PAPER

Checking Behavioral Compatibility between Objects by Extending the Methods Rule

Heung Seok CHAE[†a)], **Joon-Sang LEE**[††b)], ***Nonmembers***, ***and*** **Jung Ho BAE**[†c)], ***Member***

SUMMARY Behavioral compatibility between subtypes and supertypes in object-oriented systems is a very important issue to enable the substitution between object types since it supports the extension and evolution of an object oriented system. In other words, the subtype must be guaranteed that it can provide all behaviors (operations) of the supertype for replacing the supertype with the subtype. Invocation consistency checking is one of techniques to verify behavioral compatibility between two object types. The technique confirms weather an object type can accept all sequence of operations of the other object type or not. The classical methods rule checks behavioral compatibility by verifying invocation consistency of two object types. The rule argues that subtypes meet behavioral compatibility with supertypes if the subtypes' preconditions of inherited operations are weakened and postconditions are strengthened. Noting that the classical methods rule is not sufficient for checking behavioral compatibility between objects, we propose an extended methods rule on the basis of the classical methods rule. Based on the proposed extended methods rule, we have implemented a tool, BCCT, to automatically check behavioral compatibility between two objects.

key words: *object oriented programming*

1. Introduction

An object-oriented system can be viewed as a group of objects collaborating with one another. From the perspective of an individual object, the other objects serve as an environment context. In other words, each object plays its own role in collaborations. During a maintenance phase, some objects might be evolved to accommodate some functional extensions or requirements changes. In such a situation, software system must be preserved with its original set of collaborative behaviors in spite of substitution of such objects. To perform reliable maintenance activities, it is very important to verify whether an object can be safely substituted for another one. This problem is often referred to as *behavioral compatibility*, in contrast to *syntactical compatibility* which only guarantees success in compilation without modifying the environment context. In software system, *syntactical compatibility* can be verified through a static analysis such as signature checking. On the other hand checking compatibility with respect to object behaviors needs more contractual conditions.

The problem of behavioral compatibility could be addressed from the perspective of software reuse and evolution. In object-oriented programming, there is a language feature for maintaining behavioral compatibility based on class inheritance hierarchies. However, it is not guaranteed that all of the collaborative behaviors are preserved in spite of replacing an object O_1 with another object O_2, such that the class of O_1 is a superclass of O_2 class. A newly substituted object must not interfere with the other existing objects. In order to maintain the issue of behavioral compatibility, it is necessary to develop a clear criterion for checking behavioral compatibility in additions to the concept of inheritance or subtyping. In this paper, we propose a new rule, the extended methods rule, for checking behavioral compatibility between objects.

In component-based software development, the issue of behavioral compatibility becomes critical and more important [1]. Commercial Off-The Shelf (COTS) components may be delivered without source code, with which the system assembler could determine how suitable they are to be used within an expected software architecture (i.e. environment context). There would be various kinds of architectural mismatch in reusing existing software components [2]. Although they focused on the problems resulting from the different assumptions on low-level details of interoperability, architectural mismatch can be regarded as a kind of behavioral compatibility.

As one of the fundamental features for object-oriented systems, inheritance defines a relationship between two object types, where one object type called *subtype* inherits all of the structure and behavior from the other object called *supertype*. In general, a subtype object can be used instead of its supertype object. However, a simple use of inheritance does not necessarily guarantee the behavioral compatibility between object types related with inheritance. Instead, the behavior of subtype object should be so cautiously designed as to specialize the behavior of its supertype object according to some clearly defined criteria. Consequently, it would be an important issue to find out necessary and sufficient criteria for the behavioral compatibility between supertype and its subtypes [3]–[5].

As one of such criterion for checking compatibility of dynamic behavior between supertype and subtype, Ebert and Engels [6] pointed out that dynamic behaviors can be compared on the basis of what a user observes (*observation consistency*) and which operations a user may invoke

Manuscript received March 29, 2010.
Manuscript revised August 22, 2010.

[†]The authors are with the Department of Computer Science and Engineering, Pusan National University, Busan 609-735, Republic of Korea.
[††]The author is with Monitor Laboratory, LG Electronics, Republic of Korea.
a) E-mail: hschae@pusan.ac.kr
b) E-mail: tim.lee@lge.com
c) E-mail: jhbae83@pusan.ac.kr
DOI: 10.1587/transinf.E94.D.79

on an object (*invocation consistency*). Observation consistency indicates that each sequence of operation invocations observable in a subtype object must result in an observable sequence of its corresponding supertype object. Invocation consistency requires that each sequence of invocable operations in a supertype object is also invocable behavior for its subtype objects. In other words, all sequences of operations in a supertype object should be preserved by its subtype objects.

The invocation consistency is concerned with the idea that objects of a subtype can be used in the same way as objects of the supertype. For example, *AlarmClock* can be regarded as a subtype of *Clock* if *AlarmClock* can provide extra functionalities in addition to those of *Alarm*. The invocation consistency requires that all operation sequences supported by *Clock* should be applied to objects of its subtype *AlarmClock*. This suggests that objects of *AlarmClock* can be used safely wherever objects of *Clock* are needed. Therefore, the invocation consistency can be a useful criterion for checking behavioral compatibility between a supertype and its subtypes.

In this paper, we present formal definitions for invocation consistency between objects whose dynamic behaviors are expressed as extended finite state machine (EFSM). An EFSM describes the dynamic behavior of an object. That is, an EFSM specifies all possible sequences of method calls which may be invoked on an object. A trace is used to indicate one possible sequence of method calls that could happen on an object. For invocation consistency, each possible trace in a supertype must be also preserved in its subtype.

Our proposed method checks functional behavioral compatibility of a subtype object with its supertype object. That is, the method shows that the subtype object can accept all operations of the supertype, but does not concerned about performance change. For checking non-functional behavior including timing and schedulability, other specification such as temporal-logic and UML Profile for Schedulability, Performance, and Time can be used. In this paper, we note that non-functional properties likes timed constraints are not interested in.

There have been few studies on behavioral compatibility of object behavior represented as a state machine. In this paper, we extend the notion of invocation consistency for a more precise and practical application context by considering guard conditions of transitions. Early approaches to behavioral compatibility are based on signatures of operations, requiring operations of the subtype to be consistent with the corresponding operations of the supertype in terms of pre/post conditions. The basic rule, called methods rule [5], states that at subtypes preconditions of inherited operations are weakened (i.e., pre-condition rule) and postcondition are strengthened (i.e., post-condition rule).

We note that the simple application of the classical methods rule to each transition of trace is not sufficient to guarantee the invocation consistency. This paper proposes an extended methods rule which can guarantee the invocation consistency in a sound fashion. In other words, the proposed rule does not guarantee the self-compatibility. It is the character of the sound rules. However, *Syntactical compatibility* checking is enough to verify compatibility between same types [7], [8].

We also present an algorithm for checking behavioral compatibility using the proposed approach and describe *BCCT*, an automated tool. The BCCT is implemented as a plugin for Together [9] and can extract two EFSMs information from the current project and investigate behavioral compatibility by the proposed extended methods rule.

The remainder of this paper is organized as follows. Section 2 briefly presents the formal definitions of dynamic behavior for object type based on extended finite state machines and illustrates the notion of invocation consistency using a simple example. Section 3 presents an approach to checking invocation consistency by describing the weakness of the classical methods rule and proposing the extended methods rule. Section 4 presents an algorithm and its automated tool, *BCCT*. Section 5 presents the previous work regarding the behaviorial compatibility issues. Section 6 is for concluding remark with a promising future work.

2. Backgrounds

2.1 Dynamic Behavior of Object Type

Definition 1: An Extended Finite State Machine (EFSM) for an object type *ot* is represented by $efsm_{ot} = (S^{ot}, \mathcal{I}^{ot}, O^{ot}, \overrightarrow{x}^{ot}, \Sigma^{ot}, s_0^{ot}, S_\psi^{ot})$ where S^{ot}, $\mathcal{I}^{ot}$, O^{ot}, $\overrightarrow{x}^{ot}$ and Σ^{ot} are finite sets of states, input symbols, output symbols, variables and transitions, respectively, and $s_0^{ot} \in S^{ot}$ and $S_\psi^{ot} \subseteq S^{ot}$ are an initial state and finite set of final states. Each transition $t \in \Sigma^{ot}$ is a 6-tuple: $t = (s_t, s'_t, e_t, o_t, \mathcal{P}_t, \mathcal{Q}_t)$ where s_t, s'_t, $e_t \in \mathcal{I}^{ot}$ and $o_t \in O^{ot}$ are the source (current) state, sink (next) state, input and output, respectively. Predicates $\mathcal{P}_t(\overrightarrow{x})$ and $\mathcal{Q}_t(\overrightarrow{x})$ are a precondition and a postcondition on the current variable values $\overrightarrow{x}$, respectively.

