Consistency between UCM and PSMs in component models

Truong Ninh Thuan, Tran Vu Viet Anh and Nguyen Viet Ha
College of Technology
Vietnam National University, Hanoi
144 Xuan Thuy, Cau Giay, Hanoi
Email: {thuantn,hanv}@vnu.edu.vn

***Abstract*—In software engineering, the earlier the detection of faults, the less expensive the correction of faults. This paper proposes an approach for detecting the conformability in component models. It focuses on verifying the internal specification of component interfaces with the maps described the execution of tasks between components. Firstly, protocol state machines (PSMs) associating component interfaces are expressed by B abstract machines. Then, the UCM (Use Case Maps) describing a scenario execution in component interaction is formalised by a B implementation machine. The consistency between UCM and PSMs thus can be automatically validated by B provers.**

Keywords. UCM, PSM, Component model, verification, B method.

I. Introduction

Component based software engineering (CBSE) addresses the challenge of the software development in which software programs are becoming increasingly large and complex. It proposes an easy and efficient method for developing large softwares. With CBSE, the architecture of a system is described as a collection of components. The interaction between components are performed via their interfaces. The main feature of CBSE is to allow the construction of an application using independently developed software components. This helps reducing development costs and improving software quality. In this process, it is essential to ensure that individual components can in fact interoperate together in the system. However components may not interact seamlessly. Problems could arise in the system if there are mismatches and inadequacies of connected points between components. Verifying the correctness of component composition thus is an important task in component software development. In addition, CBSE is an approach to develop software systems, hence discovering bugs in the earlier phases will reduce much time and effort in building software systems, especially large and complex systems. Many approaches have been proposed to verify the compatibility between components by interfaces [10], [11], by behavioral protocol [14], or by models [9], etc.

Developers using component technologies such as COM, COM+ [8] or EJB[1] need to be able to define and express specifications for their components. They can do so using component diagrams of the Unified Modeling Language (UML2.0) [6], [2]. However, they also need a simple process that ensures specifications relate consistently in different views.

In order to specify component models, developers usually associating protocol state machine (PSM) [2], a new diagram in UML2.0, into interfaces of components. PSM specifies which operations of the interfaces can be called in which state and under which condition, thus specifying the allowed call sequences on the interface methods. A protocol state machine presents the possible and permitted transitions on the instances of its context classifier, together with the operations that carry the transitions.

On the other hand, Use Case Maps (UCM)[3] is a visual notation using to describe requirement behaviors of a system at a high level of abstraction. It is a scenario-based approach showing cause-effects by traveling over paths through a system. UCM is also applied in component technology to model the execution of a scenario between components.

However, the UCM specification may be incompatible with PSM of component interfaces at the design phase. As a consequence, that can lead to errors in component software deployment.

Imagine that, the interaction between component interface protocols through an use case map is considered as a running of chain gears system where each interface protocol plays the role as a gear and the use case map is a chain driven. Cogweels must be operated as a right direction. In a similar way, a scenario specified by UCM is performed according the order of events defined in protocol state machines.

In this paper, we propose an approach to check the conformability between PSM and UCMs in component specifications. The verification is performed by the B method. The B notation [4] is based on set theory, generalised substitutions and first order logic, these are easily to specify state transitions of interfaces. In addition, the proof obligations for B specifications are generated automatically by support tools like AtelierB [15], B-Toolkit [13] and B4free [7]. Checking proof obligations with B support tool allows us to find inconsistencies in different views of a model specification.

This approach contributes to the verification technique in design component models by analysing the consistency between UCM and PSMs. It can be summarised as the following:

- Specifying the component interfaces in a model software by UML protocol state machines.
- Describing the interaction between components to execute a task by use case maps.

- Transforming protocol state machines into B abstract machines.
- Transforming use case maps into a B implementation machine and using B support tools to check the conformance between UCM and PSMs.

The rest of this paper is organised as follows. Section 2 presents the background of Use Case Maps and Protocol State Machines. Section 3 proposes a B model to verify the conformance between PSM and UCM. Section 4 gives a case study - the stock quoter system. In this section we also illustrate the verification of the B model proposed for the case study. Section 5 discusses related works. We conclude and give some perspectives in Section 6.

II. Background

In this section, we give an overview of protocol state machines (PSM) and use case maps (UCM). We concentrate on the context that they are used and some basic concepts.

