Towards a Template Language for Component-based Programming

Dang Van Hung
International Institute for Software Technology
The United Nations University
Macao SAR, China

Pham Hong Thai
University of Technology
Vietnam National University
Hanoi, Vietnam

Abstract - *The aim of this work*[1] [2] *is to develop a template for component-based programs which can be used in different programming languages. We define the concepts interface, contract and component, and component combination. The definition can be used as the basis for the component template development. 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 resolving the conflict of shared method and resource uses with the presence of concurrency. The assumption is expressed as interaction protocols. We then give a guideline for extending the model for real-time component systems.*

Keywords: Unifying Theories of Programming; Component-based Programming; Contract; Interface; Interaction Protocol.

1 Introduction

Object-oriented and component-based techniques are becoming popular and widely used in modelling and design of complex software systems. They provide effective support to decomposition of an application into objects and components, that can be realized by reusing and extending existing designs and implementations. The analysis and verification of such composed system can also be easier because of the compositionality of the component architecture.

There are now a number of well established object-oriented and component-based technologies including CORBA, EJB, J2EE, COM, and .NET. Semi-formal and formal Modelling languages, such as UML [8], JML [9], and BIP [12], are becoming popular to support model-based development. However, these models either do not support high level of abstraction or have their own language and lack language notations that can be used in combination with different programming language. Furthermore, these models focus on functional contracts and design of components and objects, and do not yet provide enough support to modelling and analysis of general quality of services of systems from those of their components.

We look for a modelling technique that supports the specification of component systems at high level of abstraction, and can provide a basis for the development of a template language that can incorporate to different programming language to support component-based programming. The modelling technique rCOS [6, 10] developed in UNU-IIST could be such a modelling technique, but the interaction protocol in rCOS does not support the specification of the concurrency, and it lacks the treatment of the protocols for required interface.

In this paper, towards the development of a template language for component-based programming, we propose a model that is based on rCOS and can cope with the shortcomings mentioned above for rCOS.

Our paper is organised as follows. In the next section, we motivate our approach by looking back at the lessons learnt from the structured programming development 30 years ago. In Section 3 we propose an architecture for component systems. Our main component model is presented in Section 4. The remaining sections are the guidelines for extending the model to real-time component model, and conclusion of our paper.

2 From Structured Programming to Component-based Programming

In the seventies, a new programming method was introduced that increased the productivity of programmers significantly and has become a classical one. This method was known as structured programming method, and is still of use nowadays to write small program modules. The critical principle for this method is the restriction on the use of the jump command `goto`. By that time, a group in France introduced a template language to support the structured

[1]This work is partly supported by the research project No. 202906 granted by the State Research Programme on Natural Sciences of Vietnam

[2]The first author is on leave of absence from Institute of Information Technology, Nghia Do, Cau Giay, Hanoi

programming method called EXCEL. A program in EXCEL is considered as a black box with a unique entry and a unique exit. A basic assignment is an atomic black box. A compound black box is constructed from other black boxes using the sequential composition structure ";" , the conditional branching structure "IF-THEN-ELSE-FI" and the loop structure "DO-OD". No `goto` is allowed in EXCEL. Using the data structures and basic commands from FORTRAN as data structures and basic commands for EXCEL will result in so-called FORTRAN-EXCEL language. In FORTRAN-EXCEL a stack of Boolean values (called stack of contexts) is often used with a program to preserve the expressive power of the language because of the absence of the `goto` [7] command.

The structured programming is not suitable for the current development of information technology where software systems are growing quickly both in terms of size and complexity, which need programming techniques that have a better modularity and support the reusability and compositionality. However, a lot of things can be learnt from this programming method for the development of a component model:

- A black box in EXCEL is a functional service specified with precondition and post condition that can be reused in a different program.

- A structured method for the construction of a more complicated service from smaller ones. With Hoare logic, the specification of black boxes is calculated.

- Given a programming discipline, there are structures for modular construction which combined with different programming languages result in supporting languages for the discipline.

We do not think that a black box in EXCEL should be considered as a component because it shares program variables with all other black boxes of the program which means it interferes and depends too much on the environment, and does not support the reusability very much [11].

