 Electronic Notes in Theoretical Computer Science 176 (2007) 89–103 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

A Contract-based Approach to Specifying and Verifying Safety Critical Systems

# Wei Dong, Zhenbang Chen and Ji Wang

*National Laboratory for Parallel and Distributed Processing ChangSha, P.R.China*

{dong.wei, z.b.chen, jiwang}@mail.edu.cn

Abstract

Light-weight formal method has been regarded as an important approach to development of component- based safety critical systems. The paper proposes an approach which can formally specify and verify the contract of static structure, dynamic behavior and refinement of component systems based on UML 2.0 superstructure. As results, the correctness of static contract can be obtained via type checking of interfaces and connectors. Dynamic contract can be verified through determining the cooperativeness of integrated components, whose contracts are depicted with interface protocol state machines and their semantics models, namely contract automata. The refinement relation between high level component and its implementation will be guaranteed through defining the alternating simulation between contract automata of components at different levels.

*Keywords:* formal specification, software verification, component-based software development

# Introduction

It is an admitted fact that software has become the pivotal element in safety critical systems, such as avionics and aerospace systems, nuclear power controller, etc. Generally, these systems have the characteristics of real-time, dynamic, autonomy, fault-tolerant, and should satisfy some critical properties. Because of the complexity and demand of high confidence, the development, deployment and running of these systems are faced with grand challenges.

Component-based software development (CBSD) can effectively deal with the complexity of software construction, and provides the better manner for describ- ing architectures which present systematic frameworks of applications in specific domains. The paradigm of CBSD has also been recognized in developing safety

*٨* Supported by National Natural Science Foundation of China Grants No.60303013 and 60233020, 863 Hi-Tech Program of China Grant No.2005AA113130, National Basic Research Program of China Grant No.2005CB321802, and Program for New Century Excellent Talents in University Grant No.NCET-04- 0996.

1571-0661 © 2007 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2006.02.033

critical systems[[1](#_bookmark1)], in which it is convinced that the components with dependable guarantee will be the basis, and the corresponding methodology for component in- tegration, analysis and verification should be studied. Formal method makes it possible to calculate whether a certain description of a system is internally consis- tent,or whether requirements have been satisfied in the derivation of a design. For- mal method has been widely accepted in developing safety critical systems. Under CBSD, formal method calls for the appropriate techniques of formal specification, integration, analysis and verification around safety and reliability properties.

This paper studies the formal specification and verification of both static and dynamic contract in developing component-based safety critical systems. Its back- ground is the development of SAFE-II, the lifesaving system of manned spaceship, which uses CBSD as the paradigm. It is expected that the resulting formalism will be suitable for formal analysis and verification of SAFE-II, and the other similar systems as well. After we have investigated the characteristics and development demands of these systems, several principles will be followed in our study.

1. For the sake of practicability in component-based design and construction, a popular opinion is that the light-weight formal method should be considered under mainstream methodology and modelling language.
2. To verify essential properties of safety critical systems (e.g. via model checking), the dynamic or temporal aspects of interfaces should be specified in the contract. Moreover, the dynamic contract should support the component composition and refinement, which are common activities in CBSD.
3. The architecture pattern should be considered to improve the effectiveness of formal specification and verification. For example, in embedded safety critical systems, the interface usages of software components are usually periodical and time-triggered[[2](#_bookmark2)].
4. The formalism should be able to describe the compatibility between component system and the environment, or distinguish which are legal environments for the system.

Unified Modeling Language(UML) provides various viewpoints and diagrams to depict the characteristics of software systems. Comparing with former versions, UML 2.0 has distinctly improved the descriptive capability of component models[[3](#_bookmark3)]. This paper firstly formalizes the component model of UML 2.0, which will be used to model the static structures of safety critical systems. To depict the tempo- ral constraints of interface usages, the dynamic light-weight formal specifications are attached to the components through defining interface protocol state machines (IPSM) and their semantics models, namely contract automata. The notions of stateful and stateless are introduced into contracts to distinguish the specialties of services.

Time-triggered mode has been widely adopted in safety critical community be- cause it is somewhat predictable, and can reduce the complexity and improve the reliability and safety[[2](#_bookmark2),[4](#_bookmark4)]. SAFE-II also can be regarded as the time-triggered sys- tem. Therefore, the time-triggered pattern is brought into the component com-

position (But in current formalization, the real-time properties are not considered yet). Then the essential static and dynamic consistency rules for component inte- gration are studied respectively. By the rules, the static contract can be verified via type checking of interfaces and connectors, and the dynamic contract can be verified by investigating if the integrated components are cooperative. Based on dynamic contract, the way of how to specify and determine the legal environments is presented. The refinement relation between high level component and its imple- mentation is also studied, and the refinement consistency can be checked for both top-down design and bottom-up construction.

The method has been practically applied in SAFE-II, and it is also applicable for other safety critical systems, such as aircraft autopilot or train control systems[[5](#_bookmark5)]. These formal specifications and contracts are also the foundation of component- based system verification in our future work, such as model checking, compositional reasoning, and real-time architecture development.