A. Protocol state machine

When we want to show the sequence of events that an object reacts to – and the resulting behavior – we can use UML behavioral state diagrams. Such state diagrams have event/action pairs, entry actions, exit actions, and do activities. Sometimes, however, we just want to show a specified sequence of events that object responds to – and when it can respond – without having to show its behavior. Such a specified sequence is called an event protocol. In UML2.0, we can describe event protocols by protocol state machines. These differ from behavioral state machines and have special uses.

Normally we should use regular state diagrams to show internal sequences of behavior for all objects of a class. Sometimes, however, we want to show a complex protocol (set of rules governing communication) when using an interface for a class. For example, when we are designing classes that access a database for applications we need to use common operations like *open*, *close* and *query* a database. But these operations must be called in the right order. We cannot query the database before we open it.

Since an interface of a component specifies conformance characteristics, it does not own detailed behavior specifications. Instead, interfaces may own a protocol state machine that specifies event sequences and pre/post conditions for the operations and receptions described by the interface.

B. Use Case Maps

The Use Case Map (UCM) notation can help to describe and understand emergent behaviour of complex and dynamic systems. It is consider as causal scenarios, as architectural entities, or as behaviour patterns.

The basic idea of UCM is captured by the phrase causal paths cutting across organizational structures. The realization of this idea produces a lightweight notation that scales up, while at the same time covering all of the foregoing complexity factors in an integrated and manageable fashion. The notation represents causal paths as sets of wiggly lines that enable a person to visualize scenarios threading through a system without the scenarios actually being specified in any detailed way (e.g. with messages). Compositions of wiggly lines (which may be called behavior structures) represent large-scale units of emergent behavior cutting across systems, such as network transactions, as first-class architectural entities that are above the level of details and independent of them (because they can be realized in different detailed ways).

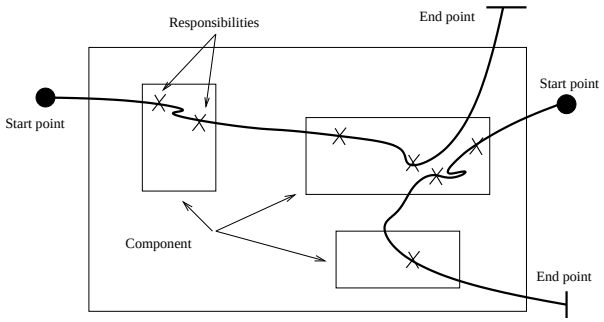

Fig. 1. Use Case Maps

UCMs do not have clearly defined semantics, their strong point is to show how things work globally. The basic UCM notation is very simple. It is comprised of three basic elements: responsibilities, paths and components. A simple UCM exemplifying the basic elements is shown in Figure 1. A path is executed as a result of the receipt of an external stimulus. Imagine that an execution pointer is now placed on the start position. Next, the pointer is moved along the path thereby entering and leaving components, and touching responsibility points. A responsibility point represents a place where the state of a system is affected or interrogated. The effect of touching a responsibility point is not defined since the concept of state is not part of UCM. Typically, the effects are described in natural language. Finally, the end position is reached and the pointer is removed from the diagram. A UCM is concurrency neutral, that is, a UCM does not prescribe the number of threads associated with a path. By the same token, nothing is said about the transfer of control or data when a pointer leaves one component and (re-)enters another one. The only thing guaranteed is the causal ordering of executing responsibility points along a path.

An UCM may have also AND-forks/joins, OR-fork/joins. The AND-construct is used to spawn multiple activities along parallel paths. When a pointer reaches an AND-fork, this pointer is removed from the diagram and replaced by two pointers at the beginning of the parallel paths. An AND-join acts as a synchronization mechanism. When both pointers have reached the AND-join, they are replaced by a single pointer and execution is continued along the path following the join. The OR-construct should be interpreted as a means to express multiple scenarios in a single diagram. It states that multiple scenarios are comprised of identical paths. Therefore, it is not necessary to specify conditions detailing the path to be

followed at an OR-fork.

III. B MODEL TO VERIFY THE CONFORMANCE BETWEEN PSMS AND UCM

The incompatibility problem between PSM and UCM in component specification represented by the execution order of methods and the states obtained after each execution in these descriptions.

Return to the example of database processing above, before querying data in database, we have to open file and after processing data, we have to close the file. The order of these behaviours is described by UML protocol state machine associating with the interface of database component. The incompatibility problem appears if we use an UCM to model the interaction between components and its scenario performs updating data but we do not open the file, or updating data is executed after closed methods, etc.

