The United Nations University

UNU/IIST

International Institute for Software Technology

A Formal Model for Component Interfaces for Real-time Systems

Dang Van Hung

March 2004

UNU-IIST Report No. 296

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macau authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macau or visit UNU-IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

Chris George, Acting Director

The United Nations University

UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

A Formal Model for Component Interfaces for Real-time Systems

Dang Van Hung

Abstract

We give a formal model of component interface for real-time component based systems. We add to the specification of a method a time constraint which is a relation between the resource availability and the amount of time spent to perform the method. We define contract to include method specification, and define component as an implementation of a contract. This implementation may require services from other components with some assumption 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.

Dang Van Hung is a research fellow for the research project "Theories and Design Methods for Real-time Systems" since October 1995 being on leave of absence from Institute of Information Technology, Nghia Do, Cau Giay, Hanoi, Vietnam.

He has a PhD (equivalent) degree in Computer Science (concurrent systems) in 1988, Computer and Automation Research Institute (SZTAKI), Hungarian Academy of Sciences, Budapest, Hungary, and a BSc degree in Mathematics (numerical methods) in 1977, Hanoi University, Hanoi, Vietnam.

His research interests include Formal Techniques of Programming, Concurrent and Distributed Computing, Design Techniques for Real-Time systems.

e-mail: dvh@iist.unu.edu

Contents

1 Introduction

Real-time embedded systems now need heterogeneous notions to describe, have high complexity, and are compositional and reactive. A formal framework for unifying those heterogeneous notions will help the development for the systems more coherent. The paper concerns three aspects in the development of real-time embedded systems using real-time UML [4]:

- the separation of functional and non-functional constraints during the development,
- the formal specification and modelling of component interfaces, and
- the formal verification of a design with real-time UML.

The reason for the separation of functional and non-functional constraints (requirements) during the development of real-time embedded systems is that removing the non-functional constraints in a component will increase the reusability. In fact, real-time requirements are simply non-functional constraints that are separated from functional requirements. In practice, software engineers rely on Moore's law, which is the hope that the abundant availability of computing power will decrease the chance for conflicts related to computing and memory resources, and handle the non-functional requirements only during the implementation phase. But this reduces the compositionality. Hence, the non-functional requirements must be treated explicitly from the beginning. In fact, functional requirements are more compositional than non-functional requirements because the latter are emergent features that depend not only on individual components but also their interaction, their timing constraints and on the available resources.

In [7], it has been identified that component interface specifications should include:

- functionality specifications like operations (preconditions, post conditions), invariants, attributes and exception handling
- resources estimations, e.g. time constraints, resources constraints (memory) and dependencies
- interactions with messages (periodic, episodic, synchronous, asynchronous or single), protocols, autonomy, invariants.

The aims of component interface specifications are to ensure the correct cooperation between components, to facilitate the schedulability analysis and to verify the correctness of components. In order to support the separation of functional and non-functional constraints, all non-functional constraints must be defined separately from the functional specifications in component interface specifications and component interaction diagrams.

Note that real-time requirements are normally associated with usecases (or a view of a user described as a SIP (System Interface Protocol) as in [14], as represented by a sequence diagram).

Questions arise as to give:

- a formalism for component interface specifications that support the functionality and non functionality separation (using projection),
- a formalism for sequence diagrams for use cases to support the formal specification of real-time requirements,
- a proof system or model-checking algorithms for the verification problem, and
- a specification of timing constraints (for methods) as a relation between parameters characterising the implementation (resources used, hardware or software) and the lower and upper bounds for the execution time of the methods to support the verification of real-time requirements without reducing the reusability of the components.

In this paper we propose a solution for these problems. We organise the paper 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 logic for specifying real-time properties. Section 4 is devoted for 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 non-functional requirements of a method, 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 will have a resource specification to specify resource requirement for its implementation, which will be a predicate over the integer variables in RES. It needs some time to perform, and this amount of time depends on the type and number of resources available. We introduce a temporal variable ℓ to represent the amount of time spent to perform a method (in term of TLA [12], ℓ is exactly $now' - now$). 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, each method $m \in Md$ is of the form $op(in, out)$, where in and out are sets of variables.*

The specification of a method in an interface is a "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 [9] to represent FP and FR (as in the unifying theory of programming by He and Hoare [8]):

- 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 the 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.