The next section discusses the related work. The formal specification of compo- nent system structure is given is section 3, and the static consistencies are proposed in section 4. The dynamic contract and consistency are studied in section 5, and the refinement relation among components is studied in section 6. The last section concludes the paper with the future work.

# Related Work

There has been some work related to applying formal methods for developing soft- ware architectures and component-based systems. Wright[[6](#_bookmark6)] and Darwin[[7](#_bookmark7)] are typical ones of early architecture description languages. Wright defines the possible interactions between a set of roles, whose behaviors are specified in CSP. Dar- win describes software architecture with *π*-calculus. In SOFA[[8](#_bookmark8)], an application is viewed as a hierarchy of nested software components which can be deployed over a network. The behavior of a component in SOFA is approximated by a regular language which can be expressed by a behavior protocol, whose conformance is also defined. Archware[[9](#_bookmark9)] provides a style-based executable language which is a framework for formalizing architectures based on components and connectors. It can be used to describe architectural structure, behavior, qualities and evolution of systems. [[10](#_bookmark10)] presents a method for compositional verification of middleware-based software architecture. It is a framework not related to any specific specification of component system. In formalization of dynamic contract, interface automata[[11](#_bookmark11)] are similar to our approach. But interface automata don’t consider the structure of component system as well as concurrency and hierarchy of dynamic contract, and use the optimistic approach in composition. Without consideration of stateless interfaces, an interface explicitly only can be used by at most one component in composition of interface automata.

Some other work is more closely related to specification and verification of component-based embedded or safety critical systems. [[12](#_bookmark12)] applies concurrency con- troller design pattern in Tactical Separation Assisted Flight Environment(TSAFE),

and separates behavior verification from interface verification. Cadena[[13](#_bookmark13)] is an integrated environment for building and modelling systems using CORBA Compo- nent Model(CCM). The model can be translated into the input language of DSpin for model checking. [[14](#_bookmark14)] also studied the model checking of component composition, in which the behavior of each component is represented with Moore state model. SaveCCM[[15](#_bookmark15)] is a component model for safety critical real-time systems, and it is a part of component technology which is intended to provide efficient development, predictable behavior, and run-time efficiency. But SaveCCM has not been formal- ized. [[16](#_bookmark16)] and [[17](#_bookmark17)] give two different approaches in defining behavioral models of components and composition. The former one considers the hierarchies, which are not included in the latter. None of them specify the refinement relation.

All the above work can not fully meet the 4 principles proposed in section 1, and may not suit UML 2.0 well. Approach presented in this paper attempts to form a light-weight specification and verification method for both composition and refinement of UML 2.0 component model.

# Component Model Based on UML 2.0

In component model of UML 2.0, provided or required interfaces are offered through ports, and an interface can contain more than one operations. Components are integrated through assembly and delegation connectors[[3](#_bookmark3)]. For example, the fault- detecting component in SAFE-II is presented in Fig.1.

Definition 3.1 (Interface) An interface *u* is defined as the tuple (*O, type*), in which *O* is the operation set {*op*1*, op*2*, ...*}, and *type* can be *provided* or *required*. For clarity, *u*! or *u*? can be used to denote that an interface *u* is a provided or required interface.

Let *u*! = *u*?, *u*? = *u*!. For simplification in the following, an operation *op* of provided(or required) interface can also be called provided(or required) operation and written as *op*!(or *op*?).

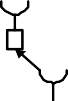
Definition 3.2 (Component) A component *C* is a 6-tuple (*IP , IR, P, R, fC, G*), in which *IP* and *IR* are provided and required interface set respectively. *P* is the set of ports. *R* ⊆ *P* × (*IP* ∪ *IR*) maps the ports to interfaces. *fC* : ∪*u*∈*IP u.O* →

{*true, false*} is a boolean function. *G* is the sub-component diagram of *C*.

An operation *op*! is stateful if *fC*(*op*!) = *true*, otherwise it is a stateless operation. *fC* will be used in dynamic contract. The component is a basic component if *G* (see definition 3*.*5) is empty, otherwise a composite component. The set *IP* and *IR* form the external view of the component, and *G* describes the internal view. Assembly and delegation connectors will be used to construct the component diagram and map one component’s external view to internal view. Port can be regarded as the bridge in delegation.

Definition 3.3 (Assembly Connector) An assembly connector is denoted as (*u*!*, v*?), which connects a provided interface to a required interface.

**SendI**



**GetArgs**

**SendParas**

**StateIds**

|  |  |
| --- | --- |
| **nstr** | **P3 P2 Fault\_Detection AO1 AI2**  **EI Escape\_ EO**  **Arg\_Reader**  **AI1 AO2 Para\_Sender**  **P1 AO3 P4**  **FI2 P5**  **Fault\_ RI Result\_**  **Adjudicator Verifier**  **FI1 FO RO** |
| **nstr** |
|  |

**GetI**



Fig. 1. Fault-detecting component in SAFE-II

Definition 3.4 (Delegation Connector)

* 1. A provided delegation connector (*p, u*!) delegates the external provided inter- face linked with port *p* to provided interface *u*! of internal view.
  2. A required delegation connector (*u*?*, p*) delegates the service required by in- terface *u*? to external required interface linked with port *p*.

