PAPER Special Section on Knowledge-Based Software Engineering

Deriving Framework Usages Based on Behavioral Models

Teruyoshi ZENMYO[†a)], **Takashi KOBAYASHI**[††], ***and*** **Motoshi SAEKI**[†], ***Members***

SUMMARY One of the critical issue in framework-based software development is a huge introduction cost caused by technical gap between developers and users of frameworks. This paper proposes a technique for deriving framework usages to implement a given requirements specification. By using the derived usages, the users can use the frameworks without understanding the framework in detail. Requirements specifications which describe definite behavioral requirements cannot be related to frameworks in as-is since the frameworks do not have definite control structure so that the users can customize them to suit given requirements specifications. To cope with this issue, a new technique based on satisfiability problems (SAT) is employed to derive the control structures of the framework model. In the proposed technique, requirements specifications and frameworks are modeled based on Labeled Transition Systems (LTSs) with branch conditions represented by predicates. Truth assignments of the branch conditions in the framework models are not given initially for representing the customizable control structure. The derivation of truth assignments of the branch conditions is regarded as the SAT by assuming relations between termination states of the requirements specification model and ones of the framework model. This derivation technique is incorporated into a technique we have proposed previously for relating actions of requirements specifications to ones of frameworks. Furthermore, this paper discuss a case study of typical use cases in e-commerce systems.

key words: *framework, labeled transition system, branch condition, satisfiability problem*

1. Introduction

In current software development, frameworks are being necessary for fast delivery and high quality. The framework can be considered as semi-complete software and provides extensible implementations which are common in a target domain. Therefore, by using the framework, final application software can be developed by implementing application-specific parts.

Software development processes using frameworks are different from conventional ones. The framework is completed into final software by customizing extensible points called hot spots. The hot spots are provided in various ways (e.g. hook methods, configuration files) and vary according to frameworks. Developers have to understand the way to customize the framework to suit their requirements.

To provide the extensibility, the frameworks are designed by expert developers and have complex designs in which high-level design and programming techniques such as design patterns and/or meta-programming may be used. On the other hand, the users of the frameworks are not limited to experts. For novice developers who are not skilled enough to design core parts of applications, adapting the framework is promising and may be crucial. However, it is not easy for the novice developers to understand the framework design and the way to extend the framework. Therefore, an introduction cost becomes a huge obstacle in framework-based development. For efficient framework-based software development, a technique to bridge the gap between the developers and the users is desired.

This paper proposes a technique to derive framework usages for implementing a given requirements specification. The proposed technique uses behavioral models of requirements specifications and frameworks as inputs and derives the usages by relating the requirements specification model to the framework model.

In our previous work, we have proposed a technique which relates actions of requirements specifications to ones of frameworks automatically [1]. The requirements specifications and the frameworks are modeled in Labeled Transition Systems (LTSs) and they are related based on observational equivalence [2] to assure equivalence of execution sequences. The usages of framework are identified based on the relations, and then, the usages are suggested to users in the form of skeleton codes.

The behavior of software systems is not characterized by only sequential execution but also branch conditions should be argued. Branch conditions are essential in the behavior since the branch conditions determine the behavior of system depending on their values. However, the branch conditions are not considered in our previous work and therefore, inappropriate relations could be also suggested to the users. Figure 1 illustrates the inappropriate relation. In Fig. 1, the requirement is not implementable with the framework although the framework has an action sequence "e1", "e2" which is equivalent to the requirement. The requirement specifies an action "e2" has to be executed if P is false. On the other hand, the framework shown in Fig. 1 executes the action "e2" only if P is true. In our previous work, the action "e2" of the requirements specification is related to one of the framework and the relation shown in Fig 1 is judged appropriate since the branch conditions are not argued.

To cope with this issue, this paper proposes a new technique for deriving consistent truth assignments of the branch

Manuscript received July 4, 2009.
Manuscript revised October 19, 2009.