- 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 Rd, 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 clear, but the one for FP needs an example to illustrate. 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, and if it takes no more than 0.001 time unite when performed by a $133Mhz$ processor, and it takes no more than 0.0005 time unite when performed by a $266Mhz$ processor.

The definition of the refinement for the timed designs is almost the same as for the designs as presented in [9]. 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, $r = (res_1, \ldots, res_n)$.

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

where $F[x/x_1]$ denotes the expression resulting from the substitution of x_1 for x in the expression F.

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:

- $\alpha \mathrel{\widehat{=}} \alpha_1 \cup \alpha_2$, $FP \mathrel{\widehat{=}} FP_1 \wedge FP_2$
- $FR \mathrel{\widehat{=}} \exists \ell_1, \ell_2, r_1, r_2 \bullet (FR_1[\ell/\ell_1, r/r_1] \wedge FR_2[\ell/\ell_2, r/r_2] \wedge \ell = \max\{\ell_1, \ell_2\} \wedge r = r_1 + r_2$

where r_1 and r_2 are vectors of resource variables, and $r_1 + r_2$ are defined componentwise. To verify the soundness of these two definitions, it is possible to use the operational semantics for the programs defined as a labeled transition system $(S, \longrightarrow, C)$, where each state $s \in S$ is a tuple (v, r, t), v is a vector of values of program variables, r is a vector of values of resource variables, 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 $FR = p_r \vdash_n S$. There is a transition $(v, r, t) \xrightarrow{c} (v', r', t')$ iff $p(v) \wedge R(v, v')$, $r = r'$ and $p_r(r) \wedge \ell = t' - t \wedge S(\ell, r, v, v')$. The disjoint parallel composition and sequential composition are defined in the obvious way in the label transition system. It is verified easily that these definitions coincide with the definition given above. A set of rules for other commands can be developed and verified using this operational semantics. 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 a 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. Inv should be satisfied by the value of the variables in the feature declaration Fd of the component at any time that accessible from outside, particularly by $Init$.*

We should emphasise here that the resource variables declared in Rd in a contract are internal in the contract, and in the components (see below) that implement the contract.

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 $Init_2|_{Fd_1} = Init_1|_{Fd_1}$, $Init_2|_{Rd_1} \leq Init_1|_{Rd_1}$.*
- *$Md_1 \subseteq Md_2$*
- *For all methods* op *declared in Md_1*

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

(Ctr_2 may provide more services, but should have at least the same resources as Ctr_1 does).

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. $f \in Fd_1 \cap Fd_2 \Rightarrow Init_1(f) = Init_2(f)$ and $op \in Md_1 \cap Md_2 \Rightarrow MSpec_1(op) \equiv MSpec_2(op)$. The combination $Ctr_1 \cup Ctr_2$ is defined as:

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

where

$$(Init_1 \uplus Init_2)(x) \;\widehat{=}\; \begin{cases} \max\{Init_1(x), Init_2(x)\} & \text{if } x \in dom(Init_1) \cap dom(Init_2) \\ Init_1(x) & \text{if } x \in dom(Init_1) \setminus dom(Init_2) \\ Init_2(x) & \text{if } x \in dom(Init_2) \setminus dom(Init_1) \end{cases}$$

It should be noticed here that when combining two constracts, the amount of resources available for the combined is defined as the maximal of the component contracts. This definition reflects our view that a method in the combined contract will have at least the same time performance as it has in the component contracts, provided the following well-formedness is satisfied.