For example, (*AO*2!*, EI*?), (*P* 4*, EO*!) and (*AI*1?*,P* 1) in Fig.1 are assembly, provided delegation and required delegation connectors respectively. A component diagram describes how the components are connected and form the composite com- ponent or system.

Definition 3.5 (Component Diagram) A component diagram *G* is a 3-tuple (*V, F, E*). *V* is the component set. *F* is a set of at most one component, which is the parent of components in *V* . *E* = (*NA, NDp , NDr* ). *NA* is the set of assembly connectors, and for ∀(*u*!*, v*?) ∈ *NA*, ∃*C*1*, C*2 ∈ *V* , *u*! ∈ *C*1*.IP* , *v*? ∈ *C*2*.IR*. *NDp* and *NDr* are provided and required delegation connector set respectively:

1. If *F* /= ∅, i.e. G is the sub-component diagram of the only component *C* = (*IP , IR, P, R, fC, G*) ∈ *F* :
   1. For ∀(*p, u*!) ∈ *NDp* , ∃*C*1 ∈ *V* , *u*! ∈ *C*1*.IP* , and *p* ∈ *C.P* .
   2. For ∀(*u*?*, p*) ∈ *NDr* , ∃*C*1 ∈ *V* , *u*? ∈ *C*1*.IR*, and *p* ∈ *C.P* .
2. If *F* = ∅, i.e. *G* is a topmost component diagram which does not belong to any component, *NDp* = ∅∧ *NDr* = ∅.

In a component diagram *G*, components in the same hierarchy are connected through assembly connectors. Some interfaces of these components can be con- nected with the ports of the component which regards *G* as sub-component diagram. In Fig.1, the elements inside *F ault Detection* form its sub-component diagram. Ev- ery component in diagram can has its own sub-component diagram, thus hierarchy model will be obtained.



**Prov1**

**Prov2 p**

**Prov3**

**Comp1**

**Comp3**

**Comp2**



**Req1**

**Req2**

**p Req3**

**Comp1**

**Comp3**

**Comp2**

Fig. 2. Consistencies for delegation connectors

# Static Consistencies

In component integration, the connected interfaces should be compatible to ensure correct mapping between requests and services. These are static constrains which can be verified via type checking. Some static consistencies that assembly and delegation connectors should satisfy are defined in the following.

Consistency 1 *An assembly connector* (*u*!*, v*?) *is consistent if v.O* ⊆ *u.O.*

This rule requires that for an assembly connector, all operations needed in re- quired interface should be served by the provided interface.

Consistency 2 *For a component C* = (*IP , IR, P, R, fC, G*) *and a port p* ∈ *P, if R*(*p*) ∈ *IP and G is not empty, let G* = (*V, F, E*)*, then for all* (*p, ui*!) ∈ *E.NDp , R*(*p*)*.O* ⊆ ∪*ui.O.*

For example, in the left of Fig.2, interfaces *Prov*2! and *Prov*3! must implement all the services that can be provided by *Prov*1!.

Consistency 3 *For a component C* = (*IP , IR, P, R, fC, G*) *and a port p* ∈ *P, if*

*R*(*p*) ∈ *IR and G is not empty, let G* = (*V, F, E*)*, then for all* (*ui*?*, p*) ∈ *E.NDr ,*

∪*ui.O* ⊆ *R*(*p*)*.O.*

For example, in the right of Fig.2, the requirements defined in *Req*1? must contain all the required operations of *Req*2? and *Req*3?.

# Dynamic Contract

Safety critical systems have rigorous requirements in temporal order of interface op- eration usage. Thus, it is necessary to define the dynamic contracts for components. But it is not for all interfaces that should have temporal constraints. Some provided operations can offer services at any time, not only in specialized states. Therefore, provided operations will be classified into stateful and stateless operations.

We define the interface protocol state machine (IPSM), which is composed of protocol states and protocol transitions, to depict dynamic usage of stateful oper- ations. Protocol state represents an observable stable situation, and the label of protocol transition is an operation of provided or required interface.

Definition 5.1 (Interface Protocol State Machine) IPSM is a 7-tuple (*S, L, T, H, K, I, F* ), in which *S* is the state set, set *L* contains the labels which are inter- face operations, *T* ⊆ *S* × *L* × *S* is the transition set. *H* : *S* → {*AN D, OR, BASIC*}

Fig. 3. Interface Protocol State Machine of component *Arg Reader*



**ReadDigitalSignal**

**DS\_Ready**

**1 AI2?.DigitalSignal**

**op1?**

**DS\_Obtained**

**4**

**AO2!.EscapePara**

**op4!**

**EscapeParaSend**

**7**

**2 AI2?.AnalogSignal**

**op2?**

**8**

**AS\_Obtained**

**5**

**SensorParaSend**

**ReadInstruction**

**AO1!.ControlInstr op6!**

**I\_Ready**

**3 AI1?.Instruction**

**op3?**

**I\_Obtained**

**6**

**ControlInstrSend**

**9**

**Reading**

**AO3!.SensorPara op5!**

**ReadAnalogSignal**

**AS\_Ready**

identifies the types of states. *K* : *S* → *S* ∪ {*ROOT* } returns the parent state of each state in *S*, and the parent of topmost states in IPSM is *ROOT* . *I* and *F* are initial and final state set respectively.