[†]The authors are with the Department of Computer Science, Tokyo Institute of Technology, Tokyo, 152–8550 Japan
[††]The author is with the Department of Information Engineering, Graduate School of Information Science, Nagoya University, Nagoya-shi, 469–8601 Japan
a) E-mail: zenmyo@se.cs.titech.ac.jp

DOI: 10.1587/transinf.E93.D.733

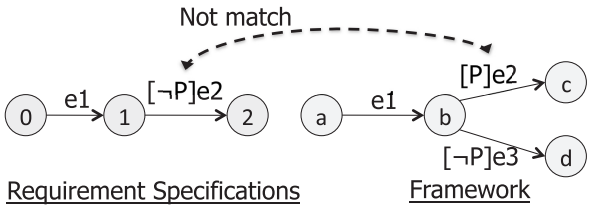

Fig. 1 Inappropriate relation.

conditions of frameworks. The derivation of the truth assignments is regarded as finding a solution of satisfiability problems (SAT) by assuming relations between termination states of requirements specification models and framework models. The new technique can be incorporated in our previous technique for relating actions, and therefore, the usage can be derived in case requirements specifications include branches.

The rest of the paper is organized as follows. In the next section, we clarify the requirements to support for framework-based software development. Section 3 overviews our project and presents the approach. Section 4 describes the proposed technique with an illustrative example. The proposed technique is evaluated in Sect. 5. Section 6 and 7 are for related work and for concluding remarks respectively.

2. Motivation and Issues

2.1 Desired Support

Frameworks provide control structures which are common in a target domain. Therefore, we consider the behavior is essential for relating requirements specifications to frameworks and this paper focuses on behavioral aspect.

Figure 2 illustrates how frameworks are used to implement requirements specifications. The framework shown in Fig. 2 executes a hook method (`hook`) according to requests. The hook method is associated with the requests in a configuration file. The developers have to implement application-specific actions with the hook methods and configure the control structure of the framework by editing the configuration file. To realize the requirements specification correctly, the actions in the requirements specification have to be executed appropriately according to situations. In Fig. 2, for instance, application-specific actions (save data and return message) are implemented by overriding the hook method. In addition, the `ActionImpl1` class is associated with registration requests by the configuration file. As just described, frameworks become complete software by implementing the application-specific actions and configuring control structure.

This example indicates below two points have to be considered for identifying the framework usages.

- Relations between the actions of requirements specification and the hot spots of frameworks.
- Configurations of frameworks in which the actions are preformed correctly according to situations.

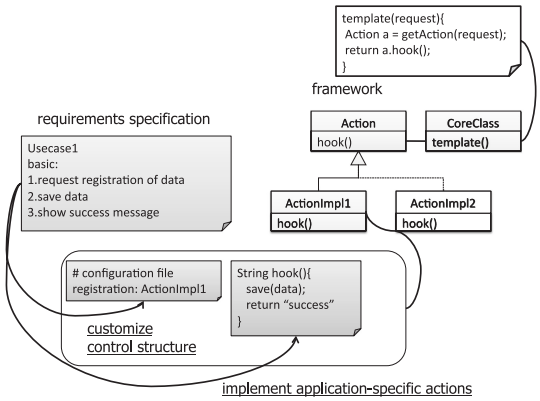

Fig. 2 Usage of frameworks.

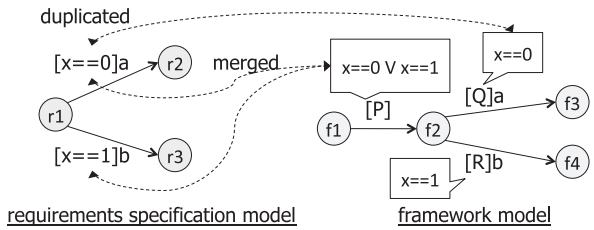

Fig. 3 Merge and duplication of branch conditions.

In the framework-based software development, developers have to identify such framework usages. However, finding the usages is not easy due to complexity of the frameworks. Therefore, we consider automated support is needed so that the developers can find the appropriate relations of actions and the configurations efficiently.

