}$$
   $$\langle \{src, dstn, route\}, true \vdash_f route' = route\_to\_the\_destination, 75MHz_processor = 1 \wedge memory = 4 \vdash_n 0 < \ell \leq 11 \rangle.$$

   The only other component that may use this component is $Controller$. The code for $find\_route(in : src, in : dstn, out : route)$ is

   $$get\_map(src, dstn, map);\ compute(src, dstn, map, route).$$

   Assume that $compute(src, dstn, map, route)$ needs 10 seconds to perform using 4 Mb memory, and a 75 MHz processor, then this code is a refinement of the specification of $find\_route(in : src, in : dstn, out : route)$.

3. Active component *Controller*: This component has an event $find\_route\_command\_arrival$, and a method $find\_route\_handler$. The resource declaration of this component has variable $75MHz\_processor$ which is initiated to 1. The code for this method is
   $dstn := read\_dstn;$
   $(Schedule(Controller, GPS) || GPS\bullet get\_pos(src));$
   $(Schedule(Controller, RouteDB) ||$
   $RouteDB\bullet find\_route(src, dstn, route));$
   $display\_route(dstn).$
   Time specification of this method is $0 < \ell \leq 14$. Assume each of $dstn := read\_dstn$ and $display\_route(dstn)$ has the time consumption less than 1 using a 75 MHz processor. We assume that the commands $Schedule(Controller, RouteDB)$ and $Schedule(Controller, RouteDB)$ do not take time, i.e. $\ell = d$ (we cannot assume $\ell = 0$ because of our earlier assumption) is their post condition for timed specification (their precondition is given later as invariant for all three components), where d is the smallest amount of time to perform a command. It is derived directly from the sequential and parallel composition rule that the code of this method is the refinement of its specification.

   A thread of this component is
   always $find\_route\_handler$
   after $findroute\_command\_arrival$.

$\square$

So, in this model of component based systems we can use the Unifying Theory of Programming and additional rules for the real-time specification of designs to verify if a method is implemented properly or not. However, in order for this model to support the verification of the temporal and real-time properties, we have to give a formal meaning for threads, and a formal specification for real-time properties.

3. MODELING REAL-TIME PROPERTIES AND THREADS IN EXTENDED DURATION CALCULUS

Although the concept of timed designs defined in the previous section can be used to specify the relation between the starting state and the final state, and the execution time of a program in case it terminates, this concept is not strong enough to specify the behaviour of a program during its execution and the liveness properties such as threads of component. Especially, nonterminating programs cannot be specified as a timed design. Hence, we need a more powerful specification language which can model real-time properties and threads of component systems. In this section, we give a summary of our specification and verification techniques for real-time systems. Namely, we use Extended Duration Calculus (EDC) introduced by Zhou et al [1] as our specification language because of its simplicity and intuitivity. We will interpret (lift) all the program variables x in our component based systems as right continuous step functions of time $\mathbf{x}$ (note that only the typefaces are changed). We assume that we are given a set $\mathcal{M}$ of real functions and a set $\mathcal{B}$ of Boolean functions of time that we are interested in. Note that for an n-ary relation R over **Reals**, for $f_1, \ldots, f_n \in \mathcal{M}$, $R(f_1, \ldots, f_n)$ is a Boolean function defined by $R(f_1, \ldots, f_n)(t) = true$ iff $R(f_1(t), \ldots, f_n(t)) = true$. We define real functions and boolean functions over the set **Intv** of time intervals $\{[a, b] | a, b \in \mathbf{Reals}, a \leq b\}$ as follows.

- For any real function $f \in \mathcal{M}$, $\mathbf{b}.f$ and $\mathbf{e}.f$, when applied to an interval, returns the value of f at the beginning and the ending points of the intervals, respectively.

- For any Boolean function $b \in \mathcal{B}$, $\lceil b \rceil$ is a boolean function of intervals which is evaluated to *true* over an interval $[c, d]$ iff $d - c > 0$ and b is interpreted as *true* everywhere inside $[c, d]$ (i.e. everywhere in the open interval (c, d)).

- For any Boolean function $b \in \mathcal{B}$, $\lceil b \rceil^0$ is a boolean function of intervals is evaluated to *true* over an interval $[c, d]$ iff $c = d$ (i.e. $[c, d]$ is a point interval) and $b(c) = true$. So, $\lceil true \rceil^0$ is satisfied by $[c, d]$ iff $[c, d]$ is a point interval.

Formulas of EDC are interpreted as a mapping from **Intv** to $\{true, false\}$ and defined by:

1. A relation between real functions of intervals defined as above is a formula, which evaluates to true for an interval iff the values of the functions at this interval satisfy the relation.

2. A Boolean function of intervals defined as above is a formula, which evaluates to true for an interval iff the value of the function at this interval is true.

3. For formulas $R1$ and $R2$, $R1; R2$ is a formula which evaluates to true for an interval $[a, b]$ iff $R1$ evaluates to true for interval $[a, m]$ and $R2$ evaluates to true for interval $[m, b]$ for some $a \leq m \leq b$.