The IPSM of a component *A* is denoted as *MA*. The *AN D* and *OR* states of IPSM are similar to that in UML Statecharts[[3](#_bookmark3)] which model the concurrency and hierarchy structures. But the transitions in IPSM are not event-triggered. Any protocol transition labelled by a required operation means that if the component locates in the source state, it will require the service from other components. A transition labelled by a provided operation means there should exist another com- ponent requiring the operation. The IPSM of sub-component *Arg Reader* in Fig.1 is depicted in Fig.3. For clarity in the following discussion, each basic state and operation is assigned a distinct number or abbreviation(e.g., *DS Ready* is assigned 1, *AI*2?*.DigitalSignal* is abbreviated as *op*1?).

For an operation *op*! of component *A*, *fA*(*op*!) = *true* if it appears in some transition label of *MA*, else *fA*(*op*!) = *f alse*. It is assumed that a stateful operation can not be used as the stateless operation simultaneously. If a provided operation can be used by more than one components, it must be stateless. Two complemented operations may become an internal operation after the component composition.

Definition 5.2 (Configuration) A configuration of IPSM is a set *Conf* ⊆ *S* that satisfies: (1) ∃1*s* ∈ *Conf* that *K*(*s*)= *ROOT* ; (2) For ∀*s* ∈ *Conf* , if *H*(*s*)= *AN D*, for ∀*s*' ∈ *S* that *K*(*s*') = *s*, *s*' ∈ *Conf* ; (3) For ∀*s* ∈ *Conf* , if *H*(*s*) = *OR*,

∃1*s*' ∈ *Conf* that *K*(*s*')= *s*.

An IPSM must locate in a configuration at any time. Interleaving semantics is considered for *AN D* states according to the characteristic of SAFE-II, and the se- mantics of entering and exiting composite states is the same as in UML Statecharts. The transition model generated from IPSM *MA* is named contract automaton, de- noted as Θ*A*.

Definition 5.3 (Contract Automaton) A contract automaton Θ*A* generated from an IPSM *MA* is a 6-tuple (*SA, sI , OP , OR, ON , TA*):

*A A A A*

- *SA* is the state set (i.e. the configurations of IPSM *MA*).

**a**

**1,2,3**

**k**

**9**

**op1?**

**op2? op3?**

**b 4,2,3**

**op2?**

**e 4,5,3**

**c 1,5,3**

**op6!**

**op3?**

**op1?**

**op3?**

**op1?**

**d 1,2,6**

**op2?**

**g 1,5,6**

**j 8**

**f 4,2,6**

**op2?**

**op5!**

**op3?**

**op1?**

**h 4,5,6**

**op4!**

**i 7**

* *sI*

Fig. 4. Contract automaton Θ*AR* obtained from the IPSM of *Arg Reader*

∈ *SA* is the initial state.

*A*

* *OP* , *OR* and *ON* are set of provided, required and internal operations respectively.

*A A A*

Let *OA* = *OP* ∪ *OR* ∪ *ON* .

*A A A*

* *TA* ⊆ *SA* × *OA* × *SA* is the transition set.

The contract automaton generated from IPSM of component *Arg Reader* is presented in Fig.4 (For clarity, only basic states are listed in the configurations). Let

*OP* (*v*), *OR*(*v*) and *ON* (*v*) be the set of provided, required and internal operations

*A A A*

labelled in the transitions from *v* respectively, and *OA*(*v*)= *OP* (*v*) ∪ *OR*(*v*) ∪ *ON* (*v*).

*A A A*

The behavior of composite component can be obtained through the composition

of the contract automata of sub-components. SAFE-II can be regarded as the system with time-triggered pattern[[2](#_bookmark2)], for example, the component *F ault Detection* will periodically execute. In the beginning of each cycle, the contract automaton of each sub-component will locate in its initial state. During the execution, the components synchronize through interface invocation. But they should return back to the initial states at the beginning of next cycle. Therefore, in the semantics of composition, both of the synchronization of interface interaction and time-triggered pattern should be included.

Two contract automata Θ*A* and Θ*B* are composable if *OP* ∩*OP* = ∅, which means

*A B*

that the services provided by two components have no overlap. *share*(*A, B*) =

(*OP* ∩*OR*)∪(*OP* ∩*OR*) includes those operations that are required in one component

*A B B A*

and provided by the other one.