2.2 Issues in Achieving Automated Support

The actions of requirements specifications have to be related to ones of framework with ensuring sequential equivalence. To this end, we have proposed a technique based on process algebra [1]. In our previous work, behavior of requirements specifications and frameworks are model in Labeled Transition Systems (LTSs) and the LTSs are related based on observational equivalence [2].

On the other hand, configurations of frameworks also have to be identified so that the actions are performed in appropriate situations. However, the situations are not considered in our previous work, and therefore, the configurations of frameworks could not be identified. In addition, inappropriate relations of actions (e.g. Fig. 1) could be suggested to users.

To deal with this issue on situations, branch conditions have to be argued. The branch conditions are evaluated as Boolean values depending on situations. The values of the branch conditions define the situations where each action is performed. Therefore, the branch conditions of frameworks have to be customized to suit the requirements specification.

Figure 3 illustrates an appropriate relation of branch

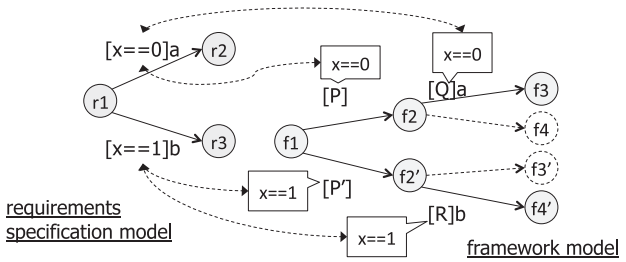

Fig. 4 Another possible relation in case the number of choices is customizable.

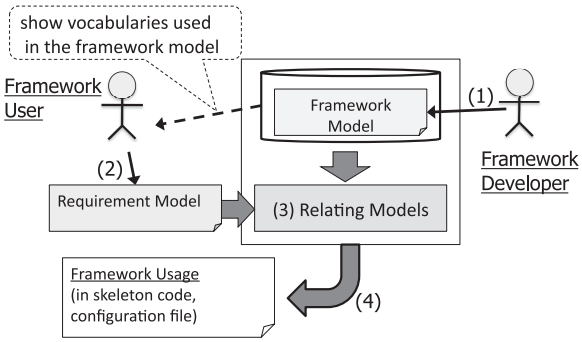

Fig. 5 The goal of the project.

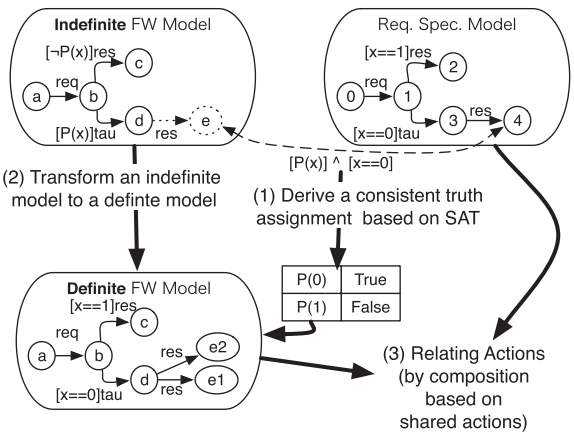

Fig. 6 Overview of the approach.

conditions. The requirements specification model specifies that two actions "a" and "b" have to be performed when x is 0 and 1 respectively. The framework model can perform these actions after the transition from "f1" to "f2" has occurred. Therefore, the branch conditions [x == 0] (from r1 to r2) and [x == 1] (from r1 to r3) in the requirements specification model are merged into [x == 0∨x == 1] and related to the condition [P] of the transition from "f1" to "f2" in the framework model. Furthermore, the branch condition [x == 0] is duplicated in [Q] in the framework model in addition to [P]. Such mergence and duplication have to be taken into account in identification of framework configurations.