We want black boxes to be more independent. For this purpose, we try to divide the program variables into groups and distribute them among the black boxes in the way that the group of variables used intensively by the commands in a black box should be located in the box, and the boxes having the same variable group should be combined together to form a module with several functions (services) that act on the same set of variables. We do the rearrangement of the black boxes keeping the same connecting edges. This way of grouping of the boxes does not change the semantics of the program, but create a new way of modularization: each module has more autonomy (see Fig. 1). In summary, a module created in this way has: variables, several functions,

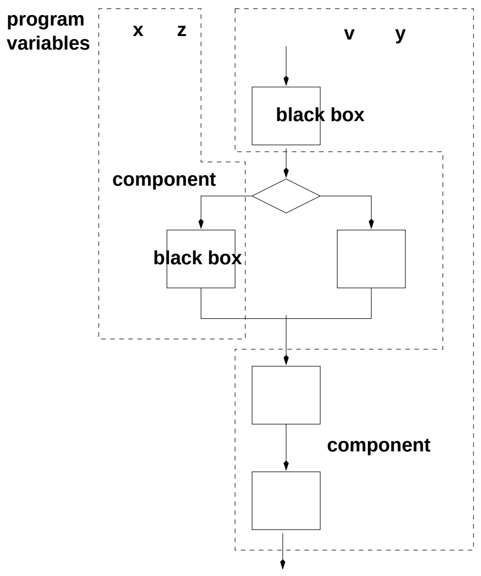

Figure 1. From EXCEL Boxes to Components

input ports, output ports, middle output ports, and middle input ports.

The middle output ports and middle input ports are for the case that a compound service has some sub-services that are located in different modules and the control flow (represented as directed edge) goes out from the module and later on comes back again. The main advantage of this kind of modules is that they can be used in a system just by plug and play, and hence, like in hardware, they are called components, and their variables, functions, ports form the "interface" for the component. So, the component interface is a new syntactical element to be used in a language that supports component programming.

To reuse a component, one needs to know what the component is doing. So, an invariant to indicate the property of variables (e. g. their type and initial value) of the component, and pre-, post-conditions for its functions (at the input and output ports) should be provided with its interface. The middle ports are connected with the functions corresponding to its sub black boxes for the component to operate correctly. These middle ports and their corresponding functions form the so-called "required interface", and the specification of those functions needed should also be provided with the interface. The specification of the functions of a component forms the so-called contract of the component.

We have come up with the clear concepts of components, their interface, and contract. Components have similar structure with classes in object-oriented programming, but have explicit interface, and hence explicit dependencies with others. The external functionality needed by a compo-

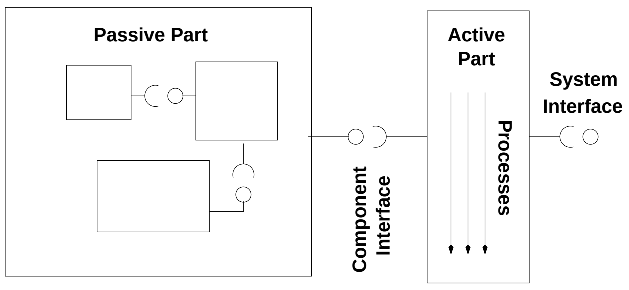

Figure 2. System Component Achitecture

nent is specified in the required interface of a component, and this makes the development of a component more independent from other components.

3 A Component-based System Architecture

In this section, we propose a component-based architecture for systems. With autonomy components, via their interface, we can put them together by connectors to form a bigger and bigger component with more and more services until it becomes closed (i.e. its required interface is empty), and that the services it provides are enough for us to satisfy our requirements specified as processes (use-cases). So, our system built in this way has two parts: the passive part that is a close component composed from a set of components, and the active part which is a set of reactive processes which are triggered by external events and using services from the passive part to satisfy the requests from the external actors of the system. This system architecture is depicted in Fig. 2.

Each process corresponds to a thread in a Java program. Multithreads are allowed. This means that there is a chance for more than one threads to access a service from a subcomponent in the passive part directly or via other services. How to handle this case to increase the performance and especially to avoid deadlock by concurrent accesses to common resources is a big problem. If this happen, even a precondition of a service is satisfied, the service does not terminate because of the deadlock. Therefore not every sequence of service-accesses should be allowed. Apart from the specification with pre- and post condition for services, a contract should also include a specification of the allowable sequences of service-accesses. This specification, which also plays as an assumption the component makes about its environment, is captured by so-called *interaction protocol*.