4. Boolean Connectives of formulas are formulas with usual semantics.

5. For a formula R, $\Diamond_r R$ is a formula which evaluates to true at interval $[a, b]$ iff R evaluates to true at interval $[b, m]$ for some $m \geq b$.

We use standard abbreviation in EDC:

$\Diamond\phi \stackrel{def}{=} true^\frown(\phi^\frown true)$ (ϕ is true for all subintervals)
$\Box\phi \stackrel{def}{=} \neg\Diamond\neg\phi$ (ϕ is true for a subintervals)

Since $\lceil b \rceil^{0\frown}\lceil p \rceil^0 \Leftrightarrow \lceil b \wedge p \rceil^0$ is valid in EDC for any Boolean functions b and p, we should assume that the computation always takes time to avoid conflict, and hence, for any design $\langle \alpha, FP, FR \rangle$ we assume that $FR \Rightarrow \ell > 0$ (without this assumption, the semantics of $x := x + 1$ cannot be defined because x would have different values at a time point). The EDC semantics and the untimed EDC semantics for a design $D \mathrel{\widehat{=}} \langle \alpha, FP, FR \rangle$ are defined respectively as the following formulas:

$$\mathcal{T}(D) \mathrel{\widehat{=}} \wedge_{x \in \alpha}(\mathbf{b.x} = x \wedge \mathbf{e.x} = x') \wedge FP \wedge FR \wedge \mathcal{TC}(D)$$
$$\mathcal{UT}(D) \mathrel{\widehat{=}} \wedge_{x \in \alpha}(\mathbf{b.x} = x \wedge \mathbf{e.x} = x') \wedge FP \wedge \ell > 0 \wedge \mathcal{UTC}(D)$$

Note the formula $\mathcal{UT}(D)$ only says about the temporal order between the changes of variables, but not time constraints. These formulas are satisfied by an interval $[a, b]$ iff the design D starts at time a (*ok* and preconditions are satisfied at time a) and terminates at time b, $b > a$ (ok' and post conditions are satisfied at time b). It requests for the first formula that the time consumption is $\ell = (b - a)$ and satisfies FR. $\mathcal{TC}(D)$ and $\mathcal{UTC}(D)$ are EDC formulas expressing the timed and untimed behaviour, respectively, of D inside the interval $[a, b]$, and is defined based on the code of D. We will not give the definition of $\mathcal{TC}(D)$ and $\mathcal{UTC}(D)$ here, and refer readers to [10] for the details.

Now we give formal semantics for events and threads in active components.

An event is a boolean expression b, and its occurrences should be isolated and not too frequent. So, for an event b it holds that:

$$\lceil b \rceil^{0\frown} true \Rightarrow \lceil b \rceil^{0\frown}\lceil \neg b \rceil^\frown true, \text{ and}$$
$$\exists\delta \bullet \Box(\lceil b \rceil^{0\frown}\lceil \neg b \rceil^\frown\lceil b \rceil^0 \Rightarrow \ell \geq \delta)$$

The semantics and the untimed semantics of a thread "**always** D **follows** e" are defined respectively as:

$$\Box(\lceil e \rceil^0 \Rightarrow \Diamond_r\mathcal{T}(D)), \text{ and}$$
$$\Box(\lceil e \rceil^0 \Rightarrow \Diamond_r\mathcal{UT}(D))$$

which means that event e always trigers the method D.

A real-time requirement R for a component based system S is an EDC formula on the events and other features of the active components of the system S. Requirement R is verified iff it is provable from the semantics of all the threads in the system provided that $SInv(C)$ holds during the time a method D in component C is performing, i.e. $\Box(\mathcal{TC}(C\centerdot D) \Rightarrow \lceil SInv(C) \rceil^0; \lceil SInv(C) \rceil; \lceil SInv(C) \rceil^0)$ (to guarantee that the methods used in the system are implemented correctly according to the definition of components) should be derivable from the timed semantics of the system. Note that the invariants $SInv$ are used as the precondition for the scheduler only, and have nothing to do with the untimed behaviour of the system (which does not depend on the scheduler). Hence, we have the following theorem which is the easiest way to verify the condition $\Box(\mathcal{TC}(C\centerdot D) \Rightarrow \lceil SInv(C) \rceil^0; \lceil SInv(C) \rceil; \lceil SInv(C) \rceil^0)$:

THEOREM 6. *If for all components C in a component based system S it is provable from the untimed EDC semantics of all threads in the system that $\Box(\lceil true \rceil^0 \Rightarrow \lceil SInv(C) \rceil^0)$ then $\Box(\lceil true \rceil^0 \Rightarrow \lceil SInv(C) \rceil^0)$ holds for the timed system.*

In general, some assumptions from the environment are needed to ensure the schedule invariant $SInv$ for components. Those assumptions could be the frequency of the trigger events, etc.

Example: Now we illustrate how our model works via the Car Navigation System in the previous example.