The duplication could change the appearance order of branch conditions, and therefore, the process algebraic approaches based on the appearance order cannot deal with the branch conditions. For instance, [x == 0] is evaluated twice in the framework model shown in Fig. 3. By contrast, it appears once in the requirements specification model in Fig. 3. To cope with these issues, a distinctive technique is needed to find the appropriate configurations of frameworks.

Furthermore, the number of choices may be customizable in addition to the branch conditions. For instance, web applications frameworks (e.g. Struts [3]) can output a variety of web pages (choices) according to user's input. Making relations between requirements specifications and frameworks becomes more complicated due to such kind of branches. For instance, as shown in Fig. 4, another relation is possible in case that the number of choices of the branch at node f1 is customizable. In Fig. 4, there are two choices at the f1 (to f2 and to f2') and the branch conditions of the requirements specification [x == 0] and [x == 1] can be related [P] and [P'], respectively.

3. Proposed Support for Framework-Based Software Development

3.1 Goal

Figure 5 shows the goal of our projects. We aim to realize a tool which takes a framework model and a requirements specification model as inputs and derives framework usages for implementing the input requirements specification. The frameworks are modeled by framework developers. On the other hand, the requirements specifications are modeled by framework users.

The development using the tool is performed as follows.

1. Framework developers describe the model of the framework and register the model to the repository included in the tool.
2. Framework users describe requirements specification models. In this phase, the tool shows a vocabulary used in the framework models to the users for establishing the correspondence on atomic actions.
3. The tool relates the input requirements specification model to the registered framework models.
4. The usages of the frameworks can be identified based on the relations. The tool output the usages in form of skeleton codes for instance.

We have developed a prototype [1] which targets requirements specifications without branches. This paper presents an extension of the function relating requirements specifications to frameworks (step 3) for dealing with requirements specification including branches.

3.2 Approach

Figure 6 shows the overview of the proposed technique. Frameworks and requirements specifications are modeled based on LTSs where the branch conditions are attached as

predicates. The input framework model is called an indefinite framework model since the truth assignments of branch conditions are not defined and may have branches where the number of choices can be customizable.

The indefinite framework model is transformed into a definite framework model based on consistency of the branch conditions. The truth assignments and the control structure of the definite model are derived by finding a solution of a satisfiability problem (SAT) on branch conditions. If the solution can be found, we can obtain the definite model which has truth assignments consistent with requirements specification.

In the next step, the actions of the requirements specification model are related to ones of the definite framework model based on equivalence of action sequence. The actions are related based on observational equivalence as well as our previous work. In this paper, we use composition based on shared actions [4] to this end. By the composition based on shared actions, the definite framework model and the requirements model can compose a model which represents the relations, and then, the usages can be identified based on the composed model.

3.2.1 Deriving Truth Assignments of Branch Conditions

The derivations of truth assignments for branch conditions are regarded as finding a solution of SAT by assuming relations between termination states of a requirements specification model and a framework model.

In this paper, we use flows of processing from user's input to system's output as a unit of requirements. The termination states of requirements specification models are states after the output events occur. For instances, the requirements specification model shown in Fig. 6 has two termination states (2 and 4) where "req" and "res" are an input and an output event respectively. Similarly, termination states of a framework model are states when the framework finishes its processing (e.g. c and e in Fig. 6).

Branch conditions that exist on a path to a termination state have to be true to reach the termination state. For instance, to reach the state 4 of the requirements specification model shown in Fig. 6, the branch condition [*x==0*] has to be true. In other words, a termination state represents a combination of branch conditions which exist on a path to the termination state.

In order for the relation between the termination states of the requirements specification model and ones of the framework model to be valid, a consistent truth assignment has to exist. All of the branch conditions on the paths incoming to the termination states have to be true under the consistent truth assignment. By assuming the relations between the termination states, such truth assignment can be regarded as a solution of the SAT on the branch conditions of which relations are assumed. If the solution exists, a consistent truth assignment for branch conditions can be derived as the solution. Otherwise, the assumed relations between the termination states are not appropriate.