A timed design $\langle \alpha, FP, FR \rangle$ is said to be well-formed iff FR satisfies $\forall r, r_1 \bullet (r \geq r_1 \Rightarrow (FR(RES/r) \Rightarrow FR(RES/r_1)$, where r and r_1 are vectors of resource variables' value (recall that RES is the vector of resource variables, and FR is a relation on RES, ℓ and α). So, a well-formed design will provide the better timed performance if more resources are provided. 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 with well-formed specifications for methods. 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. □

A passive component is an implementation of a contract using services from other passive components via their contract. We are going to formalise this concept. 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. 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. We assume that there is a scheduler in the system to handle this matter. This scheduler may be centralized or distributed. We introduce a set Dep of component names in the declaration of a component $Comp$ for that purpose. Like for the set of resource names, we fix a set of global variables Π to express the requests for components from a component, 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 a calls from components. 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 the conflict and may force some calls to wait a certain amount of time.

Definition 4 (Passive Components) *A real-time passive component $Comp = \langle Ctr, Dep, SDep, Mcode, SInv \rangle$, where $Comp$ is identified with the name of component, consists 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 component that $Comp$ depends on.*
- *$SDep$ is the set of variables in Π (representing calls from other components that may have a call to a method in Md of $Comp$ and acted on by 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 defined in [11] with 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 "$C.$" 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:
  $Schedule(Comp, C)||C.op$, 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, $Schedule(Comp, C)||C.op$ implies the functional specification of $C.op$, 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 $\wedge_{U \in Dep}(Inv(Ctr(U)) \wedge SInv(U))$. Therefore, op can be used as a proper service with the specification $Mspec(op)$. The implementation of op relies on the methods in the components with name in in Dep. But the implementation of those methods may eventually relies on op. This situation may cause circular reasoning, and may make op not implemented properly. 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 $\mathbf{Dep} \triangleq \{(C_1, C_2) | C_2 \in Dep(C_1)\}$ (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_1). Let

$\mathbf{Dep}^+$ and $\mathbf{Dep}^*$ be the transitive closure and the reflexive and transitive closure of **Dep** respectively.

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

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

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 $\cup_{U \in \mathcal{U}} U.Dep \subseteq \mathcal{U}$. 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 $V \in U.Dep$. $\mathcal{U}$ is *well structured* iff its dependency graph has no cycle. A passive component U is said to be *self-contained* iff $U.Dep = \emptyset$.

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 $\wedge_{U \in Dep}(Inv(Ctr(U)) \wedge SInv(U))$. 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 change 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 only 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 threads 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 $Comp_2$ iff $Comp_2$ is better than $Comp_1$ in the sense that $Comp_2$ provides more service 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).

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* $SInv_2 \Rightarrow SInv_1$ *($Comp_2$ does not need more services from the system than $Comp_1$, and ensures stronger invariants)*

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 component based system, and provides more services than S.*

Proof:

The only thing we need to proof is that after the replacement of the occurrences of the name $Comp_1$ by the occurrences of the name $Comp_2$, the resulting systems 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
$Schedule(Comp, Comp_1)||Comp_1.op \sqsubseteq Schedule(Comp, Comp_2)||Comp_2.op$
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 $SInv(Comp_2) \Rightarrow SInv(Comp_1)$

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

holds under the assumption $\wedge_{U \in Dep(C)}(Inv(Ctr(U)) \wedge SInv(U))$ for any method op in the contract of C in the system S_1.

□

So, in this model of component based systems we can use UTP 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 formal meaning for threads, and formal specification for real-time properties.

3 Interval Logic with Time Functions

In this section, we give a summary of our specification and verification techniques for real-time systems. Namely, we use Extended Duration Calculus (EDC) as our specification language. Extended Duration Calculus is just an Interval Logic with chop in which temporal variables and temporal propositional letters are defined via functions of time. We refer the readers to [5, 2] for more details of interval logic and EDC.