Initially, the machine is at an initial state $s_0 \in S$ with initial variable values $\overrightarrow{x}_0$. Suppose that the machine is at the state s_t with the current variable values $\overrightarrow{x}$. Upon an input e_t, if $\overrightarrow{x}$ is valid for $\mathcal{P}_t$, i.e. $\mathcal{P}_t(\overrightarrow{x})$ = TRUE, then the machine follows the transition t, outputs o_t, changes the current variable values to $\overrightarrow{x}$ which valid for $\mathcal{Q}_t$, and moves to the state s'_t.

A particular behavior of an object type can be described by the sequence of the transitions. And, all the possible sequence of the transitions describe the complete behavior of the object type.

2.2 Invocation Consistency

Informally speaking, the invocation consistency states that each trace possible in the supertype object should be possible in its subtype objects. Let us explain the invocation consistency using simple examples. Figure 1 shows dynamic models of three clock object types using finite state machines. For simplicity, the examples do not include some

features of EFSM such as transition with guard condition.

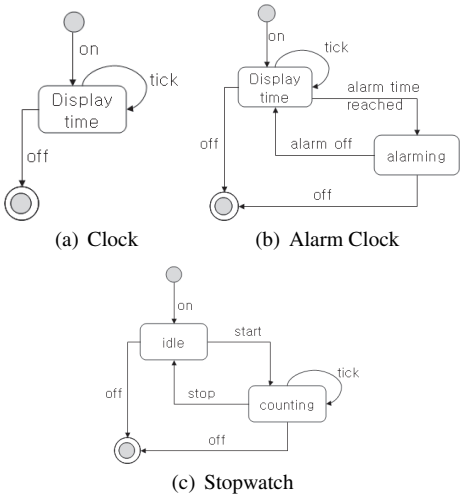
Figure 1 (a) describes a very simple behavior of a clock, which just displays the current time whenever the event *tick* arrives between two events *on* and *off*. Figure 1 (b) shows a simple alarm clock, which additionally supports alarming based on the *Clock*. The alarm clock starts alarming when event *alarm time reached* arrives while displaying the current time, and stops its ringing at the event of *alarm off*. Figure 1 (c) shows a behavior of a simple stopwatch, which displays the elapsed second after the event of *start*. The stopwatch returns to the state *idle* from the state *counting* at the event of stop.

Definition 2: A trace of an EFSM $t_1 \cdot t_{n-1} \cdots t_n$ is a sequence of adjacent transitions; that is, $s'_{t_i} = s_{t_{i+1}}$ for i=1 ...n-1.

We can obviously recognize that all the possible transition sequences in the clock can also occur in the alarm clock. The set of traces of the clock is $on \cdot (tick)^* \cdot off$, which is evidentially allowed in the dynamic model of the alarm clock. Therefore, there is invocation consistency between the clock and the alarm clock. However, some traces of the clock are not possible in the stopwatch. For example, the trace $on \cdot tick \cdot off$ can not be realized in the stopwatch since the stopwatch requires the event *start* before accepting the event *tick*. Therefore, the invocation consistency is not satisfied between the clock and the stopwatch. These relationships between the three objects suggests that objects of alarm clock can be used instead of objects of clock, but objects of stopwatch not.

Invocation consistency indicates that each invocable behavior at the level of a supertype is also an invocable behavior for its subtype. Using the notion of trace in EFSM, invocation consistency can be defined as follows:

Definition 3: (Invocation Consistency) Let T_{ot_1} and T_{ot_2} be sets of all traces of object types ot_1 and ot_2, respectively. If $T_{ot_1} \subseteq T_{ot_2}$, ot_2 is defined to be invocation consistent with ot_1.



(a) Clock (b) Alarm Clock

(c) Stopwatch

Fig. 1 Examples for invocation consistency.

3. Checking Invocation Consistency

This section presents a criterion for invocation consistency in EFSM and gives a simple proof that the presented criteria guarantees invocation consistency. We propose an extended methods rule to overcome shortcoming of the classical methods rule.

3.1 Invocation Consistency in EFSM

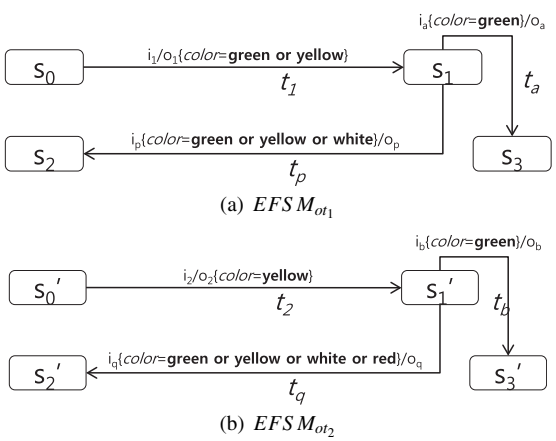
Consider Fig. 2 which shows two EFSMs for two object types ot_1 and ot_2. The diagram has a typical form of an EFSM where each transition is described by a notation *input*{*pre-condition*} / *output*{*post-condition*}. Note that *true* pre/post-condition is omitted for simplicity. Each transition is associated with an unique label for the sake of naming. It is obvious that S_0 and S'_0 are the initial states of $EFSM_{ot_1}$ and $EFSM_{ot_2}$.

Definition 4: (Satisfiability). A pre/post condition $\mathcal{P}$ satisfies an other pre/post condition $\mathcal{Q}$, denoted by $\mathcal{P} \rightarrow \mathcal{Q}$ iff let $\overrightarrow{X} = \{\overrightarrow{x} \mid \mathcal{P}(\overrightarrow{x}) = \text{TRUE}\}$, $\mathcal{Q}(\overrightarrow{x}) = \text{TRUE}$ where $\forall \overrightarrow{x} \in \overrightarrow{X}$.

In Fig. 2, a set of values of variable *color* $\overrightarrow{X}_P$ which valid for $\mathcal{P}_{t_a}$ is {{*color* = **green**}}. And a set of values of variable *color* $\overrightarrow{X}_Q$ which $\mathcal{Q}_{t_1}(color)$ = TRUE is {{*color* = **green**}, {*color* = **yellow**}}. On these two predicates, $\mathcal{P}_{t_a}$ satisfies $\mathcal{Q}_{t_1}$; that is, $\mathcal{P}_{t_a} \rightarrow \mathcal{Q}_{t_1}$ because $\overrightarrow{X}_P \subseteq \overrightarrow{X}_Q$.

Definition 5: (Correspondence between states or transitions).

(1) The initial states of object types ot_1 and ot_2 are simply defined to be correspondent to each other. That is, $s_0^{ot_1}$ corresponds to $s_0^{ot_2}$ and $s_0^{ot_2}$ corresponds to $s_0^{ot_1}$.
(2) State $s_2 \in S^{ot_2}$ corresponds to state $s_1 \in S^{ot_1}$, denoted by $s_2 \Rightarrow s_1$ iff $\forall t_1 \in in\_trans(s1)$, $\exists t_2 \in in\_trans(s2) \cdot t_2$ corresponds to t_1. $in\_trans(s)$ is the set of the transitions leading to the state s. That is, $in\_trans(s) = \{t \in \Sigma | s'_t = s\}$



(a) $EFSM_{ot_1}$

(b) $EFSM_{ot_2}$

Fig. 2 EFSMs of two example objects.

(3) Transition $t_2 \in \Sigma_{ot_2}$ corresponds to transition $t_1 \in \Sigma_{ot_1}$, denoted by $t_2 \Rightarrow t_1$, iff $s_{t_2} \Rightarrow s_{t_1}$, and there exist renaming maps, $R(e_{t_2} = o_{t_1}$ and $R(o_{t_2} = o_{t_1}$, where $R : \mathcal{I} \cup \mathcal{O} \rightarrow \mathcal{I} \cup \mathcal{O}$ is an input/output mapping between the subtype and the corresponding supertype.