The thread of active component *Controller*
always *find_route_handler*
after *findroute_command_arrival*
has the EDC semantics:

$$\lceil findroute\_command\_arrival \rceil^0 \Rightarrow \Diamond_r\mathcal{T}(find\_route\_handler)$$

Let the invariant $SInv$ for scheduler for all components C be $w_C + r_C \leq 1$, where w_C is the number of calls to a method in C that are waiting, and r_C is number of calls that are on processing. This invariant for a component C just says that the concurrent use of a component is not allowed, and when a component is in use by another component, then there is no other request for a service from it.

One of the requirement for the CNS is that the deadline for finding a route is 15 seconds which is specified as the following EDC formulas

$$\lceil findroute\_command\_arrival \rceil^0 \Rightarrow \\ \diamond_r \ell \leq 15^\frown \lceil display\_route(dstn) \rceil^0$$

Because

$$\mathcal{T}(find\_route\_handler) \Rightarrow \\ \ell < 14^\frown \lceil display\_route(dstn) \rceil,$$

the requirement is implied by the semantics of the thread, provided that $SInv$ is provable from the timed semantics of the system. $SInv$ is provable if we have an assumption

$$\lceil findroute\_command\_arrival \rceil^{0\frown} \\ \lceil \neg findroute\_command\_arrival \rceil^\frown \\ \lceil findroute\_command\_arrival \rceil^0 \Rightarrow \ell > 15$$

A formal proof of this would involve the proof system of EDC which is not given here. But we do believe that it can be done with the assistance of a theorem prover like PVS. □

4. CONCLUSION AND RELATED WORK

This paper has presented a model for component-based real-time embedded systems. The model is an extension of the one for untimed systems proposed in He and Liu's work [6] to cover the timing and resource aspects of component-based systems. There are significant differences between this model and the original one. A component in this paper is defined to carry some architectural information to support 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 the verification of some real-time properties. This is especially useful for the development of safety critical systems. 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 requiremenst as well, particularly when the real-time requirements are in the form of deadline constraints. We can give a small extension to a specification language to support our model.

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 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.

In this paper, for simplicity, we have assumed a very simple way of communicating between components. The model can be extended for handling communication by introducing communication events and methods in the active components. There is still quite a lot of work to make our model more detailed. Also, there is a question if our verification and analysis techniques can be supported by any automatic tool. The answer is yes at least for the verification using a theorem prover like PVS. This will be in our future work.

We would like to mention here some work in the literatures related to this topic.

In [8, 12] a temporal logic is introduced to specify real-time properties in specification classes. Extended class diagrams and extended statechart 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. In [3], OCL is extended to specify real-time properties. In [9], timing properties are introduced as guards for transition, statecharts can specify real-time behaviour. 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. In [2], 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.

In [11], they 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 behaviour of the systems external environment and scheduling policy. Their model can be analysed by using time analysis techniques to check relevent real-time properies. 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 reusability of a component, we specify time as a relation between resource and time contraints.

Acknowledgement The author is grateful to the anonymous referees for their valuable comments that helped to improve this paper.

5. REFERENCES

[1] Zhou Chaochen, Anders P. Ravn, and Michael R. Hansen. An Extended Duration Calculus for Real-time Systems. Published in: *Hybrid Systems*, LNCS 736, 1993.

[2] Dino Distefano, Joost-Pierter Katoen, and Arend 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*, p. 305–326. Kluwer Academic Publisher, 2000.

[3] Stephan Flake and Wolfgang Mueller. A UML Profile for Real-Time Constraints with OCL. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2003*, volume 2460 of *LNCS*. Springer-Verlag, 2002.

[4] Dieter K. Hammer. *Software Architectures and Component Technology (Editor: Mehmet Aksit)*,

chapter Component-based Architecting for Distributed Real-time Systems. Kluwer, 2002.

[5] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice Hall, 1998.

[6] He Jifeng, Zhiming Liu, and Li Xiaoshan. Contract-Oriented Component Software Development. Technical Report 276, UNU-IIST, P.O.Box 3058, Macau, April 2003.

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

[8] E.E. Roubtsova, J. van Katwijk, W.J.Toetenel, and R.C.M.de Rooij. Real-Time Systems: Specification of Properties in UML. In *ASCI 2001 conference*, pages pp.188–195, Het Heijderbos, Heijen, The Netherlands, May 30 - June 1 2001.

[9] Shane Sendall and Alfred Strohmeier. Specifying concurrent system behavior and timing constraints using OCL and UML. In Martin Gogolla and Cris 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.

[10] François Siewe and Dang Van Hung. Deriving Real-Time Programs from Duration Calculus Specifications. Technical Report 222, UNU-IIST, P.O. Box 3058, Macau, December 2000. Published in the proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001), Livingston-Edinburgh, Scotland, 4–7 September 2001, LNSC 2144, Springer-Verlag, 2001, pp. 92–97.

[11] 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.

[12] Hans Toetenel, Ella Roubtsova, and Jan 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.