Definition 5.4 (Composition) For two composable contract automata Θ*A* and Θ*B*, their composition Θ*C* = Θ*A* ⊗ Θ*B* is also a contract automaton:

* *SC* ⊆ *SA* × *SB*
* *sI* = (*sI , sI* )

*C A B*

* *OP* = (*OP* ∪ *OP* ) \ {*o*|*o* ∈ *share*(*A, B*)*, fA*(*o*!) ∨ *fB*(*o*!)}

*C A B*

* *OR* = (*OR* ∪ *OR*) \ *share*(*A, B*)

*C A B*

* *ON* = *ON* ∪ *ON* ∪ *share*(*A, B*)

*C A B*

* *fC* = (*fA* ∪ *fB* )|(*DOM* (*fA*)∪*DOM* (*fB* ))\*share*(*A,B*)



**RecieveSignal**

**P\_Ready**

**1 FI2?.SensorPara**

**op5?**

**P\_Obtained**

**3**

**ReadInstruction**

**I\_Ready 2 FI1?.Instruction**

**op3?**

**I\_Obtained**

**4**

**FaultFlagSend**

**6**

**FO!.FaultFlag**

**op7!**

**Adjudging**

**5**

**Incept**

**l 1,2**

**q 6**

**op5?**

**op3?**

**m 3,2**

**n 1,4**

**op3? op5?**

**op7!**

**o**

**3,4**

**p**

**5**

Fig. 5. IPSM and contract automaton Θ*FA* of component *F ault Adjudicator*

- *TC* = {((*v, u*)*, o,* (*v*'*, u*))|(*v, o, v*') ∈ *TA* ∧ *o* ∈*/ share*(*A, B*) ∧ *u* ∈ *SB*}

∪{((*v, u*)*, o,* (*v, u*'))|(*u, o, u*') ∈ *TB* ∧ *o* ∈*/ share*(*A, B*) ∧ *v* ∈ *SA*}

∪{((*v, u*)*, o,* (*v*'*, u*'))|(*v, o, v*') ∈ *TA* ∧ (*u, o, u*') ∈ *TB*

∧(*o* ∈ *share*(*A, B*) ∨ ¬*fC*(*o*!))}

- For any path (*v*0*, u*0)*,* (*v*1*, u*1)*, ...* in Θ*C*, (*vi, ui*) is the first state that *vi* /= *sI*

*A*

and

*ui* /= *sI* , if *vj* (*uj*) is the first state after *vi* (*ui*) in sequence *v*0*, v*1*, ...* (*u*0*, u*1*, ...*)

*B*

that *vj* = *sI* (*uj* = *sI* ), *vj* (*uj*) can not be left before the sequence *u*0*, u*1*, ...*

*A*

*B*

(*v*0*, v*1*, ...*) returns to *sI* (*sI* ) after *ui* (*vi*).

*B*

*A*

The computing of *TC* illuminates that the executions of two contract automata are interleaved if the operations labelled on transitions are not included in *share*(*A, B*) or are stateless, and should synchronize if the transitions labelled with stateful op- erations in *share*(*A, B*) are met. The last item in above definition requires that both of the two contract automata of time-triggered periodic components should return to the initial states before the start of next cycle. The stateless operations can still be used by other components after the composition of *A* and *B*.

Fig.5 gives the IPSM and contract automaton Θ*FA* of *F ault Adjudicator*, which also is the sub-component of *F ault Detection* and will be integrated with *Arg Reader* through the composition of Θ*AR* and Θ*FA*. For simplification, we only consider two paths *δ*1 : *a, b, e, h, i, j, k, a, ...* in Θ*AR* and the path *δ*2 : *l, n, o, p, q, l, ...* in Θ*FA*. From the IPSM models, operation *op*5 is included in *share*(*AR, FA*), because *op*5? in *F ault Adjudicator* will be served by *op*5! in *Arg Reader*. In the composition, the prefix before *i* in *δ*1 and prefix *l, n* in *δ*2 can be interleaved. Then the transition (*i, j*) in *δ*1 and (*n, o*) in *δ*2 should be synchronized. When the composition automa- ton reaches to state (*j, o*), *δ*1 and *δ*2 can interleave again. But when either of two path returns to the initial state, supposing that *δ*1 enters *a*, it should wait for the other path returns to the initial state, in this case *δ*2 enters *l*.

The synchronization in initial states comes from the requirement in practical implementation, and also can reduce the number of transitions, even states, of the composed system, which is beneficial to formal analysis and verification. It can be proved that the composition is commutative and associative.

Theorem 5.5 *If contract automata* Θ*A,* Θ*B and* Θ*C are pairwise composable,*



**v**

**u**

**o1**

**o2**

**v'**

**u'**



**v**

**v'**

**u**

**o**

**…**

**u'**

**(a) (b)**

Fig. 6. Deadlock and pseudo-deadlock states

(Θ*A* ⊗ Θ*B*) ⊗ Θ*C* = Θ*A* ⊗ (Θ*B* ⊗ Θ*C*)*.*

For composable contract automata Θ*A* and Θ*B*, there may exist some states in Θ*A* ⊗ Θ*B* from which the automaton can not find the next step to execute. Some of these states exist because of the deadlock, and will be put into the set *dl*(*A, B*). The other of these states exist because of the synchronization requirement for initial states between two automata in time-triggered pattern. It can be regarded as the pseudo-deadlock, and the states will be put into the set *pdl*(*A, B*).