The state s_b corresponds to s_a, if there exists some corresponding transition to s_b for each of the incoming transitions to s_a. For example in Fig. 2, state s'_1 corresponds to state s_1 because the transition t_1 is the only incoming transition to the state s_1. At the same time, transition t_2, an incoming transition to s'_1, corresponds to t_1 because the source state of t_2, s'_0, corresponds to the source state of t_1, s_0 by the definition of correspondence between the initial states.

The transition t_2 corresponds to t_1, if the source state of t_2 corresponds to that of t_1, and there is a mapping between inputs/outputs of t_1 and t_2. Let's reconsider $efsm_{ot_1}$ and $efsm_{ot_2}$ in Fig. 2. Transition t_2 can correspond to transition t_1 if the source state of t_2, s'_0, corresponds to the source state of t_1, s_0; that is, $s'_0 \Rightarrow s_0$. At the same time, there exists a mapping, (i_2, i_1) and (o_2, o_1) between the inputs/outputs of t_2 and t_1.

Definition 6: (The methods rule: correspondence between guarded transitions). Transition $t_2 \in \Sigma_{ot_2}$ corresponds to transition $t_1 \in \Sigma_{ot_1}$ with respect to the methods rule, denoted by $t_1 \sqsubseteq^m t_2$, if $[\ t_2 \Rightarrow t_1\ ] \wedge [\ (\mathcal{P}_{t_1} \rightarrow \mathcal{P}_{t_2}) \wedge (\mathcal{Q}_{t_2} \rightarrow \mathcal{Q}_{t_1})\ ]$

This definition is concerned with correspondence between transitions with guard conditions. That is, each transition is associated with its enabling condition (pre-condition) and post-execution condition (post-condition). When considering pre/post-condition, it is not sufficient to check the correspondence between their source states and input/output mapping as discussed in Definition 5.

To take into account pre/post-conditions of transitions, we adopts contra/covariance approach to pre- and postconditions of operations [10]. That is, at subtypes preconditions of inherited operations are weakened and postcondition are strengthened, which permits an instance of a subtype to be safely substituted for an instance of a supertype without runtime errors. This rule was also discussed in [5] and referred to as the methods rule.

We adopts the methods rule in defining correspondence between guarded transitions. The Definition 6 covers the methods rule. That is, the precondition of t_1 is weakened in t_2 and the postcondition of t_1 is strengthened in t_2. We specify the methods rule by implication relations between pre/post conditions of two corresponding transitions. For example, let's consider transitions t_1 and t_2 in Fig. 2. Transition t_2 satisfies the method rule against transition t_1 because the preconditions of t_1 and t_2 are equally true and the postcondition of t_2 implies that of transition t_1. Similarly, transition t_q corresponds to transition t_p with respect to the methods rule since the source state of t_q, s'_1, corresponds to the source state of t_p, s_1, and there can be a mapping (i_p, i_q) and (o_p, o_q). In addition, the precondition of t_p, $\{color = green$ or *yellow* or *white*$\}$ is preserved by that of t_q, $\{color = green$ or *yellow* or *white* or *red*$\}$. The postconditions of them are assumed to be equally true.

This classical methods rule may be used to check invocation consistency. To put it simply, a trace $t_1 \cdot t_2 \cdots t_n$ in $EFSM_{ot_1}$ can occur correspondently on $EFSM_{ot_2}$ if there is a trace $t'_1 \cdot t'_2 \cdots t'_n$ in $EFSM_{ot_2}$ such that $t_i \sqsubseteq^m t'_i$. In other words, the preservation of a trace can be evaluated by applying the methods rule to each corresponding transition pair in the trace. For example, consider a two-step trace $t_1 \cdot t_p$ in $efsm_{ot_1}$. There exists a two-step sequence of transitions $t_2 \cdot t_q$ in $efsm_{ot_2}$ if $t_1 \sqsubseteq^m t_2$ and $t_p \sqsubseteq^m t_q$ hold. Thus, we can argue that the trace $t_1 \cdot t_p$ in $efsm_{ot_1}$ is preserved as $t_2 \cdot t_q$ in $efsm_{ot_2}$.

However, we noted that the simple application of the methods rule to each transition in a trace is not sufficient to satisfy the invocation consistency. That is, the preservation of trace is not guaranteed by checking the methods rule against the corresponding transitions. Consider another trace $t_1 \cdot t_a$ in $efsm_{ot_1}$. Transition t_b obeys the methods rule against transition t_a; that is, $t_a \sqsubseteq^m t_b$. Therefore, it seems that trace $t_2 \cdot t_b$ is the corresponding trace of trace $t_1 \cdot t_a$. However, the trace $t_2 \cdot t_b$ is not possible in $efsm_{ot_2}$ because the precondition of the transition t_b cannot be satisfied by the postcondition of the transition t_2.

Figure 3 is Venn diagrams which show the pre/post conditions of the relevant transitions for describing the difference between trace $t_1 \cdot t_a$ and $t_2 \cdot t_b$. Figure 3 (a) and (b)

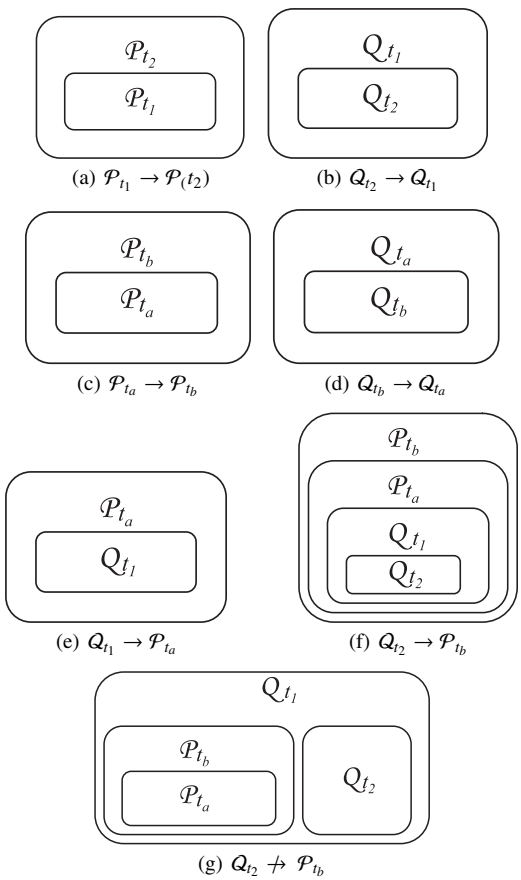


Fig. 3 Venn diagrams for illustrating the methods rules.

illustrate the pre/post condition between transitions t_1 and t_2, and (c) and (d) describe the pre/post condition between t_a and t_b. These are the result of the classical methods rule in Definition 6.

First let us consider the case that transition t_a is always firable after transition t_1. In other words, $\mathcal{Q}_{t_1}$ satisfies $\mathcal{P}_{t_a}$ which is shown in Fig. 3 (e). By composing Fig. 3 (b), (c) and (e), we can obtain Fig. 3 (f) which states that the postcondition of t_2 implies the precondition of t_b; that is, $\mathcal{Q}_{t_2} \rightarrow \mathcal{P}_{t_b}$. Therefore, the trace $t_2 \cdot t_b$, that corresponds to the trace $t_1 \cdot t_a$, is always possible. This is the case of traces $t_1 \cdot t_p$ and $t_2 \cdot t_q$ in Fig. 2.

However, there exists the case that the postcondition of transition t_1 does not imply the precondition of transition t_a. In that case, the execution of transition t_b after transition t_2 is not guaranteed. Figure 3 (g) shows the case that the postcondition of transition t_2 does not imply the precondition of transition t_b when Fig. 3 (e) is not assumed. In summary, trace cannot be preserved when a transition in the trace can be conditionally executed depending on the pre/post conditions. The simple application of the classical methods rule to each transition is not sufficient to check whether a trace in supertypes can be preserved in subtypes. We propose an extended methods rule to resolve such a problem,

def: The extended methods ruledef: The extended methods rule

Definition 7: (The extended methods rule). Transition $t_2 \in \Sigma_{ot_2}$ corresponds to transition $t_1 \in \Sigma_{ot_1}$ with respect to the extended methods rule, denoted by $t_1 \sqsubseteq^s t_2$, if

(1) $t_1 \sqsubseteq^m t_2 \ \wedge$
(2) for all $t_a \in out\_trans(s'_{t_1})$, there exists $t_b \in out\_trans(s'_{t_2})$ such that
(2.1) $[\ t_a \sqsubseteq^m t_b\ ] \ \wedge$
(2.2) $[\ (\mathcal{P}_{t_a} \rightarrow \mathcal{Q}_{t_1}) \rightarrow (\mathcal{Q}_{t_2} \rightarrow \mathcal{P}_{t_b})\ ]$
$out\_trans(s)$ is a set of the transition from the state s. That is, $out\_trans(s) = \{t \in \Sigma | s_t = s\}$.