In case that the number of choices is customizable, branches with such kind of choices are transformed to the branches with a fixed number of choices based on a given requirements specification. In the indefinite framework model, branches where the number of choices is customizable are modeled distinctly by attaching a type *switch*. The switch typed branch is transformed to a branch with a finite number of choices based on the given requirements specifications. The requirements specification includes a finite number of action sequences. Therefore, by using the given requirements specification, the number of choices of the switch typed branch can be bounded.

3.2.2 Relating Actions and Checking Sequential Equivalence

In this paper, a model representing the relations between requirements specifications and framework is composed of the requirements specification model and the framework model based on shared actions [4]. Although an algorithm for relating the actions has been developed in our previous work, we employ the composition based on shared actions as a generalized method to deal with behavioral models including branches. Once the composed model is obtained, an existence of deadlock states is checked for ensuring the composed model represents appropriate relations. An existence of deadlock states which have no outgoing transitions in the composed model means that the sequences of the shared actions differ between the requirements specification model and the definite framework model.

4. Deriving Framework Usages

This section describes the proposed technique with an illustrative example. In the illustrative example, the requirement specification (use case description) shown in Fig. 7 is implemented with Apache Struts framework [3]. Figure 7 represents New-order use case in JPetStore [5] which is an on-line shopping web application.

Use case: New Order
Basic Sequence:
1 A user inputs order information.
2 The system checks whether all of mandatory information is input.
3 The system checks whether shipping address is requested to change.
4 If change of the shipping address is not requested, show the order confirmation page.
Alternative Sequences:
3a If the mandatory information isn't input, the system shows the order input page again.
4a If change of the shipping address is requested, show the address input page.

Fig. 7 New-order use case description.

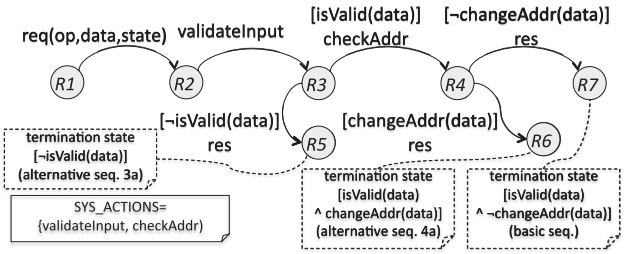

Fig. 8 A model of a use case description.

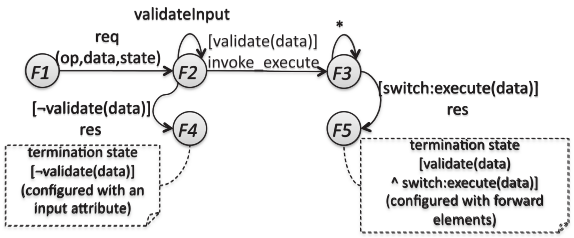

Fig. 9 A framework model.

4.1 Modeling Behavior

4.1.1 Modeling Requirements Specifications

The requirements specification shown in Fig. 7 has two alternative sequences, which are for invalid input (3a) and for changing shipping address (4a). There are two branch conditions, which are whether all of mandatory information is input and whether change of the shipping address is requested. These branch conditions relate to alternative sequences 3a and 4a, respectively.

Figure 8 shows a LTS which models the New-order use case. Branch conditions are modeled as predicates. In Fig. 8, "req" and "res" represent in an input and an output event respectively. We assume the system receives three values with input events (req). They are requested operation (op), input data (data) and a state identifier (state). The requested operations are instructions given by users. For instance, URLs correspond to the requested operations in web applications. The input data is sent to the system together with the input events. The state identifier indicates the state of the system when the input event occurs.

In the example shown in Fig. 8, the truth assignments of the branch conditions are determined by user's input (e.g. whether all of the mandatory information is included). Therefore, the branch conditions are modeled as predicates which have an argument corresponding to input data (data).

The New-order use case has below three termination states.