Interval Logic with Chop (Dutertre 95) and Neighborhood Operator (Zhou and Hansen, 1996) We are given the following symbol sets: the set $GVar$ of global variables

$x, y, z, ...$, independent of *time*; the set $Tvar$ of temporal variables $v, v_1, v_2, ...$ interpreted as functions of time intervals, in which ℓ is an interval function denoting interval length (which coincides with the meaning given to it in the previous section); the set $FSymb$ of global function symbols $f, g, ...$ (such as *constants* and $+, -, *, ...$), and the set $RSymb$ of global relation symbols $G, H, ...$ (such as $true$ and $false$, and $=, \geq, ...$) which have standard meaning; and the set $PLetter$ of temporal propositional letters $X, Y, ...$ which will be interpreted as truth-valued functions of time intervals .

Terms θ and formulas ϕ are built using the following grammars

$$\theta ::= x \mid \ell \mid v \mid f(\theta, ..., \theta)$$
$$\phi ::= X \mid G(\theta, ..., \theta) \mid \neg\phi \mid \phi \vee \phi \mid \exists x.\phi \mid \phi^\frown\phi \mid \Diamond_r\phi$$

In addition to the standard abbreviations, we denote

$$\Diamond\phi \stackrel{def}{=} true^\frown(\phi^\frown true) \qquad \Box\phi \stackrel{def}{=} \neg\Diamond\neg\phi$$

The semantic of terms and formulas are defined as follows. Let **Intv** denote the set of time intervals. Let a value assignment $\mathcal{V}$ ($\mathcal{V}: GVar \rightarrow$ **Reals**) for global variables, an Interpretation $\mathcal{I}$ for the given symbols be given. The semantics of a term θ is a mapping $\theta_\mathcal{I}$: **Intv** $\rightarrow$ **Reals**. The semantics for a term of the kinds x or v, is exactly the one given by $\mathcal{V}$ and $\mathcal{I}$. For a term $\theta = f(\theta_1, ..., \theta_n)$, $\theta_\mathcal{I}([a, b])$ is defined defined as $f(c_1, ..., c_n)$ where c_i is $(\theta_i)_\mathcal{I}([b, e])$ for $i = 1, ..., n$.

The semantics of a formula ϕ is a mapping $\mathcal{I}[\![\phi]\!]$: **Intv** $\rightarrow$ $\{true, false\}$ defined as
$\mathcal{I}[\![X]\!]([b, e]) = true$ iff $X_\mathcal{I}([b, e]) = true$;
$\mathcal{I}[\![G(\theta_1, ..., \theta_n)]\!]([b, e]) = true$ iff $G(c_1, ..., c_n) = true$, where $c_i = (\theta_i)_\mathcal{I}([b, e])$ for $i = 1, ..., n$;
$\mathcal{I}[\![\phi^\frown\psi]\!]([b, e]) = true$ iff $\mathcal{I}[\![\phi]\!]([b, m]) = true$ and $\mathcal{I}[\![\psi]\!]([m, e]) = true$ for some $m \in [b, e]$.
$\mathcal{I}[\![\Diamond_r\phi]\!]([b, e]) = true$ iff $\mathcal{I}[\![\phi]\!]([e, m]) = true$ for some $m \geq e$.

For other cases of ϕ, the definition is just the same as in predicate calculus. ITL has a powerful proof system which is relatively complete and very useful in the formal verification of real-time systems. We refer readers to [1, 5] for the details.

Extended Duration Calculus 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 time-function-based temporal variables and temporal propositional letters for the interval logic as follows.

- For any real function $f \in \mathcal{M}$, $\mathbf{b}.f$ and $\mathbf{e}.f$ are temporal variables interpreted as the value of f at the beginning and the ending points of intervals, respectively.

- For any Boolean function $b \in \mathcal{B}$, $\lceil b \rceil$ is a propositional letter 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 $b \in \mathcal{B}$, $\lceil b \rceil^o$ is a propositional letter which 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. As a standard denotation in DC, $\lceil true \rceil^0$ is abbreviated as $\lceil\ \rceil$.

A formula in Interval Logic in which only time-function-based temporal variables and temporal propositional letters are allowed, is called an EDC formula.