Based on the classical methods rule expressed by (1), the extended methods rule is designed to guarantee the execution of transition t_b after t_2 even when transition t_a conditionally follows by t_1. The term (2.1) states that some transition t_b, the follower of t_2, should correspond to transition t_a, the follower of t_1, according to the classical methods rule.

Generally, transition t_a can be conditionally executed by its precondition whose evaluation may be affected by the postcondition of its preceding transition t_1. For example, the execution of transition t_a depends on the postcondition of transition t_1 in Fig. 2. In case *color* is set to *green*, transition t_a will be fired. The term (2.2) in Definition 7 ensures that transition t_b is always possible after transition t_2 by enforcing that the postcondition of t_2 should imply the precondition of t_b; that is, $(\mathcal{Q}_{t_2} \rightarrow \mathcal{P}_{t_b})$. In addition, for transition t_a to follow by transition t_1, the precondition of t_a is satisfied by the postcondition of t_1. The first clause in (2.2), $(\mathcal{P}_{t_a} \rightarrow \mathcal{Q}_{t_1})$ is added to incorporate this condition into the extended methods rule.

3.2 Checking Behavioral Compatibility by the Extended Methods Rule

The subtype relation between two EFSM can be defined by applying the extended methods rule against the corresponding transitions.

Definition 8: (Subtype relation between two EFSM). The extended finite machine $efsm_2$ is a subtype of $efsm_1$, denoted by $efsm_1 \prec^s efsm_2$, if there exists a corresponding transition with the extended methods rule in $efsm_2$ for each transition in $efsm_1$. In other words, $efsm_1 \prec^s efsm_2$ if $\forall t_1 \in \Sigma_1, \exists t_2 \in \Sigma_2\ [t_1 \sqsubseteq^s t_2]$

We formally prove that the definition for the subtype relation between the EFSM (Definition 8) is sound for invocation consistency. In an EFSM, there are two types of sequence of transition occurrences: *always-possible* trace (APT) and *conditionally-possible* trace (CPT). The APT means that every transition on that trace can occur whenever the source state from it is activated, and may be selected to occur in an externally non-deterministic fashion. On the other hand, the CPS means that a trace can occur, or not depending on the specific states of local variables, namely conditionally. We assume that all transition traces on an EFSM have its domain as $\{\ (APT \cup CPT)^*\ \}$.

Axiom 1: The transition t_1 can be followed by transition t_2 if $\mathcal{Q}_{t_1} \rightarrow \mathcal{P}_{t_2}$ or $\mathcal{P}_{t_2} \rightarrow \mathcal{Q}_{t_1}$. In the first case, t_1 can be always followed by t_2 (i.e., $t_1 \cdot t_2 \in APT$), and in the second case, t_1 can be conditionally followed by t_2 (i.e., $t_1 \cdot t_2 \in CPT$).

We need the following lemma which plays a central role in proving that Definition 8 is a sufficient condition of the invocation consistency.

Lemma 1: If $efsm_1 \prec^s efsm_2$, for all trace $t_1 \cdot t_a \in APT(efsm_1) \cup CPT(efsm_1)$, there exists some trace $t_2 \cdot t_b$ such that $t_2 \cdot t_b \in APT(efsm_2)$ and $t_2 \Rightarrow t_1$ and $t_b \Rightarrow t_a$.

Case 1: Let $t_1 \cdot t_a \in APT(efsm_1)$. By Axiom 1, we know $\mathcal{Q}_{t_1} \rightarrow \mathcal{P}_{t_a}$. The existence of t_2 and t_b such that $t_1 \sqsubseteq^s t_2$ and $t_a \sqsubseteq^s t_b$ is satisfied by Definition 8. Using Definition 7, we know $\mathcal{P}_{t_a} \rightarrow \mathcal{P}_{t_b}$ and $\mathcal{Q}_{t_2} \rightarrow \mathcal{Q}_{t_1}$. Applying transitivity on the three predicates, we conclude that $\mathcal{Q}_{t_2} \rightarrow \mathcal{P}_{t_b}$. Hence, by Axiom 1, $t_2 \cdot t_b \in APT(efsm_2)$

Case 2: Let $t_1 \cdot t_a \in CPT(efsm_1)$. The existence of t_2 and t_b such that $t_1 \sqsubseteq^s t_2$ and $t_a \sqsubseteq^s t_b$ is satisfied by Definition 8. From Definition 7 (2), we know $[\ (\mathcal{P}_{t_a} \rightarrow \mathcal{Q}_{t_1}) \rightarrow (\mathcal{Q}_{t_2} \rightarrow \mathcal{P}_{t_b})\ ]$. Applying Axiom 1, $\mathcal{P}_{t_a} \rightarrow \mathcal{Q}_{t_1}$. Thus, $\mathcal{Q}_{t_2} \rightarrow \mathcal{P}_{t_b}$. Therefore, following the above axiom, $t_2 \cdot t_b \in APT(efsm_2)$

Theorem 1: Definition 8 is a sufficient condition of the invocation consistency.

Definition 8 ensures that all elements of transition sequence in $efsm_1$ are also included by its subtype EFSM $efsm_2$. In other words, $APT(efsm_1) \cup CPT(efsm_1) \subseteq APT(efsm_2)$. Now, we can prove Theorem 1 by each trace

that could occur in $efsm_1$ has a corresponding trace in $efsm_2$.

The proof is done by mathematical induction on the length of transitions that compose the trace. We first consider a trace of two transitions. And then, assuming that a trace of length n is satisfied, we try to prove that the theorem is satisfied for a trace of length $n + 1$.

Proof:

Theorem 1 can be rewritten as follows:

If $efsm_1 \prec^s efsm_2$, there is a corresponding trace tr_2 in $efsm_2$ for each trace tr_1 in $efsm_1$

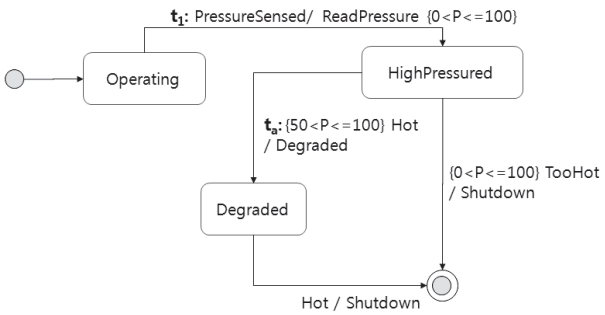
Base case: Assume that the length of tr_1 is 1: The trace consists of only one transition t_1, which by Definition 8 has a corresponding transition t_2 such that $t_1 \sqsubseteq^s t_2$. According to Definition 7, $\mathcal{P}_{t_1} \rightarrow \mathcal{P}_{t_2}$. Therefore, if the transition t_1 occurs in $efsm_1$, t_2 also can occur in $efsm_2$. Hence, the invocation consistency is satisfied.

Induction case: Let us assume that for a trace tr_1 of the length n in $sfsm_1$, there is a corresponding trace tr_2 of the same length in $efsm_2$. Let t_1 and t_2 be the last transition of tr_1 and tr_2, respectively.