*dl*(*A, B*)= {(*v, u*) ∈ *SA* × *SB*|(*OA*(*v*) ⊆ *share*(*A, B*)) ∧ (*OB*(*u*) ⊆ *share*(*A, B*))

∧(∀*o* ∈ *OA*(*v*)*,* ∀*o*' ∈ *OB*(*u*)*, o* /= *o*')

∧(∀*o*? ∈ *OR*(*v*)*.fB*(*o*!)) ∧ (∀*o*'? ∈ *OR*(*u*)*.fA*(*o*'!))}

*A*

*pdl*(*A, B*)= {(*v, u*) ∈ *SA* × *SB*|

((*OA*(*v*) ⊆ *share*(*A, B*)) ∧ *u* = *sI*

*B*

*B*

∧ (∀*o* ∈ *OR*(*v*)*.fB*(*o*!)))

*A*

∨((*OB*(*u*) ⊆ *share*(*A, B*)) ∧ *v* = *sI* ∧ (∀*o* ∈ *OR*(*u*)*.fA*(*o*!)))}

*A*

*B*

For example, in Fig.6 (a), when *o*1*, o*2 ∈ *share*(*A, B*), *o*1 /= *o*2, and actually *o*1 is *o*1!, or it is *o*1? but *fB*(*o*1!) = *true*, and *o*2 is *o*2!, or it is *o*2? but *fA*(*o*2!) = *true*, then the deadlock arises and state (*v, u*) will be included in *dl*(*A, B*). In Fig.6 (b), when *o* ∈ *share*(*A, B*), and if *o* is actually *o*!, or it is *o*? but *fA*(*o*!) = *true*, then the state (*v, u*) will be included in *pdl*(*A, B*). The reason is that after Θ*A* and Θ*B* running from the initial states in some cycle, when Θ*A* again returns to initial state *v*, it should wait for the moment that Θ*B* returns to its initial state. But when Θ*B* locates in *u*, it will wait for the synchronization through operation *o*. Then a pseudo-deadlock appears, which is actually a design error for time-triggered pattern.

Definition 5.6 (Cooperative) Two contract automata Θ*A* and Θ*B* are coopera- tive if:

1. Θ*A* and Θ*B* are composable;
2. None of states in *dl*(*A, B*) is reachable;
3. For any path (*v*0*, u*0)*,* (*v*1*, u*1)*, ...* in Θ*A* ⊗ Θ*B*, (*vi, ui*) is the first state that

*vi* /= *sI* and *ui* /= *sI* , if (*vj, uj*) is the first state that *vj* = *sI* and *uj* = *sI*

*A B A B*

after (*vi, ui*), then none of states in *pdl*(*A, B*) appears in the sequence segment

(*vi, ui*)*, ...,* (*vj*−1*, uj*−1).

When one component is integrated with the others, it is expected that the states

**op5!**

**l,x**

**q,x**

**op3**

**op7!**

**n,y**

**op5**

**o,x**

**p,x**

**x**

**op3!**

**y**

**(a) (b)**

Fig. 7. One legal environment of Θ*FA* and their composition

in *dl* are unreachable at any time, and the states in *pdl* are unreachable after each component has left initial state in a cycle.

Consistency 4 *A component diagram* (*V, F, E*) *is consistent iff the contract au- tomata of any two components in V are cooperative.*

In practice, it is unnecessary to check each pair of components in verifying this rule. The verification can be combined with the process of composition. Because

⊗ is commutative and associative, it can be proved that if the above consistency is not satisfied, then in any composition sequence Θ*An* ⊗ (Θ*An*−1 ⊗ (*...* ⊗ (Θ*A*2 ⊗ Θ*A*1 )*...*) selected to compute ⊗*V* (denotes the composition of contract automata of all components in *V* ), there will exist some *i* that (Θ*Ai* ⊗ (Θ*Ai*−1 *...* ⊗ (Θ*A*2 ⊗ Θ*A*1 )*...*) is not cooperative with Θ*Ai*+1 ; on the other side, if the consistency is satisfied, the case of uncooperative will not exist in any composition sequence. Therefore, we can choose any one composition sequence to compute ⊗*V* , and if an uncooperative case is faced, the consistency is not met, otherwise it is satisfied.

The above consistency can only ensure the correctness among components within the system, but there still may be some required interfaces should be served by the components outside the system, i.e. the environment. To make testing, analysis and verification feasible, the way to specify legal environment with which the component system can work well should be provided.

Definition 5.7 (Legal Environment) A Legal environment of a contract automa- ton Θ*A* is a nonempty contract automaton Θ*E* such that:

1. *OR* = *OP* ;

*A E*

1. Θ*A* and Θ*E* are cooperative.