4 Modeling Real-time Properties and Threads in ITL

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). Since $\lceil b \rceil^0 \frown \lceil p \rceil^0 \Leftrightarrow \lceil b \wedge p \rceil^0$ is valid in ITL, 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 $FR \Rightarrow \ell > 0$. The ITL semantics and the untimed ITL 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 method 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 request for the first formula that the time consumption is $\ell = (b - a)$ and satisfies FR. $\mathcal{TC}(D)$ and $\mathcal{UTC}(D)$ are ITL 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 [15] 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:

$$\lceil b \rceil^0 \frown true \Rightarrow \lceil b \rceil^0 \frown \lceil \neg b \rceil \frown true$$
$$\exists \delta \bullet \Box(\lceil b \rceil^0 \frown \lceil \neg b \rceil \frown \lceil b \rceil^0 \Rightarrow \ell \geq \delta)$$

In an active component, together with the event declaration, we introduce event specification *ESpec* which assigns to an event e a boolean expression as semantics for e and ITL formula as assumption for the occurrences of e.

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

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

A real-time requirement R for a component based system S is an ITL 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.
$\Box(\mathcal{TC}(C.D) \Rightarrow \lceil SInv(C) \rceil^0; \lceil SInv(C) \rceil; \lceil SInv(C) \rceil^0)$
(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 $\Box(\mathcal{TC}(C.D) \Rightarrow \lceil SInv(C) \rceil^0; \lceil SInv(C) \rceil; \lceil SInv(C) \rceil^0)$:

Theorem 6 *If for all component C in a component based system S $\Box(\lceil\ \rceil \Rightarrow \lceil SInv(C) \rceil^0)$ is provable from the untimed ITL semantics of all threads in the system, $\Box(\lceil\ \rceil \Rightarrow \lceil SInv(C) \rceil^0)$ 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.

Now we illustrate how our model works via a simple example.

Example: A Car Navigation System [7] 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>"). UML like use case diagram, class diagram and sequence diagram are shown in Figs 1, 2 and 3.

A component based design for CNS consists of the following main components

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

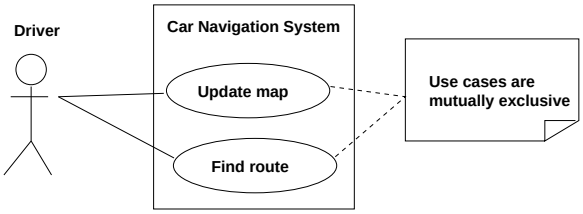

Figure 1: Usecases for CNS

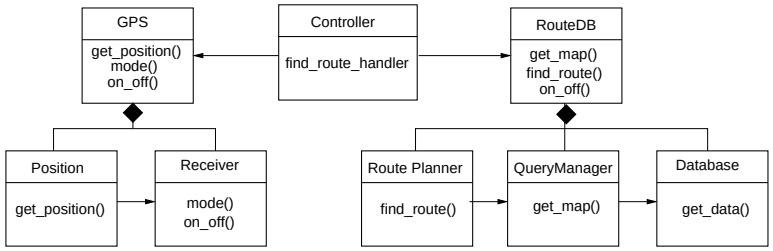

Figure 2: Class Diagram for CNS

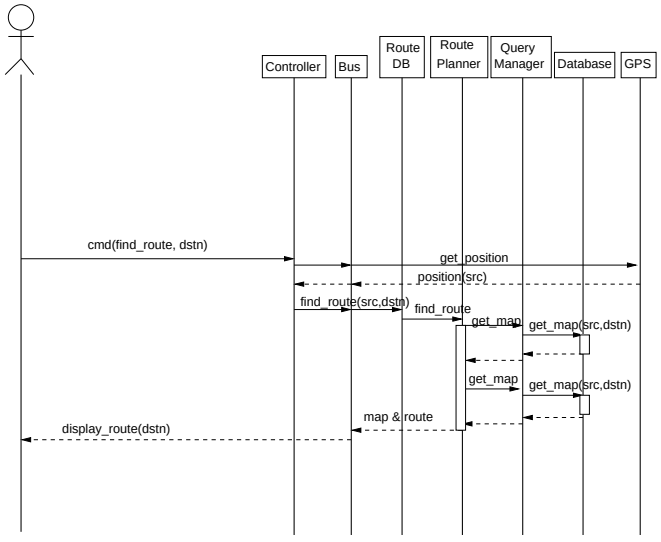

Figure 3: A Timed Sequence Diagram for "Find Route"