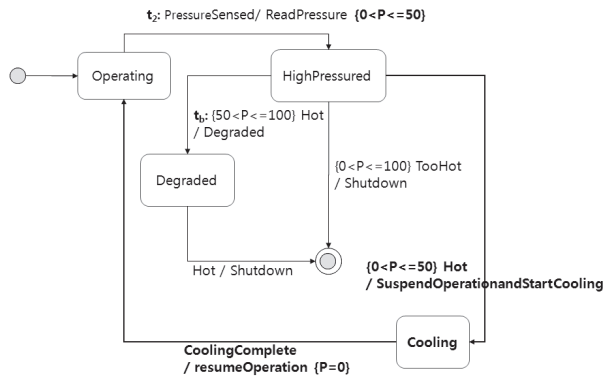
We want to show that for a new trace tr_1' of the length $n + 1$ such that a transition t_a can follow t_1 in $efsm_1$, there is a transition t_b that follows t_2 in $efsm_2$. If t_a can follow t_1 in $efsm_1$, then $t_1 \cdot t_a \in APT(efsm_1) \cup CPT(efsm_1)$ by Axiom 1. By Lemma 1, we see that there must exist some t_b that follows t_2 in $efsm_2$. That is, $t_2 \cdot t_b \in APT(efsm_2)$. Thus, a trace of the length $n + 1$ which is constructed by appending one transition to a trace of the length n in $efsm_1$ can be preserved in $efsm_2$.

Figure 4 shows two EFSMs of *NormalReactor* and *LightReactorWithCooler*. *NormalReactor* controls reactions of a nuclear reactor. Initially, a reactor operates normally, generating electricity from chain reaction of the reactor. When *NormalReactor* reads an event *PressureSensed*, it changes its state into *HighPressured*. The acceptable pressure is assumed to range from 0 to 100, which is denoted by the transition t_1. On the state *HighPressured*, it can accept two events concerning the temperature of the reactor. When an event *TooHot* is received, it shutdowns the reactor. On the contrary, when an event *Hot* is received and the pressure ranges from 50 to 100, it degrades the reactor. When it receives *Hot* once more, it shutdowns the reactor.

LightReactorWithCooler is intended to be an extension of *NormalReactor*. That is, *LightReactorWithCooler* is very similar to *NormalReactor* except two features. First, *LightReactorWithCooler* is designed to operate with less pressure. That is, *LightReactorWithCooler* assumes that its operating pressure ranges from 0 to 50, which is denoted by the transition t_2. Second, *LightReactorWithCooler* can suspend the reactor and start cooling when it receives *Hot*, which is denoted by the transition from *HighPressured* to



(a) $EFSM_{NormalReactor}$



(b) $EFSM_{LightReactorWithCooler}$

Fig. 4 EFSMs of Normal Reactor and Light Reactor with Cooler.

Cooling. When cooling completes, the reactor can resume its operation.

LightReactorWithCooler seems to be behaviorally compatible with *NormalReactor* since *LightReactorWithCooler* looks to be an extended version of *NormalReactor*. However some traces that are allowed in *NormalReactor* cannot be observed in *LightReactorWithCooler*. For example, *NormalReactor* can shutdown the reactor when it receives *PressureSensed* with $50 < P \leq 100$, *Hot*, and *Hot*. However, this trace cannot be possible in *LightReactorWithCooler*.

This incompatibility cannot be checked only by the classical methods rule. All the transitions except t_1 in *NormalReactor* have the identical transitions in *LightReactorWithCooler*. Therefore those identical transitions obviously satisfy the classical methods rule. The transition t_1 in *NormalReactor* and the transition t_2 in *LightReactorWithCooler* satisfy the classical methods rule; that is $Q_{t_2} \rightarrow Q_{t_1}$. Even though all the transitions in *NormalReactor* satisfy the classical methods rule against *LightReactorWithCooler*, we cannot discover that the trace $HighPressure \cdot Hot \cdot Hot$ cannot be preserved by *LightReactorWithCooler*.

On the contrary, the transition t_1 cannot correspond to the transition t_2 according to the extended methods rule. The extended methods rule requires that trace $t_2 \cdot t_b$ should always possible in *LightReactorWithCooler* for the conditional trace $t_1 \cdot t_a$ in *NormalReactor*. Since Q_{t_2} does not always imply $\mathcal{P}_{t_b}$, we can realize that the trace may not be observed in *LightReactorWithCooler*.

4. Tool Support

This section presents an algorithm for checking behavioral compatibility using the proposed extended methods rule and then describes an automated tool for supporting our approach.

4.1 Algorithm for Checking Behavioral Compatibility

Figure 5 shows an algorithm for checking behavioral compatibility between objects using the proposed extended methods rule.

The function *CheckBehavioralCompatibility()* determines the behavioral compatibility between the given two EFSMs SM^1 and SM^2. The function consists of two phases: Phase 1 for checking correspondence between states and transitions and Phase 2 for checking the extended methods rule between two corresponding transitions.

In Phase 1, the correspondence between two EFSMs on the basis of their structures only; that is, guards on transitions are ignored. Phase 1 is realized by *CheckStateCorrespondence()* and *CheckTransitionCorrespondence()*. Two functions are the straight implementations of Definitions 5. Pairs of corresponding states and transitions are maintained in *CS* and *CT*, respectively. *PCT* is used to consider traces with cycle; that is, a state may depend on each other in a cyclic manner. Each state/transition in a cycle is set to be correspondent only when all states/transitions in the cycle are already "corresponding" or "partially corresponding".

In Phase 2, the proposed extended methods rule is applied to each transition pair in *CT*. First, the conventional methods rule is evaluated against the given two transitions themselves. And then, the following transitions are investigated according to the Definition 7 (2).

4.2 BCCT: Behavioral Compatibility Checking Tool

We have developed a tool, named BCCT, to support the automated analysis for checking behavioral compatibility based on the proposed extended methods rule. Figure 6 is a screen shot of the tool.

The BCCT has been implemented as a plugin module on the Together Platform. Together is one of the popular UML modeling tools. By using the UML modeling functions from Together, the BCCT can focus only on the analysis of behavioral compatibility. Together provides an open API for accessing and manipulating diagrams. Using the open API, the BCCT extracts necessary information from state machine diagrams in the currently active project.

Initially, a developer describes dynamic behaviors of two objects with two state machine diagrams using Together. Currently the state machine diagram for *NormalReactor* is shown. Developers can interact with the BCCT by the lower pane, named *BCCT*. The leftmost pane shows all state machine diagrams in the current project. The button "Retrieve StateChart Diagrams" is used to extract all state machine diagrams from the current project. The next pane contains two state machine diagrams to be checked; the first diagram regarded as a supertype and the second as a subtype. Developers can select/deselect state machine diagrams to be checked from the leftmost pane by two buttons "==>" and "<==" in the rightmost pane. The message box in the center shows the result of the behavioral compatibility checking between *NormalReactor* and *LightReactor*-

```
let SM^1 = (S^1, s_0^1, S_ψ^1, I^1, O^1, σ^1, V^1), let SM^2 = (S^2, s_0^2, S_ψ^2, I^2, O^2, σ^2, V^2)

// CS: a set of corresponding states
// CT: a set of corresponding transitions
// PCT: a set of partially corresponding transitions

function CheckBehaviorCorrespondence(SM^1, SM^2) boolean begin
CS = ∅, CT = ∅
//Phase 1 : Finding Corresponding States and Transitions
 for each s_1 ∈ S^1 begin
  is_corespondence = false
  for each s_2 ∈ S^2
   PCT = ∅
   if CheckStateCorrespondence(s_1, s_2) then begin
   is_corespondence = true; break
  end if
 end for
 if not is_corespondence then return false
end for

//Phase 2 : Checking Extended Methods Rule
 for each < t_1, t_2 > ∈ CT
  if not CheckExtendedMethodsRule(t_1, t_2) then return false
 return true
end function

function CheckStateCorrespondence(s_1 ∈ S^1, s_2 ∈ S^2) boolean begin
 if (s_1 = S_0^1 ∧ s_2 = S_0^2) ∨ < s_1, s_2 > ∈ CS then return true
 T_1 = in_trans(s_1), T_2 = in_trans(s_2)
 for each t_1 ∈ T_1 begin
  find_correspondence = false
  for each t_2 ∈ T_2 such that there is a mapping t_1 to t_2 begin
   if CheckTransitionCorrespondence(t_1, t_2) then begin
   find_correspondence = true; break
   end if
  end for
  if not find_correspondence then return false
 end for
 CS = CS ∪ {< s_1, s_2 >}
 return true
end function

function CheckTransitionCorrespondence(t_1 ∈ σ^1, t_2 ∈ σ^2) boolean begin
 if < t_1, t_2 > ∈ CT ∨ < t_1, t_2 > ∈ PCT then return true
 PCT = PCT ∪ {< t_1, t_2 >}
 if not CheckStateCorrespondence(source(t_1), source(t_2)) then return false
 CT = CT ∪ {< t_1, t_2 >}
 return true
end function

function CheckExtendedMethodsRule(t_1 ∈ σ^1, t_2 ∈ σ^2) boolean begin
 //check Definition 5 (1)
 if not (pre(t_1) → pre(t_2) ∧ (post(t_2) → post(t_1)) then return false
 //check Definition 5 (2)
 for each t_a ∈ out_trans(sink(t_1)) begin
  let t_b such that < t_a, t_b > ∈ CT
  if post(t_1) → pre(t_a) then continue
  else if pre(t_a) → post(t_1) and not post(t_2) → pre(t_b) then return false
  end if
 end for
 return true
end function
```

Fig. 5 An algorithm for checking behavioral compatibility.