4 A Component Model

In this section, we formalise the concepts presented in the previous sections. Our model here is both a simplification and extension of rCOS developed by He, Liu and Li in [6] and its timed version by us in [4]. Here we extends rCOS with detailed protocols and the way components are interacting using protocols, and how to specify the concurrent accesses to a service in a protocol. These aspects are typically needed when the model is extended to handle non-functional properties.

4.1 Designs

As said in the previous section, a component provides services to its clients. The services could be either data or methods.

To specify the functionality for methods, we use the traditional UTP notations (He and Hoare [3]) in which a method is specified as a relation together with its signature of the form $op(in, out)$, where in and out are sets of variables.

Definition 1 *A design is a tuple $\langle \alpha, FP \rangle$, where α denotes the set of (program) variables used by the method, FP denotes the functionality specification of the method written in UTP notations.*

- *FP is a predicate of the form*

  $$(p \vdash R) \mathrel{\widehat{=}} (ok \wedge p) \Rightarrow (ok' \wedge R)$$

  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 borrow the notation $\widehat{=}$ from the B method for the definition of a name.

Refinement of designs

We recall the definition of the refinement relation for designs presented in UTP and also in [6]. A design $D_1 = \langle \alpha, FP_1 \rangle$ is refined by a design $D_2 = \langle \alpha, FP_2 \rangle$ (denoted by $D_1 \sqsubseteq D_2$) iff

$$(\forall ok, ok', v, v' \bullet FP_2 \Rightarrow FP_1)$$

where v, v' are vectors of the program variables.

Sequential Composition

Let $D_1 = \langle \alpha, FP_1 \rangle$ and $D_2 = \langle \alpha, FP_2 \rangle$ be designs. Then

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

where $FP \mathrel{\widehat{=}} \exists m \bullet FP_1(m) \wedge FP_2(m)$ with assuming that $FP_1 = FP_1(v')$ and $FP_2 = FP_2(v)$. 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.

4.2 Interfaces and Contracts

The signature for a component is its interface which specifies what services it provides and what services it requires from the environment. Contract is the specification of its interface. From the informal discussion in the previous sections, we give the formal definition of these two concepts in component-based programming as follows.

Definition 2 *An interface is a pair $I = (I_p, I_r)$, where $I_p = \langle Fd_p, Md_p \rangle$, and $I_r = \langle Fd_r, Md_r \rangle$. I_p is called provide interface of I, and I_r is called required interface of I.*

Definition 3 *A contract is a tuple $\langle I, Init, MSpec, Inv_p, Inv_r, Pro \rangle$, where:*

- *$I = (I_p, I_r)$ is an interface. Let $Md = Md_r \cup Md_p$, $Fd = Fd_r \cup Fd_p$.*
- *$Init$ is an initialization, which associates each variable in Fd and each local variable with a value of the same type.*
- *$MSpec$ is method specification which associates each method $op(in, out)$ in $Md = Md_r \cup Md_p$ with a design $\langle \alpha, FP \rangle$, where $(\alpha \setminus (in \cup out)) \subseteq Fd$.*
- *Inv_p and Inv_r are predicates on the provide features and required features, respectively, in the contract (called contract invariance). Inv_p represents an invariant property of the value of the variables in the feature declaration Fd_p that can be relied on at any time that it is accessible from other components. Hence, Inv_p is satisfied particularly by $Init$. Inv_r represents the property that required for the value of variables in Fd_r when are provided.*
- *Pro is a protocol which is a subset of $\{m?, m! | m \in Fd_p\}^* \cup \{m?, m! | m \in Fd_r\}^*$. Only the behaviours of which the projections on $\{m?, m! | m \in Fd_p\}^*$ and on $\{m?, m! | m \in Fd_r\}^*$ belong to Pro are allowed.*

The contract of a component expresses what the component expects from its environment and what it can offer to the environment.

The variables in Fd are read-only to the other contracts. Inv_p in a contract expresses a property of the variables of the contract that it offers to the environment, and hence should be guaranteed by any method of the contract.