As mentioned above, we use the B method [4] to detect the incompatibility between UCM and PSMs. The general model is presented in Figure 2. In this model, the Library machine is used to define sets of states specified in each protocol state machine. Each protocol state machine of interface is defined by an abstract machine called PSM machine. The UCM describes sequence event protocol allowing us to define two machines: an abstract machine (UCM.mch) and an implementation machine (UCM_REF.imp).

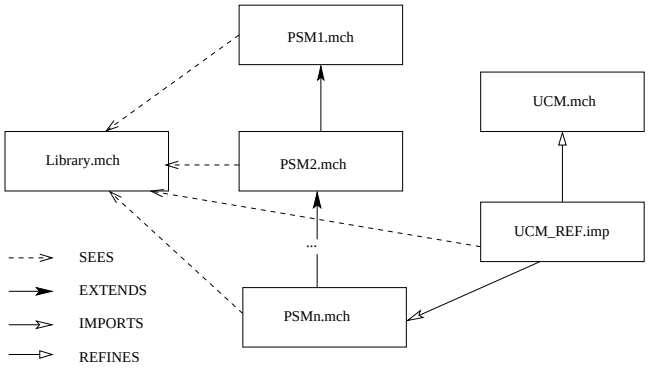

Fig. 2. B model to verify the conformance between PSM and UCM

In the implementation machine, we specify the execution of methods according to the order in UCM scenario by refining the abstract machine of UCM. Following the order execution of methods in the path of UCM, the abstract machine of PSM which contains latter methods (PSM_{i+1}) extends (EXTENDS) the one contains previous methods (PSM_i). This relation allows PSM_{i+1} machine can use data and get promotion of operations in PSM_i machine. The last abstract machine of PSM (PSM_n) is imported into the implementation machine of UCM (UCM_REF.imp). All B machines in the model see (SEES) the Library machine.

IV. A CASE STUDY - THE STOCK QUOTER SYSTEM

To illustrate the proposed approach, we use a case study of the Stock Quoter System. This system is designed by three components: Database (DB), Stock Distributor, Stock Broker. Generally, the flow of information of the stock-quoter system works as follows:

- Stock broker clients subscribe with a stock distributor server to receive notification events whenever a stock value of interest to them changes.
- The stock distributor server monitors a real-time stock feed database.
- Whenever the value of a stock changes, the distributor publishes an event to interested stock brokers.

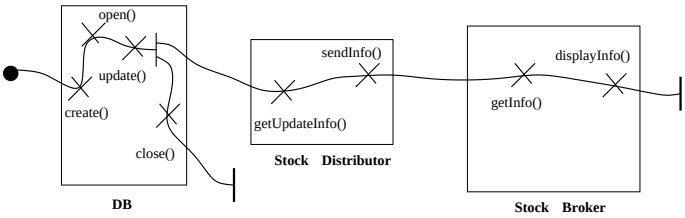

Fig. 3. An UCM in the Stock Quoter System model

Stock brokers only contact the stock distributor server via request/response operations when the value of a stock changes. Figure 3 illustrates a map execution of this system using UCM. In this scenario, we begin by the order of the execution of events in the database component. The Stock Distributor component is designed to monitor the real-time stock database and, when the values of data are changed, the Stock Distributor will notify to the Stock Broker component.

```
Interface DBAccess{
        voidcreate()
        voidopen()
        voidupdate()
        voidclose()
}
```

Fig. 4. DBaccess interface

In this system, we have to develop a DatabaseAccessor class with a DBaccess interface as shown in Figure 4 to access database component. But the DBaccess interface has a complex protocol that governs its use because of the rules governing communication between any other object and the DatabaseAccessor class implementing the DBaccess interface. To use the interface properly, we have to open the database and then perform updating. We can put these rules in a protocol state diagram to indicate the order of events that must be followed when using the interface.

Regular state diagrams are not allowed to work with interfaces because interfaces do not describe behavior implementation, they just declare what operations the class must perform. Protocol state machine's aim is to specify the implementation of a class for an interface. On the other hand, it enables us to declare what operations can happen and the order they can happen without having to say anything about behavior implementation.