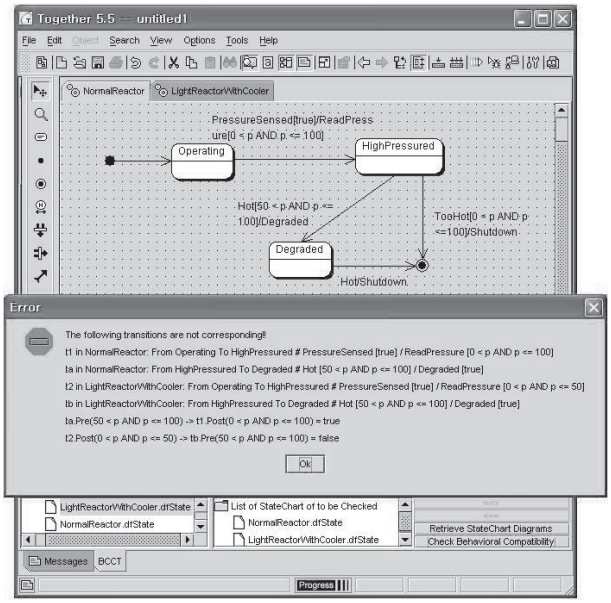


Fig. 6 A screen shot of BCCT under Together platform.

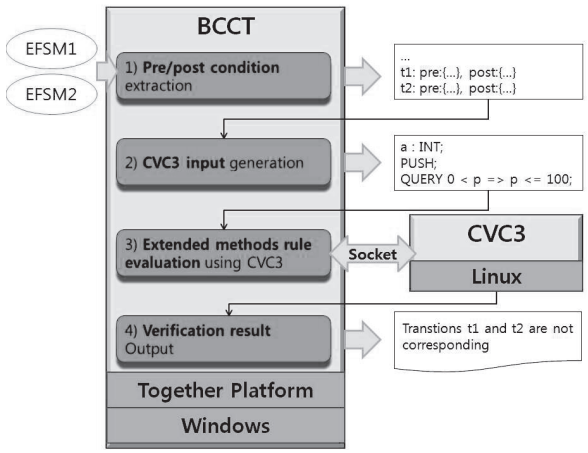


Fig. 7 An overview of BCCT.

WithCooler. The message says that the transitions *t1* and *t2* are not corresponding.

Figure 7 shows a logical architecture of BCCT. The figure illustrates the main modules of BCCT for checking behavioral compatibility between two statechart diagrams using the proposed approach.

We have adopted CVC3 [11] to support an automatic evaluation of the extended methods rule. CVC3 is an automatic theorem prover for determining the satisfiability of a first order formula. CVC3 is the latest in the Cooperating Validity Checker family of tools, building on its predecessors, CVC [12] and CVC Lite [13]. The base versions of CVC3 have several applications: a proof-producing decision procedure for HOL Light [14]; a verification tool for C programs [15], a translation validator for optimizing compilers [16], and a study on the verification of clock synchronization algorithms [17].

t1 in NormalReactor : From Operating To HighPressured # Pressure-Sensed[true] / ReadPressure[0 < p and $p \leq 100$]
ta in NormalReactor : From HighPressured To Degraded # Hot [50 < p and p ≤ 100] / Degraded [true]

t2 in LightReactorWithCooler : From Operating To HighPressured # PressureSensed [true] / ReadPressure [0 < p and $p \leq 50$]
tb in LightReactorWithCooler : From HighPressured To Degraded # Hot [50 < p and p ≤ 100] / Degraded [true]

Fig. 8 Some transitions extracted from state machines *NormalReactor* and *LightReactorWithCooler*.

The evaluation of the extended method rules are performed with the help of CVC3. Therefore, BCCT supports the pre- and postconditions which can be expressed in the CVC3. CVC3 supports a variety of types: rational and integer linear arithmetic, arrays, tuples, records, inductive data types, bit vectors and quantifier. CVC3 supports many fundamental operators: arithmetic operators, comparison operators and logical operators. Therefore, we think that pre/post conditions can be easily expressed in CVC3. Therefore, pre/post conditions can be expressed by a formula which consists of primitive and composite variables and fundamental operators such as arithmetic, comparison and logical operators. For a more detailed and complete information, you can refer to the CVC3 User's Manual [18].

1. **Pre/post condition extraction**. Initially, pre/post condition specifications are extracted from each state machine. BCCT can automatically extract them from UML state machine diagrams. For example, Fig. 8 is part of the transition specifications including pre/post conditions which were extracted from *NormalReactor* and *LightReactorWithCooler*. These are same transition specifications with the message showed in Fig. 6. Each line represents a specification of a particular transition by the internal identifier, the source state, the destination state, an input event with precondition, and an output event with postcondition. For example, the transition *t1 in NormalReactor* is one from *Operating* to *HighPressured* in *NormalReactor*.
2. **CVC3 input generation**.
   CVC3 requires a special form of input which describes the formula to be verified. BCCT can automatically generate such an input file to CVC3 from the pre/post conditions extracted at the previous step.
   As seen from the algorithm in POFFig. 5, the function $CheckExtendedMethodsRule\ (t_1 \in \Sigma^{NormalReactor}, t_2 \in \Sigma^{LightReactorWithCooler})$ depends on five implication relationships between pre/post conditions of transitions: $\mathcal{P}_{t_1} \rightarrow \mathcal{P}_{t_2}$, $\mathcal{Q}_{t_2} \rightarrow \mathcal{Q}_{t_1}$, $\mathcal{Q}_{t_1} \rightarrow \mathcal{P}_{t_a}$, $\mathcal{P}_{t_a} \rightarrow \mathcal{Q}_{t_1}$ and $\mathcal{Q}_{t_2} \rightarrow \mathcal{P}_{t_b}$.
   Figure 9 shows the input and the output of CVC3 to evaluate those five formulas for the transitions t_1(=t1 in NormalReactor) and t_2(= t1 in LightReactorWithCooler) in Fig. 4. The first column represents the formula to be evaluated. The second column describes the input for CVC3 to evaluate the implication of the for-

Formula	CVC3	
	Input	Output
pre(t_1) -> pre(t_2)	PUSH; QUERY (TRUE) => (TRUE); POP;	Valid.
post(t_2) ->post(t_1)	p : INT; PUSH; QUERY (0 < p AND p <= 50) => (0 < p AND p <= 100); POP;	Valid.
post(t_1) -> pre(t_a)	p : INT; PUSH; QUERY (0 < p AND p <= 100) => (50 < p AND p <= 100); POP;	Invalid.
pre(t_a) -> post(t_1)	p : INT; PUSH; QUERY (50 < p AND p <= 100) => (0 < p AND p <= 100); POP;	Valid.
post(t_2) -> pre(t_b)	p : INT; PUSH; QUERY (0 < p AND p <= 50) => (50 < p AND p <= 100); POP;	Invalid.

* t_1 = t1 sm1, t_a = t2 sm1, t_2 = t1 sm2, t_b = t2 sm2

Fig. 9 Evaluation of extended methods rule using CVC3.

mulas given in the first column. Command *QUERY* in CVC3 is used for evaluating each implication formula.