Perhaps it is less clear what a protocol is, and how it relates to the specification of services. We give a clarification of this concept as one of the contribution of this paper. For a method m, denote $?m$ and $!m$ as the invocation (giving value of parameter) and termination (getting the service result) of m (we use CSP like communication style). Then, it is natural to required that each $?m$ should correspond to exactly one following $!m$, and vice-versa, each $!m$ should correspond to exactly one invoking action $!m$. For a method m, it may allow several threads to use m at the same time (e.g. several copies of m exists). In this case, for a $?m$ and its corresponding $!m$ there may be several other occurrences of $?m$ between them. The maximal number of invocations $?m$ with no corresponding termination at a time is the concurrency degree that m can offer and is specified clearly in the protocol. Normally, any method m can only be used mutual exclusively with itself and other methods in the component. In this case, the protocol can be specified as a regular expression $\{?m!m | m \in Md\}^*$. When all method m can be used mutual exclusively with itself and in parallel with other methods, the protocol can be specified as a regular expression $||_{m \in Md}\{?m!m\}^*$ where $||$ denotes the interleaving parallel composition operation.

Definition 4 *Contract $Ctr_1 = \langle (I_{p1}, I_{r1}), MSpec_1, Init_1, Inv_{p1}, Inv_{r1}, Pro_1 \rangle$ is refined by contract $Ctr_2 = \langle (I_{p2}, I_{r2}), MSpec_2, Init_2, Inv_{p2}, Inv_{r2}, Pro_2 \rangle$ (denoted $Ctr_1 \sqsubseteq Ctr_2$) iff:*

- *$Fd_{p1} \subseteq Fd_{p2}$, $Fd_{r2} \subseteq Fd_{r1}$, and $Init_2|_{Fd_{p1}} = Init_1|_{Fd_{p1}}$ (where for functions f and a set A, $f|_A$ denotes the restriction of f on A).*
- *$Md_{p1} \subseteq Md_{p2}$ and $Md_{r2} \subseteq Md_{r1}$*
- *For all methods* op *declared in Md_{p1} $Mspec_1(op) \sqsubseteq Mspec_2(op)$, and $Inv_{p2} \Rightarrow Inv_{p1}$.*
- *For all methods* op *declared in Md_{r2} $Mspec_2(op) \sqsubseteq Mspec_1(op)$, and $Inv_{r1} \Rightarrow Inv_{r2}$.*
- *$Pro_1|_{\{m?, m! | m \in Fd_{p1}\}} \subseteq Pro_2|_{\{m?, m! | m \in Fd_{p1}\}}$ and $Pro_2|_{\{m?, m! | m \in Fd_{r2}\}} \subseteq Pro_1|_{\{m?, m! | m \in Fd_{r2}\}}$.*

We justify this definition as follows. Ctr_2 provide all services that Ctr_1 does, even better, and may provide more, and Ctr_2 should need less and and worst services than Ctr_1 does. The condition $Inv_2 \Rightarrow Inv_1$ says that the property of variables guaranteed by Ctr_1 is also ensured by Ctr_2. In brief, contract Ctr_2 provides more and better services to and needs less services from its environment than contract Ctr_2. Hence we can use Ctr_2 to replace Ctr_1 without losing any service.

4.3 Combination of Contracts

Contracts can be combined in many ways to form a new contract. A difficulty for the calculation of a compound contract is the calculation of its protocol. This was left out in the theory of contracts in [13]. The simplest way to combine two contracts is to put them together if they have disjoint sets of features and methods.

Definition 5 *Let* $Ctr_i = \langle (I_{pi}, I_{ri}), Mspec_i, Init_i, Inv_{pi}, Inv_{ri}, Pro_i\rangle$, $i = 1, 2$ *be contracts that have disjoint sets of (required and provide) features and methods. The union of* Ctr_1 *and* Ctr_2 *is the contract* $Ctr_1 \cup Ctr_2 = \langle (I_{p1} \cup I_{p2}, I_{r1} \cup I_{r2}), Mspec_1 \cup Mspec_2, Init_1 \cup Init_2, Inv_{p1} \cup Inv_{p2}, Inv_{r1} \cup Inv_{r2}, Pro_1 \cup Pro_2 \cup (Pro_1 || Pro_2)\rangle$.

The only thing needs to be explained in this definition is how the protocol for the union contract is defined. When putting contracts together, because they are assumed to be independent, all of their methods and features can be used in parallel. The parallel composition $Pro_1||Pro_2$ specified precisely that situation. However, the compound contract should also allow methods in a individual component to be used in their original way.