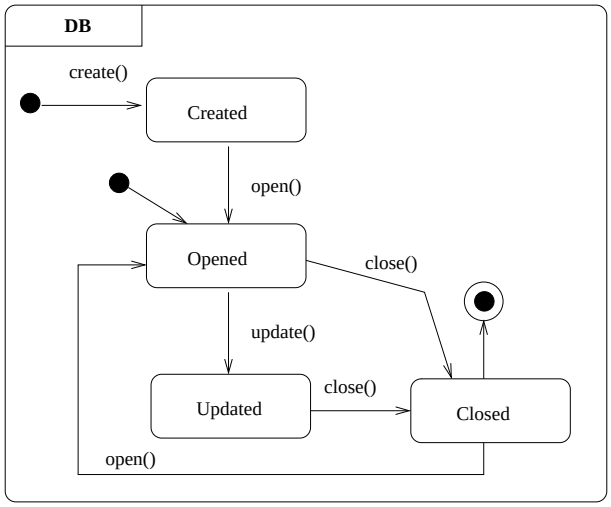

Fig. 5. An UML2.0 protocol state machine for DBaccess interface

Figure 5 shows the DBaccess interface attached to the DatabaseAccessor class; the DatabaseAccessor class must conform to the operation sequence (that is, the protocol) of the DBaccess interface: The create, open, update, close operations must be implemented in the order specified by the DBaccess interface's protocol.

A. Expressing UML protocol state diagrams by B notations

We check the conformance between UCM and PSM of the Stock Quoter System model based on the B model proposed in section III. The Library abstract machine is used to contain sets of states extracted from protocol state machines in the model (StateDB, StateSD,...); The abstract machines DBProtocol, SDProtocol, SBProtocol correspond to components DBaccess interface, Stock Distributor component interface, Stock Broker component interface, respectively.

MACHINE *Library*
SETS
$StateDB = \{Created, Opened, Updated, Closed\}$;
$StateSD = \{GET\_OK, SEND\_OK\}$;
...
END

Fig. 6. Library abstract machine

The DBProtocol abstract machine (Figure 7) describes the DBaccess interface's protocol specified in Figure 5. Based on the set of states declared in the Library machine, functions in the DBProtocol machine define possible state transitions of interface's protocol. For instance, the *update* operation is performed if the file is opened (formalised by B notations is $state = Opened$).

In a similar way, the SDProtocol abstract machine (Figure 8) describes the Stock Distributor interface's protocol which contains only two methods: *getUpdateInfo* and *sendInfo*. The method *sendInfo* must be executed after *getUpdateInfo* in the design of protocol state machine. Note that this machine extends (EXTENDS clause) the DBProtocol machine following the order execution of methods in the path of UCM. The method *getUpdateInfo* of SDProtocol component interface is executed if the *update* method of DBaccess interface performed. That means that its precondition is $state = Updated$.

MACHINE *DBProtocol*
SEES *Library*
VARIABLES *state*
INVARIANT $state \in StateDB$
INITIALISATION
    $state := Created$
OPERATIONS
create =
BEGIN
    $state := Created$
END;
open =
PRE
    $state = Created \text{ or } state = Closed$
THEN
    $state := Opened$
END;
update =
PRE
    $state = Opened$
THEN
    $state := Updated$
END;
close =
PRE
    $state = Updated \text{ or } state = Opened$
THEN
    $state := Closed$
END
END

Fig. 7. B abstract machine of DBaccess protocol

B. Verification of conformance specification in design

The verification of conformance between UCM and PSMs is performed in an implementation machine called UCMStock which describes the execution order of methods of component interfaces based on the specfication of UCM. The order execution of these methods is correct if it satisfies the specification of state transitions in B abstract machines of protocol state machines.

V. Related works

Several proposals for verifying the consistency in component specification have been made.

The paper [9] presents a tool called Cadena, an integrated environment for building and modeling CCM systems. Cadena provides facilities for defining component types using CCM IDL, specifying dependency information and transition system

```
MACHINE SDProtocol
SEES Library
EXTENDS DBProtocol
VARIABLES state1
INVARIANT
    state1 ∈ StateSD
INITIALISATION
    state1 := GET_OK
OPERATIONS
getUpdateInfo =
PRE
    state = Updated /* From DBProtocol machine */
THEN
    state1 := GET_OK
END;
sendInfo =
PRE
    state1 = GET_OK
THEN
    state1 := SEND_OK
END
END
```

Fig. 8. B abstract machine of SDProtocol interface

```
IMPLEMENTATION UCMStock
SEES Library
IMPORTS SBProtocol
REFINES UCM
OPERATIONS
scenario =
BEGIN
    create;
    open;
    update;
    getUpdateInfo;
    sendInfo;
    ...
END
END
```

Fig. 9. UCMStock implementation machine

semantics for these types, assembling systems from CCM components, visualizing various dependence relationships between components, specifying and verifying correctness properties of models of CCM systems derived from CCM IDL, component assembly information, and Cadena specifications, and producing CORBA stubs and skeletons implemented in Java.