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
$\langle\{src, dstn, map\}, true \vdash_f map' = map\_for\_the\_area, 0 < \ell \leq 1\rangle$ 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 declaration $find\_route\_command\_arrival$, 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
$dstn := read\_dstn; Schedule(Controller, GPS)||GPS.get\_pos(src);$
$Schedule(Controller, RouteDB)||RouteDB.find\_route(src, dstn, route);$
$display\_route(dstn)$.
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 $Schedule(Controller, RouteDB)$ and $Schedule(Controller, RouteDB)$ do not take time, i.e. $\ell = d$ (we cannot assume $\ell = 0$ because of our ealier assumption) is their post condition for timed specification (their precondition is given later as invariant for all three components), where d is the smallest ammount 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** $findroute\_command\_arrival$. ITL semantics of the thread is:

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

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 ITL 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 ITL which is not given here. But we do believe that it can be done with the assistant from a theorem prover like PVS.

5 Conclusion

This paper has presented a formal 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 [9, 10] to cover the timing and resource aspects of component-based systems. There are significant differences between this model and the original one. A component 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. Our model supports the separation between functionality specification from non-functionality specification of components, which can simplify the verification of the functionality requirements, and in many case can simplify the verification of non-functional requirement 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 sine 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 complementary to UML to support the timed design and the formal verification of the safety critical systems. With the separation of the non-functionality and functionality during the system development, we first use UML to design an untimed system that satisfy 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 for communication between components. The model can be extended for handling communication by introducing communication events and methods in the active components. There are still quite a lot of work to

make our model more detailed. This will be in our future work. We would like to mention here some work in the literatures related to this topic.

In [13, 16] a temporal logic is introduced to specify real-time properties in specification classes, and 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 [6], OCL is extended to specify real-time properties. In [14], 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 [3], a temporal logics 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.

References

[1] Zhou Chaochen and Micheal R. Hansen. *Duration Calculus*. Springer-Verlag, 2004.

[2] Zhou Chaochen, Anders P. Ravn, and Michael R. Hansen. An Extended Duration Calculus for Real-time Systems. Research Report 9, UNU-IIST, P.O.Box 3058, Macau, January 1993. Published in: *Hybrid Systems*, LNCS 736, 1993.

[3] Dino Distefano, Joost-Pierter Katoen, and Arend Rensink. On a Lemporal 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.

[4] Bruce Powel Douglass. *Real-Time UML, Developing Efficient Objects for Embedded Systems*. Addison Wesley Longman Inc., 1998.

[5] B. Dutertre. On First Order Interval Temporal Logic. Technical Report CSD-TR-94-3, Department of Computer Science, Royal Holloway, University of London, Egham, Surrey TW20 0EX, England, 1995.

[6] 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.

[7] Dieter K. Hammer. *Software Architectures and Component Technology (Editor: Mehmet Aksit)*, chapter Component-based Architecting for Distributed Real-time Systems. Kluwer, 2002.

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

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

[10] He Jifeng, Liu Zhiming, and Li Xiaoshan. Component Calculus. Technical Report 285, UNU-IIST, P.O.Box 3058, Macau, September 2003. Presented as an Invited Talk at Workshop on Formal Aspects of Component Software FACS'03, Satellite Workshop of the FM 2003, Pisa, Italy - September 8-9, 2003.

[11] 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.

[12] Leslie Lamport. *Specifying Systems - The TLA$^+$ Language and Tools for Haedware and Software Engineers*. Addison-Wesley, 2003.

[13] 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.

[14] 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.

[15] 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.

[16] 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.