Another way to combine contracts is to connect required methods of one contract to provide methods of the other. Let $Ctr_i = \langle (I_{pi}, I_{ri}), Mspec_i, Init_i, Inv_{pi}, Inv_{ri}, Pro_i\rangle$, $i = 1, 2$ be contracts which have the compatible sets of provided features and methods, the compatible sets of required features and methods, i.e. $f \in (Fd_{p1} \cap Fd_{p2}))$ implies $Init_1(f) = Init_2(f)$ and $op \in (Md_{p1} \cap Md_{p2}) \cup (Md_{r1} \cap Md_{r2})$ implies $MSpec_1(op) \Leftrightarrow MSpec_2(op)$. Assume that $I_{r1} \subseteq I_{p2}$, and $Inv_{p2} \Rightarrow Inv_{r1}$ and $Mspec_1(op) \sqsubseteq Mspec_2(op)$ for all $op \in Md_{r1}$.

The full plug of Ctr_1 to Ctr_2, denoted as $Ctr_1 >> Ctr_2$ is defined as:

$$\begin{array}{l} Ctr_1 >> Ctr_2 = \\ \langle (I_{p1} \cup I_{p2}, I_{r2}), Mspec_1|_{Md_{p1}} \cup Mspec_2, \\ Init_1|_{Fd_{r1}} \uplus Init_2, Inv_{p1} \wedge Inv_{p2}, Inv_{r2}, Pro\rangle \end{array}$$

where Pro is defined below, and $(Init_1 \uplus Init_2)(x)$ is defined to be

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

We should discuss here how the protocol Pro is calculated from $Pro_i, i = 1, 2$ for the contract $Ctr_1 >> Ctr_2$.

- First, the compound contract $Ctr_1 >> Ctr_2$ should also allow methods in a individual component be used in their original way. So, $Pro_1 \cup Pro_2$ should be included in Pro.
- The methods m from Ctr_2 that are not required by Ctr_1 can be used in parallel with methods in Ctr_1. So, Pro should include $Pro_1||(Pro_2 \cap \{!m, ?m | m \in Md_{p2} \setminus Md_{r1}\}^*$.
- The question is how a method m in $Md_{p2} \cap Md_{r1}$ is used with a method in Md_{p1}. The answer depends on the concurrency degree of m. The calculation for this situation is complicated, and is left open here. For safety, we do not allow them to run in parallel (less efficient).

So, we define

$$\begin{array}{rcl} Pro & = & Pro_1 \cup Pro_2 \cup \\ & & Pro_1||(Pro_2 \cap \{!m, ?m | m \in Md_{p2} \setminus Md_{r1}\}^*) \end{array}$$

When $Ctr_1 >> Ctr_2$ is defined, we say that Ctr_1 is connectible to Ctr_2. Note that when combining two contracts in this way, the resulting component may not be used to replace Ctr_1. The reason is that it might require somethings from the environment that Ctr_1 does not.

Theorem 1 *Let* Ctr_1 *be connectible to* Ctr_2*. If* Ctr_2 *is closed (i.e.* $I_{r2} = \emptyset$*) then* $Ctr_1 \sqsubseteq (Ctr_1 >> Ctr_2)$*.*

Proof: By direct check from the definition of the plug operator and the definition of the design refinement. □

The definition can be modified for the general case that $I_{r1} \not\subseteq I_{p2}$, and the theorem is still correct. We leave this to the readers as an exercise.

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 use a simple architectural style with the client/server initiative, and synchronous communication.

Definition 6 *A passive component is a tuple* $Comp = \langle Ctr, Mcode\rangle$*, where* $Comp$ *is identified with the name of the component, consisting of*

- *A contract* $Ctr = \langle (I_p, I_r), Mspec, Init, Inv_p, Inv_r, Pro\rangle$.
- $Mcode$ *assigns to each method* op *in* Md_p *a design built from basic operators (as well understood or defined in [10]) and the method in* I_r *such that the projection of the traces of this design in which any method* $op \in Md_r$ *is replaced by* $?op!op$ *(its starting and its ending actions), on* $\{?m, !m | m \in Md_r\}$ *are included in* Pro*. The following condition should be satisfied by* $Mcode$*:* $(Mspec(op) \sqsubseteq Mcode(op))$*, and* Inv_p *is preserved by any operation used in* $Mcode$*.*

- *The implementation of all methods m should be compatible with their concurrency degree described in Pro, i.e. either a method m is not a mutual exclusively used method, or a suitable number of copy of m is available.*

Contract Ctr is said to be implemented by $Comp$.

So, a component should be correct by its design, i.e. the implementation of their methods should be correct according to their specification. Its correctness can be verified separately from the environment.