Fig.7 (a) presents an environment of contract automaton Θ*FA*, and their com- position is given in Fig.7 (b), from which it can be concluded that this environment is legal for Θ*FA*.

For a close system, none external service will be needed, only the consistency 4 should be satisfied. When an open system has been constructed, it should be considered which are legal environments of the system, and whether the system can correctly work in some given environment. It is obvious that an open system can only work well in its legal environments.

# Component Refinement

In CBSD, a high level component may be refined into an implementation compo- nent, or be constructed through integrating more than one sub-components, which can also be regarded as its implementation. The high level component and its im- plementation must satisfy not only the static consistencies of delegation, but also the consistency between their dynamic behaviors, e.g. IPSMs or contract automata. Hence there should exist a refinement relation between a component and its imple- mentation.

For two components *A* and *B*, if *B* refines *A*, it is said *A* is the specification and *B* is the implementation. Refinement is traditionally defined as trace containment or simulation. However, it is not appropriate in component systems, and refinement defined through alternating simulation has been proposed[[18](#_bookmark18),[11](#_bookmark11)]. In practice, if *B* implements *A*, *B* must be able to provide at least all the services defined in *A*. Otherwise, the other components can not find the services they need according to the specification of *A*. On the other hand, *B* should not require more services provided by other components than that required in *A*, otherwise it perhaps can not work correctly because there may be no component provides such services according to the requirement of *A*. Then it can be concluded that if *B* refines *A*, *B* can have more provided interfaces and less required interfaces than *A*. Moreover, the temporal of dynamic behaviors of *B* should be consistent with that of *A*. In one word, *B* must be able to work correctly in the environments in which *A* can work well.

If *A* is implemented by more than one components *Bi*, *B* is regarded as the composition of these *Bi*, then *A* and *B* should also satisfy the above claim. But for each sub-component, it may has its own interface operations which interact only with other sub-components in the same level. These operations will become the internal operations after integration, and will not affect the refinement relation between *B* and *A*. We will study the refinement based on contract automata.

The internal transitions can not be seen by the environment, which leads to the two states connected by an internal transition can not be distinguished. For the simulation focuses on provided and required operations here, a state *s* and the states reached only by internal transitions from *s* can be merged.

Definition 6.1 Given a contract automaton Θ*A* and a state *s* ∈ *SA*, the set Γ*A*(*s*) ⊆ *SA* is the smallest set such that 1) *s* ∈ Γ*A*(*s*), 2) if *v* ∈ Γ*A*(*s*) and

∃*o* ∈ *ON .*(*v, o, u*) ∈ *TA*, then *u* ∈ Γ*A*(*s*).

*A*

Each required operation that may execute in the transition from some state in Γ*A*(*s*) should be served by the environment, because any of these transitions may be issued without forewarning. But the environment can only require the service that can execute in some transition from each state in Γ*A*(*s*), because it can not distinguish in which state the component is locating. Therefore, some notations will be presented according these facts. Given a contract automaton Θ*A* and a state *s* ∈ *SA*,

*EXR*(*s*)= {*o*|∃*v* ∈ Γ*A*(*s*)*.o* ∈ *OR*(*v*)}

*A A*

*EXP* (*s*)= {*o*|∀*v* ∈ Γ*A*(*s*)*.o* ∈ *OP* (*v*)}

*A A*

For any *o* ∈ *EXP* (*s*) ∪ *EXR*(*s*), let

*A A*

*DestA*(*s, o*)= {*u*|∃(*v, o, u*) ∈ *TA. v* ∈ Γ*A*(*s*)}

Then the alternating simulation between contract automata can be defined.

Definition 6.2 Given two contract automata Θ*A* and Θ*B*, a binary relation ≥⊆ *SA* × *SB* is an alternating simulation if for any two states *v* ∈ *SA* and *u* ∈ *SB* that *v* ≥ *u*, the following conditions are satisfied:

1. *EXP* (*v*) ⊆ *EXP* (*u*), *EXR*(*u*) ⊆ *EXR*(*v*).

*A B B A*

1. For each operation *o* ∈ *EXP* (*v*) ∪ *EXR*(*u*)and each state *u*' ∈ *DestB*(*u, o*),

*A B*

there is a state *v*' ∈ *DestA*(*v, o*) such that *v*' ≥ *u*'.

Two contract automata has the refinement relation if the initial state of first component is alternating simulated by the initial state of second one.

Definition 6.3 The contract automaton Θ*B* refines the contract automaton Θ*A*,

written Θ*A* ≥ Θ*B*, if 1) *OP*

*A*

*B*

*B*

*A*

⊆ *OP*

and *OR*

⊆ *OR*, 2) there is an alternating

simulation ≥ from Θ*B* to Θ*A*, such that *sI* ≥ *sI* .

*A*

*B*

It should be guaranteed that the refinement relation must be kept for composi- tion ⊗, which will ensure the specification can be safely replaced by its implemen- tation in component integration. The following theorem will ensure the refinement relation can still be hold after composition. The cooperativeness also is kept after the refinement.