As a point of comparison, this paper generated a DSpin model for the scenario that check the number of timeouts issued in a system execution.

In [10], [11], protocols are specified using a temporal logic based approach, which leads to a rich specification for component interfaces. Henzinger and Alfaro [5] propose an approach allowing the verification of interfaces interoperability based on automata and game theories.

Our approach is similar with the paper [12] which proposes a method for component-based software and system development using B. This method uses existing notations and languages with their associated tools: context diagrams for analyzing and structuring the problem, composite structure diagrams for describing the overall system in terms of components and interfaces, sequence diagrams to describe the behavior of each component, and the B formal method for specifying the interfaces of the different components and for proving their interoperability. However, this paper focuses on the development component model but it does not consider the checking the compatibility between UCM and PSM in component models.

The paper [14] proposes the Port State Machine (PoSM) to model the communication on a Port. Building on their experience with behavior protocols, they model an operation call as two atomic events request and response, permitting PoSM to capture the interleaving and nesting of operation calls on provided and required interfaces of the Port. The trace semantics of PoSM yields a regular language. They apply the compliance relation of behavior protocols to PoSMs, allowing to reason on behavior compliance of components in software architectures.

Our work focuses on the verification of consistency of specification in component models. We have found the conformance properties between UCM and PSMs in a component models and propose a B model for verifying these properties.

VI. Conclusion

We have presented an approach for checking component-based software models. In this method, the order execution of methods in each component interfaces is specified by PSMs. The communication of component composition is described by UCM. PSMs and UCM are then transformed to B notations in order to check the consistency between these specifications. Our contribution includes identifying the consistency between UCM and PSMs and proposing a B model which can be used to check their consistency. The proposed approach not only avoids the incompatibility in component model specification but also ensures the execution order of interface methods at runtime.

However, due to limited expression in B notations such that we cannot describe the parallel processing in the implementation machine, we are not possible to express all features of UCMs in a B implementation machine. In the future work, we will consider combining with other methods, for resolving the expression problem of UCM features in a formal way. In addition, an extended approach which checks the conformability between component specifications such as UCM, PSM and the implementation of component model is considered.

Acknowlegments. This work is partially supported by the VietNam National IT Fundamental Research grant 204006.

REFERENCES

[1] Sun Microsystems, JavaBeans 1.01 Specification, http://java.sun.com/beans.

[2] http://www.omg.org.

[3] http://www.usecasemaps.org/.

[4] J.-R. Abrial. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, 1996.

[5] L. Alfaro and T. A. Henzinger. Interface automata. In *9th Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM press, 2001.

[6] G. Booch, J. Rumbaugh, and I. Jacopson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[7] Clearsy. *B4free*. Available at *http://www.b4free.com*, 2004.

[8] G. Eddon and H. Eddon. *Inside COM+ Base Services*. Microsoft Press, 2000.

[9] J. Hatcliff et al. Cadena: an integrated development, analysis, and verification environment for component-based systems. In *Proceedings of 25th International Conference on Software Engineering*, pages 160–172, 2003.

[10] J. Han. A comprehensive interface definition framework for software components. In *Asia Pacific software engineering conference*, pages 110–117. IEEE Computer Society, 1998.

[11] J. Han. Temporal logic based specification of component interaction protocols. In *Proceedings of the Second Workshop on Object Interoperability ECOOP'2000*, pages 12–16. Springer-Verlag, 2000.

[12] D. Hatebur, M. Heisel, and J. Souquieres. A method for component-based software and system development. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 72 – 80. IEEE, 2006.

[13] B-Core(UK) Ltd. *B-Toolkit User's Manual*. Oxford (UK), 1996. Release 3.2.

[14] V. Mencl. Specifying component behavior with port state machines. *Electronic Notes in Theoretical Computer Science*, 101C:129–153, 2004. Special issue: Proceedings of the Workshop on the Compositional Verification of UML Models.

[15] Steria. *Obligations de preuve: Manuel de référence*. Steria - Technologies de l'information, version 3.0. Available at *http://www.atelierb.societe.com*.