Let $Comp_i = \langle Ctr_i, Mcode_i\rangle$, $i = 1, 2$ be components, such that $Ctr_1 >> Ctr_2$ is defined. Let $Mcode'_1$ be obtained from $Mcode_1$ by replacing each occurrence of $op \in (Md_{r_1} \cap Md_{p2})$ by $Mcode_2(op)$ in the range of $Mcode_1$. It is easy to check that the full plug operator defined on contracts is carried over to components:

Theorem 2 $\langle Ctr_1 >> Ctr_2, Mcode'_1 \cup Mcode_2|_{Md_2 \setminus Md_1}\rangle$ *is a component.*

Example: Let $double\_quadr$ be a component with a service $dquadr(in : real\ a, b, c; out : real\ x1)$ that gives a solution $x1$ of the quadratic in x^2 equation $ax^4 + bx^2 + c = 0$. The component $double\_quadr$ requires a service $quadr(in : real\ a, b, c; out : real\ x1)$ from another component which when called with the proper parameters a, b and c returns a solution of the quadratic equation $ax^2 + bx + c = 0$. Component $double\_quadr$ can be described as follows.

```
component $double\_quadr${
    provide method{
        name{ $dquadr(in : real\ a, b, c; out : real\ x1)$},
        specification{ $ac \leq 0 \vdash ax1^4 + bx1^2 + c = 0$}}
    required method{
        name{ $quadr(in : real\ a, b, c; out : real\ x1)$},
        specification{ $ac \leq 0 \vdash ax1^2 + bx1 + c = 0 \wedge x1 \geq 0$}}
    protocol{ $(?dquadr!dquadr)^* \oplus (?quadr!quadr)^*$}
    implementation{
        name{ $dquadr(real\ in : a, b, c, real\ out : x1)$},
        code{$quadr(in : a, b, c; out : x1)$;
        $x1 := sqrt(x1)$} }}
```
□

The active part consists of several processes and a required interface, that can be plugged to the passive part. A process is described by a program that uses the services from the passive part to react with events from the environment of the system. The events are not controlled by the system. So, interesting events from the environment and the system services together with its specification and interaction protocol form the contract between the system and its environment.

Definition 7 *A system interface is a pair $SI = (E, Fd, SMd_p)$, where SMd_p is a finite set of methods, Fd is a finite set of features, and E is a finite set of events.*

Definition 8 *A system contract is a tuple $SysCtr = \langle SI, SMSpec, Inv, Behav\rangle$, where*

- *$SI = (E, Fd, SMd_p)$ is a system interface.*
- *$MSpec$ is method specification which associates each method $op(in, out)$ in SMd_p with a design $\langle \alpha, FP\rangle$, where $(\alpha \setminus (in \cup out)) \subseteq Fd$, and*
- *$Behav$ is an external behaviour description which is a finite subset of $\{e, m | e \in E$ and $m \in SMd_p\}^*$. Each element of $Behav$ is called a process specification.*

We want to avoid the introduction of a logic in this paper and use elements of $Behav$ as system specification. An informal meaning for a sequence α in $Behav$ is that if the system environment offers the sequence of events as occurring in α then the system offers the sequence of services (methods) specified by α in that order. Elements of $Behav$ are described by program threads running in parallel.

Definition 9 *An active component $ActComp = \langle Ctr, SysCtr, Mcode\rangle$, consists of*

- *A contract*
  $Ctr = \langle (I_p, I_r), Mspec, Init, Inv_p, Inv_r, Pro\rangle$ with the empty provide interface, $I_p = (\emptyset, \emptyset)$.
- *A system contract*
  $SysCtr = \langle SI, SMSpec, Inv, Behav\rangle$.
- *A process implementation $Mcode$ assigns to each method op in SMd_p a design built from basic operators and the method in I_r such that the projection of the traces of this design in which any method $op \in Md_r$ is replaced by $?op!op$ (its starting and its ending actions), on $\{?m, !m | m \in Md_r\}$ are included in Pro. The following condition should be satisfied by $Mcode$: $(SMspec(op) \sqsubseteq Mcode(op))$ for all $op \in SMd_p$.*

A system in our component model is an active component plugged to a closed passive component.