Theorem 6.4 *For the contract automata* Θ*A,* Θ*B,* Θ*A*' *and* Θ*B*' *that* Θ*A* ≥ Θ' *,*

*A*

Θ*B* ≥ Θ' *:*

*B*

1. *If* Θ*A and* Θ*B,* Θ*A*' *and* Θ*B*' *are composable respectively, then* (Θ*A* ⊗ Θ*B*) ≥

(Θ*A*' ⊗ Θ*B*' )*;*

1. *If* Θ*A and* Θ*B are cooperative, then* Θ'

*A*

*and* Θ'

*are cooperative too.*

Component-based software development can be considered in two directions: top-down design and bottom-up construction. In both methods, the following re- finement consistency should be followed.

*B*

Consistency 5 *For a component A* = (*IP , IR, P, R, fA, G*) *in which G* = (*V, F, E*)*, if V* /= ∅*, then* Θ*A* ≥ ⊗*V should be satisﬁed.*

# Conclusion and Future Work

The industry community has attempted to take the advantages of CBSD paradigm in safety critical system development, in which formal method is desired to be ap- plied. This paper presents our first step to bridge the gap between formal method and the most widely used modeling language UML. The improvement of component model in UML 2.0 also helps us to realize the purpose. We study the characteristics

and pattern of component-based safety critical systems like SAFE-II, and propose the formal specification of static structure and dynamic behavior. The consistency rules for static connection, dynamic composition and component refinement are also studied, which can be regarded as one way of verification. The method and support- ing tool are being applied in the practical development of SAFE-II. The preliminary results show that they have good usability, and provide a kind of rigorous way to develop component-based safety critical systems.

Around these formal specifications, model checking of component-based safety critical systems is now being studied, especially combined with compositional rea- soning to improve the scalability. Timing constraints are unavoidable in these sys- tems (e.g. SAFE-II), and how to introduce real-time model and related performance interfaces into specification and verification will be further studied.

# References

1. Ivica Crnkovic. *Component-based approach for Embedded Systems*. Ninth International Workshop on Component-Oriented Programming, 2004.
2. Michael J. Pont. “Patterns for Time-Triggered Embedded Systems”. Addison Wesley, 2001.
3. “UML 2.0 Superstructure Specification”. Object Management Group, 2003.
4. MISRA. “Development guidelines for vehicle-based software”. Motor Industry Software Reliability Report, 1994.
5. Hermann Kopetz. *The Time-Triggered Architecture*. First International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE Computer Society, 1998.
6. Robert J. Allen, Remi Douence, and David Garlan. *Specifying Dynamism in Software Architectures*. Proceedings of the Workshop on Foundations of Component-Based Software Engineering, 1997.
7. D. Giannakopoulou, J. Kramer, and S. Chi Cheung. *Behaviour analysis of distributed systems using the tracta approach*. Automated Software Engg., 6(1):7-35, 1999.
8. F. Plasil and S. Visnovsky. *Behavior protocols for software components*. IEEE Transactions on software Engineering, 28(11):1056C1076, November 2002.
9. Ron Morrison, Graham Kirby, Dharini Balasubramaniam, Kath Mickan. *Support for Evolving Software* *Architectures in the ArchWare ADL*. Fourth Working IEEE/IFIP Conference on Software Architecture, 2004.
10. Mauro Caporuscio, Paola Inverardi, Patrizio Pelliccione. *Compositional Verification of Middleware-* *Based Software Architecture Descriptions*. 26th International Conference on Software Engineering, 2004.
11. Luca de Alfaro, Thomas A. Henzinger. *Interface Automata*. Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering, 2001.
12. Aysu Betin-Can, Tevfik Bultan, Mikael Lindvall, Benjamin Lux and Stefan Topp. *Application of Design for Verification with Concurrency Controllers to Air Traffic Control Software*. In Proc. of ASE, 2005.
13. John Hatcliff, William Deng, Matthew B. Dwyer, et.al. *Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems*. 25th International Conference on Software Engineering, 2003.
14. Fei Xie, James C. Browne. *Verified systems by composition from verified components*.The 9th European software engineering conference, 2003.
15. M. ˚Akerholm, A. Mo¨ller, H. Hansson and M. Nolin. *Towards a Dependable Component Technology for Embedded System Applications*. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. 2005.
16. T. Barros, L. Henrio, E. Madelaine. *Behavioural Models for Hierarchical Components*. Proceedings of 12th International SPIN Workshop on Model Checking of Software, LNCS 3639. Springer-Verlag, 2005.
17. C. Carrez, A. Fantechi, and E. Najm. *Behavioural contracts for a sound assembly of components*. In Springer-Verlag, editor, in proceedings of FORTE03, volume LNCS 2767, 2003.
18. R.Alur, T.Henzinger, O.Kupferman, and M.Vardi. *Alternating refinement relations*. In Concurrency Theory, LNCS 1466. Springer-Verlag, 1998.