3. **Extended methods rule evaluation using CVC3**. By invoking CVC3, we can automatically evaluate the implication relationship between pre condition and post condition and then evaluate the extended methods rule. In Fig. 9, the third column represents the output from CVC3 for the input in the second column.
   As seen from the figure, $\mathcal{P}_{t_1} \rightarrow \mathcal{P}_{t_2}$ is valid, $\mathcal{Q}_{t_2} \rightarrow \mathcal{Q}_{t_1}$ is valid, $\mathcal{Q}_{t_1} \rightarrow \mathcal{P}_{t_a}$ is invalid, $\mathcal{P}_{t_a} \rightarrow \mathcal{Q}_{t_1}$ **is valid and** $\mathcal{Q}_{t_2} \rightarrow \mathcal{P}_{t_b}$ **is invalid**. Accordingly, the transition t_1 in *NormalReactor* violates the extended methods rule. Therefore, a trace $t_1 \cdot t_a$ in *NormalReactor* does not have a corresponding trace in *LightReactorWithCooler*, which suggests that *LightReactorWithCooler* is not behaviorally compatible with *NormalReactor*.
4. **Verification result output**. BCCT outputs the result of checking behavioral compatibility between two state machines. If they are evaluated not to be correspondent, BCCT displays the transition pairs which are not correspondent.

5. Related Work

Several work has treated the problem of behavioral compatibility between object types; that is, the verification of the behavioral conformance of subtype objects to that of its supertype object. Some of the research on defining subtype relations is concerned with capturing constraints on method signatures via the contra/covariance approach. According to the contra/covariance approach, the domains of input parameters are generalized and the domains of output parameters are specialized at subtypes [3], [19].

Design by contracts [10] applies contra/covariance approach to pre- and postconditions of operations. That is, at subtypes preconditions of inherited operations are weakened and postcondition are strengthened, which permits an instance of a subtype to be safely substituted for an instance of a supertype without run-time errors. Pre/post conditions are widely used to specify the behavior of procedures or methods, and to check the behavioral compatibility. However, the constraint, referred to as the methods rule in [5], between the contracts of a type and the contracts of its subtype does not sufficiently address the behavioral compatibility with respect to invocation consistency.

Findler and Felleisen discussed the contract soundness on the basis of the Java operational semantics, but they little addressed the issues on behavioral compatibility [20]. As addressed by many researchers, the methods rule for behavioral compatibility is not sufficient to check the properties of supertype objects and subtype objects. For example, Coleman et al. briefly described that some liveness properties cannot always be preserved only by the methods rule [21].

Liskov and Wing made an important contribution in the area of programming languages [5]. Using the Larch specification language, they defined subtype relations in terms of implications between pre- and postconditions of individual mutator operations plus additional constraints. Based on Larch++, Dhara and Leavens extended the work of Liskov and Wing [5] by generalizing the scope of the consistency criteria and by adding an additional consistency type, weak behavioral subtyping [22], which corresponds to invocation consistency discussed in this paper. Although they provide explicit criteria for subtype relation between individual operations, but subtype relation is not addressed from the viewpoint of dynamic behavior of objects.

There are researches on behavioral subtyping in the realm of state diagrams [6], [23]. Those approaches tried to make a mapping between super- and subtypes based on graph (homo-)morphisms, similar to the work of Ehrich and Sernadas [24], [25]. Although they considered transitions with guards, their approach is concerned with observation consistency and guard conditions are just considered between two corresponding transitions, not under the context of trace.

Schrefl and Stumptner [26]–[28] presented the formal definitions for the two kinds of consistency under the context of object behavior diagram, which is similar to Petri-net. In addition, they classified invocation consistency into weak invocation consistency, which corresponds to the notion of invocation consistency discussed in this paper, and strong invocation consistency. They proposed a set of necessary and sufficient rules for checking behavior consistency between object life cycles of object types in specialization hierarchies. The object life cycle can be represented with not only a set of operations but also an evolution fashion over time. So, the concept of object life cycle is similar to trace of object types in this paper. They define the behavior checking rules in the realm of object behavior diagrams, something like Petri-net. However, the object behavior diagram is not so popular as state diagram; tool support may not be so easy about object behavior diagrams as state diagram. In addition, the incorporation of guarded transitions

are not explicitly considered in their approach.

Fischer and Wehrheim proposed four behavioral subtyping relations based on the process algebra CSP [29] in the context of distributed systems [30], [31]. Recently Wehrheim tried to propose a systematic view of subtyping for specification integrating state-based and behavior-based views [32]. Based on that work, Olderog and Wehrheim [33] investigated the notion of inheritance with CSP-OZ [30], [34], which is a combination of CSP and Object-Z.

The simulation or bi-simulation relation [35] and language containment relation [36], [37] lay its computation model on process algebra such as CSP, CCS or π-calculus [38]. The notion of language containment can be used to examine whether or not a language L is contained to a language $L0$ by checking the intersection of L and the complemented $L0$. A bi-simulation is a binary relation to verify if one state transition system simulates the other one, and vice versa. These two standard definitions are based on theoretical model of Labeled Transition System (LTS), where each element is associated with a propositional event label. Therefore, bi-simulation could be considered for the problem of behavioral compatibility. However, as far as we know, pre/post conditions are not generally considered in LTS and even bi-simulation has not been considerably discussed for LTS with pre/post conditions.

Recently, there are many ongoing researches toward behavioral compatibility analysis about Web Service. To composes web services, Z. Wu et al. and P. Xiong et al. proposed checking methodology which considerate under context [39], [40]. Z. Wu et al. adopted π-calculus formalism to model service behaviors and interactions in a formal way. P. Xiong et al. modeled multiple web services interaction with a Petri-net called Composition net (C-net for short). The two studies introduced same approaches which validate compatibility of web services in an interaction aspect. That is, they have focused on interaction with other web services rather than correct behavior of supertype. Our study has verified whole behavior compatibility of object with it's supertype but, they had verified some operations only what interact with other objects (web services in their research). Moreover they didn't consider the pre/post condition of operations also.

6. Conclusion and Future Work

In this paper, we have proposed an approach to checking the behavioral compatibility between object types. The proposed approach is based on dynamic object models, i.e. extended finite state machines. By extending classical methods rule, we suggested the extended methods rule which can be used to check the behavioral compatibility between a supertype and a subtype. In addition, we described BCCT to automate the proposed approach. BCCT, implemented on Borland Together Platform, extracts pre/post conditions from UML state diagrams and verifies behavioral compatibility based on the extended methods rule.

We are going to extend the scope of our approach to checking behavioral compatibility of components in component-based development [41] and services in SOA [42]. In particular, behavioral compatibility between components can be a crucial issue because the maintainability and extensibility of component-based systems can be achieved mainly by replacing one component with another one. To guarantee the reliable operation of the systems even after the replacement of some components, it is very important to verify that the new component provides a behavior compatible with the old one. Our extended methods rule can also be applied to verifying the behavioral compatibility between such components. There are a lot of works for checking behavioral compatibility between components [43]–[46] and between services [47]–[50]. To the best of our knowledge, they do not, however, take into account the notion of dynamic behaviors with pre/post conditions.

Acknowledgments

This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD)" (The Regional Research Universities Program/Institute of Logistics Information Technology).

References

[1] A. Brown and K. Wallnau, "Engineering of component-based systems," Component-Based Software Engineering: Selected Papers from the Software Engineering Institute, pp.7–15, IEEE Computer Society Press, 1996.
[2] D. Garlen, R. Allen, and J. Ockerbloom, "Architectural mismatch or why it's hard to build systems out of existing parts," Proc. ICSE '95, pp.179–185, 1995.
[3] P. Wegner and S. Zdonik, "Inheritance as an incremental modification mechanism or what like is and isn'tlike," ECOOP '88 (European Conference on Object-Oriented Programming), pp.55–77, Springer-Verlag, London, UK, 1988.
[4] P. America, "Designing an object-oriented programming language with behavioural subtyping," Proc. REX School/Workshop on Foundations of Object-Oriented Languages, pp.60–90, Springer-Verlag, London, UK, 1991.
[5] B. Liskov and J. Wing, "A behavioral notion of subtyping," ACM Transactions on Programming Languages and Systems, vol.15, no.6, pp.1911–1841, Nov. 1994.
[6] J. Ebert and G. Engels, "Observable or invocable behaviour.you have to choose," Tech. Rep., Universit: at Koblenz, Koblenz, Germany, 1994.
[7] G. Decker and M. Weske, "Behavioral consistency for b2b process integration," Advanced Information Systems Engineering, vol.4495, pp.81–95, June 2007.
[8] W. van der Aalst, "The application of petri nets to workflow management," J. Circuits, Syst. Comput., vol.8, no.1, pp.21–66, 1998.
[9] Boland, "Boland together." http://www.borland.com/us/products/together/index.html.
[10] B. Meyer, "Design by contract," Computer, vol.25, no.10, pp.40–51, Oct. 1992.
[11] C. Barrett and C. Tinelli, "CVC3," CAV, pp.298–302, 2007.
[12] A. Stump, C.W. Barrett, and D.L. Dill, "CVC: A cooperating validity checker," CAV, pp.500–504, 2002.
[13] C.W. Barrett and S. Berezin, "CVC Lite: A new implementation of the cooperating validity checker category B," CAV, pp.515–518, 2004.