Definition 10 *A system is a pair of an active component $ActComp = \langle Ctr, SysCtr, Mcode\rangle$ and a closed passive component $Comp = \langle Ctr', Mcode'\rangle$ such that $Ctr >> Ctr'$ is defined.*

So, a component system is a closed system which does not require any service from its environment, and provides its services to the environment as its reactions to the stimulus events from environment. The specification of a system is system contract $SysCtr$. Two systems are equivalent iff they have the same specification, i.e. they have the equivalent system contract. The following theorem shows the most important feature of component-based programming.

Theorem 3 *Let $S = (ActComp, Comp')$ be a system formed by active component $ActComp = \langle Ctr, SysCtr, Mcode\rangle$ and passive component $Comp' = \langle Ctr', Mcode'\rangle$. Let $Comp'' = \langle Ctr'', Mcode''\rangle$ be a passive component such that $Ctr' \sqsubseteq Ctr''$. Then, $(ActComp, Comp'')$ is also a system which is equivalent to S.*

5 Extension to Real-time Component Model

The model presented in the previous section can be extended with some timing features to become a component model for real-time systems. In this section, we discuss how this can be done. The critical part of the extension lies on real-time services, real-time interaction protocols and real-time contracts.

Real-time systems have some time constraints on the services such that response time, and resource constraints such that memory requirement, bandwidth and and power consumption. Using the Unifying Theories of Programming to specify services as designs makes it easy to be extended to the so-called "timed design" which can specify also the resources requirement and the worst case execution time for a service as a relation together with pre-and post condition for the functional part of the service [4, 5]. Namely, the precondition for a method is a predicate on program variables as well as on resource variables, and the post-condition for a method is a relation on the program variables and their dashed version, and the worst case execution time dur and the resource variables. Compared with the automata approach [12], our timed designs are more denotational.

Interaction protocols for real-time components have to carry not only the information on the temporal order of the communication actions but also their timing constraints. We can label an occurrence of a communication action with the time it occurs. A sequence of communication actions labelled in that way is called a timed word. A set of a timed sequence of communication actions can express the assumption made by a component about the real-time behaviour of its environment. Timed languages are well studied and understood [1, 2], and hence using them for the specification of real-timed processes is convenient.

For the efficiency of the verification of a real-time method of components, the protocols in contracts should not have any timing information, and the timing information for a protocol is then can be derived from the specification of real-time methods from the protocol.

Real-time contracts are defined in the same way as for contracts in the previous section. The definition of interface is extended by allowing the declaration of resources and resource constraints. In real-time contracts timed designs are used instead of designs.

6 Conclusion

We have presented a component model and proposed an architecture for component-based systems. Our model can be extended easily to handle non-functional properties of component such as the quality of services. The main feature of our model lies on its formal semantic in the unifying theory of programming which can play as the basis for the development of a template language for component-based programming.

Our future work is to develop language notations based on this basis, techniques for the verification of system properties. We then combine our language notations with different programming languages to make them supporting language for component-based programming.

References

[1] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, pages 183–235, 1994.

[2] E. Asarin, P. Caspi, and O. Maler. A Kleene Theorem for Timed Automata. LICS'97, pages 160–171. IEEE computer Society Press, 1997.

[3] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.

[4] Dang Van Hung. Toward a formal model for component interfaces for real-time systems. In *FMICS '05,*pages 106–114, New York, 2005. ACM Press.

[5] Hung Ledang Dang Van Hung. Timing and Concurrency Specification in Component-based Real-Time Embedded Systems Development. Proceedings of TASE'07, June 6-8, 2007, Shanghai, China.

[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] Pham Ngoc Khoi, Ngo Trung Viet, Doan Van Ban and Dang Van Hung (editors). *Structured Programming, Bernard Robinet's lecture in Hanoi*. Nha Xuat Ban Cuc May Tinh, 1978.

[8] J. Rumbaugh and I. Jacobson and G. Booch, *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.

[9] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212-232. June 2005.

[10] J. He and X. Li and Z. Liu, A refinement calculus for object systems, *Theoretical Computer Science*, Volume 365, Issues 1-2, 10 November 2006, Pages 109-142 .

[11] Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall, 2nd edition, 1997.

[12] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. Proceedings of *SEFM'06*, pages 3–12, September 2006. IEEE Computer Society.

[13] He Jifeng, Zhiming Liu, and Xiaoshan Li. A Theories of Contracts. *Electronic Notes of Theoretical Computer Science*, Volume 160 , pp. 173-195, 2006.