[14] S. McLaughlin, C. Barrett, and Y. Ge, "Cooperating theorem provers: A case study combining HOL-Light and CVC Lite," Electr. Notes Theor. Comput. Sci., vol.144, no.2, pp.43–51, 2006.
[15] J.C. Filliâtre and C. Marché, "Multi-prover verification of C programs," ICFEM, pp.15–29, 2004.
[16] C.W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L.D. Zuck, "TVOC: A translation validator for optimizing compilers," CAV, pp.291–295, 2005.
[17] D. Barsotti, L.P. Nieto, and A.F. Tiu, "Verification of clock synchronization algorithms: Experiments on a combination of deductive tools," Electr. Notes Theor. Comput. Sci., vol.145, pp.63–78, 2006.
[18] C. Barrett and C. Tinelli, "The CVC3 user's manual." http://www.cs.nyu.edu/acsys/cvc3/doc/index.html, 2007.
[19] P. Canning, W. Cook, W. Hill, and W. Olthoff, "Interfaces for strongly-typed object-oriented programming," OOPSLA '89: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, pp.457–467, ACM Press, New York, USA, 1989.
[20] R.B. Findler and M. Felleisen, "Contract soundness for object-oriented languages," Proc. OOPSLA '01, pp.1–15, 2001.
[21] D. Coleman, F. Hayes, and S. Bear, "Introducing objectcharts of how to use Statecharts in object-oriented design," IEEE Trans. Softw. Eng., vol.18, no.1, pp.9–18, Jan. 1992.
[22] K. Dhara and G. Leavens, "Forcing behavioral subtyping through specification inheritance," Proc. 18th International Conference on Software Engineering, pp.258–267, IEEE Computer Society Press, Berlin, Germany, 1996.
[23] G. Saake, R. Jungclaus, R. Wieringa, and R. Feenstra, "Inheritance conditions for object life cycle diagrams," Proc. EMISA, pp.79–88, 1994.
[24] H.D. Ehrich, J. Goguen, and A. Sernadas, "A categorial theory of objects as observed processes," Proc. Foundations of Object-Oriented Languages (REX School/Workshop), pp.203–228, 1990.
[25] A. Sernadas and H.D. Ehrich, "What is an object, after all?," Proc. IFIP WG 2.6 Working Conference on Object-oriented Databases: Analysis, Design and Construction, pp.39–70, 1991.
[26] M. Schrefl and M. Stumptner, "Behavior consistent extension of object life cycles," Proc. OOER'95, pp.133–145, 1995.
[27] M. Schrefl and M. Stumptner, "Behavior consistent refinement of object life cycles," Proc. ER'97, pp.155–168, 1997.
[28] M. Schrefl and M. Stumptner, "Behavior-consistent specialization of object life cycles," ACM Trans. Softw. Eng. Methodol., vol.11, no.1, pp.92–148, Jan. 2002.
[29] C. Hoare, Communicating Sequential Process, Prentice Hall, 1985.
[30] C. Fischer and H. Wehrheim, "Behavioural subtyping relations for object-oriented formalisms," Lect. Notes Comput. Sci., vol.1816, pp.469–484, 2000.
[31] H. Wehrheim, "Behavioral subtyping relations for active objects," Form. Methods Syst. Des., vol.23, no.2, pp.143–170, 2003.
[32] H. Wehrheim, "Behavioral subtyping relations for active objects," Formal Methods in System Design, vol.23, no.2, pp.143–170, 2003.
[33] E.R. Olderog and H. Wehrheim, "Specification and (property) inheritance in CSP-OZ," Sci. Comput. Program., vol.55, no.1-3, pp.227–257, 2005.
[34] C. Fischer, "CSP-OZ: a combination of Object-Z and CSP," Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS), ed. H. Bowman and J. Derrick, pp.423–438, Chapman and Hall, Canterbury, UK, London, 1997.
[35] R. Milner, Communication and concurrency, Prentice-Hall, Upper Saddle River, NJ, USA, 1989.
[36] H. Touati, R.K. Brayton, and R. Kurshan, "Testing language containment for ω-automata using BDDs," Inf. Comput., vol.119, no.1, pp.101–109, April 1995.
[37] B. Finkbeiner, "Language containment checking with nondeterministic BDDs," TACAS 2001: Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp.24–38, Springer-Verlag, London, UK, 2001.
[38] R. Milner, Communicating and mobile systems: the π-calculus, Cambridge University Press, 1999.
[39] Z. Wu, S. Deng, Y. Li, and J. Wu, "Computing compatibility in dynamic service composition," Knowl. Inf. Syst., vol.19, no.1, pp.107–129, 2009.
[40] Y.F.P. Xiong and M. Zhou, "A petri net approach to analysis and composition of web services," IEEE Trans. Syst., Man Cybern., A, Syst. Humans, vol.40, no.2, pp.376–387, 2010.
[41] P. Vitharana, "Risks and challenges of component-based software development," Commun. ACM, vol.46, no.8, pp.67–72, 2003.
[42] S. Jones, "Toward an acceptable definition of service," IEEE Softw., vol.22, no.3, pp.87–93, 2005.
[43] N. Hameurlain, "On compatibility and behavioural substitutability of component protocols," SEFM, pp.394–403, 2005.
[44] J. Souquières and S. Chouali, "Verifying the compatibility of component interfaces using the B formal method," Software Engineering Research and Practice, pp.850–856, 2005.
[45] L. Wang and P. Krishnan, "A framework for checking behavioral compatibility for component selection," ASWEC, pp.49–60, 2006.
[46] P.C. Attie, D.H. Lorenz, A. Portnova, and H. Chockler, "Behavioral compatibility without state explosion: Design and verification of a component-based elevator control system," CBSE, pp.33–49, 2006.
[47] M. Mecella, B. Pernici, and P. Craca, "Compatibility of e -services in a cooperative multi-platform environment," TES '01: Proc. 2nd International Workshop on Technologies for E-Services, pp.44–57, Springer-Verlag, London, UK, 2001.
[48] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Compatibility verification for web service choreography," ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04), p.738, IEEE Computer Society, Washington, DC, USA, 2004.
[49] V.D. Antonellis, M. Melchiori, and P. Plebani, "An approach to web service compatibility in cooperative processes," SAINT-W '03: Proc. 2003 Symposium on Applications and the Internet Workshops (SAINT'03 Workshops), p.95, IEEE Computer Society, Washington, DC, USA, 2003.
[50] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella, "When are two web services compatible?," TES, pp.15–28, 2004.

Heung Seok Chae received the BS degree in nuclear engineering from Seoul National University in 1994 and the MS and PhD degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 1996 and 2000, respectively. He worked as a senior consultant for the TongYang Systems between 2000 and 2002. During the year of 2003, he was as a visiting professor in the Department of Computer Science at KAIST. Since 2004, he has been on the faculty of the Department of Computer Engineering, Pusan National University, Busan, Korea. His current research interests include object-oriented analysis and design, component-based software development, component testing, software metrics, middleware architecture for availability, and scalability. He is a member of the Korea Information Science Society.

Joon-Sang Lee received the B.S. degree in computer engineering from Dongguk University, Seoul, Korea, and the M.S. and Ph.D. degrees in computer science from KAIST, Daejeon, Korea. He was a senior researcher for LG Electronics, Seoul from 2003 to 2007. He was a research professor of Korea University, Seoul from 2007 to 2008. He is currently a general manager of the Group of Quality Engineering, Monitor Laboratory, LG Electronics. His research interests include software architecture and software verification.

Jung Ho Bae received the BS and MS degrees in computer science and engineering from Pusan National University, Pusan, Korea in 2007. He is currently a PhD student in the Object Oriented System Laboratory at the Pusan national University. His research interests include MDT, Testing, design patterns and Android framework.