- The order input page is displayed in case of invalid inputs (R5 in case of $\neg isValid(data)$).
- The address input page is displayed in case of valid inputs with a change address request (R6 in case of $isValid(data) \wedge changeAddr(data)$).
- The confirmation page is displayed in case of valid inputs without a change address request (R7 in case of $isValid(data) \wedge \neg changeAddr(data)$).

Additionally, actions performed by the system are explicitly declared in the model. In the model shown in Fig. 8, "`SYS_ACTIONS`" declares that two actions (*validateInput* and *checkAddr*) are performed by the system.

4.1.2 Modeling Frameworks

Struts is one of the most popular frameworks for web applications. Although Struts provides a variety of hook methods, for brevity, this paper focuses on two of them, which are `ActionForm.validate` method for input validation and `Action.execute` method for application specific processing.

An indefinite model of Struts is shown in Fig. 9. In Fig.9, "req" and "res" represent in an input and an output event respectively. The branch conditions are modeled with predicates and their truth assignments are not defined. The method executions are modeled as loop actions based on the approach used in our previous work [1]. By modeling as loop actions, hook methods in which developers can implement any code and method implementing functions which are not mandatory in requirements specifications can be neglected. A hook method in which developers can implement any code (the `execute` method) is modeled as a transition labeled "$*$" [1]. The model expresses following framework behavior.

1. Struts receives an input from users ($F1 \xrightarrow{req} F2$). Then, depending on configurations, input validation is processed by executing the `validate` method ($F2 \xrightarrow{validateInput} F2$)
2. If the `validate` method returns `false`, the Struts sends a web page to retry the input ($F2 \xrightarrow{res} F4$), which is configured with `input` attribute of a configuration file. Otherwise, the `execute` method is invoked ($F2 \xrightarrow{invoke\_execute} F3$) and application specific processing is executed ($F3 \xrightarrow{*} F3$).
3. A next web page is sent to the user ($F3 \xrightarrow{res} F5$) according to a result of the `execute` method ($[switch : execute(data)]$). The next pages are configured with `forward` elements of the configuration file.

The number of choices from the state $F3$ is not defined. Therefore, related branch conditions cannot be defined completely. In the model shown in Fig. 9, this kind of branches is modeled with a *switch* type. The switch typed branch conditions are tentative expressions and are to be transformed to branch conditions without the *switch*. The transformation is explained later.

There are two termination states in the model shown in Fig. 9, which are $F4$ and $F5$. The state $F5$ relates to the switch typed branch condition, therefore, the state is tentative and is to be transformed as well as the switch typed branch condition.

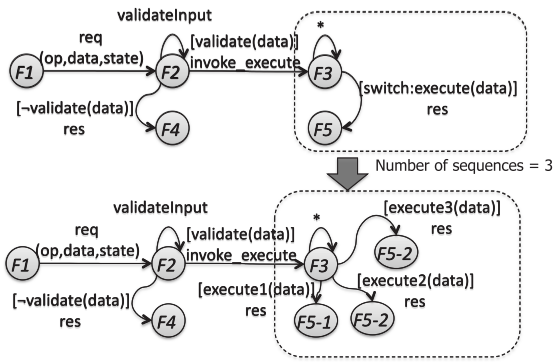

Fig. 10 Transforming a switch typed branch condition.

4.2 Deriving Truth Assignments of Branch Conditions Based on SAT

4.2.1 Transforming Tentative Expressions

The indefinite framework model can include tentative expressions, the switch typed branch conditions and the related states. The tentative expressions are transformed to regular expressions based on the given use case descriptions. For instance, the requirements specification shown in Fig. 7 has three action sequences (one basic sequence and two alternatives). Therefore, each branch can have at most three choices in the indefinite framework model. By using the maximum number of choices, the framework model shown in Fig. 9 can be transformed as shown in Fig. 10.

In Fig. 10, the switch typed branch condition [*switch* : *execute*(*data*)] and the final state F5 are transformed into three untyped conditions ([*execute*1(*data*)], [*execute*2(*data*)], [*execute*3(*data*)]) and three final states (F5-1, F5-2, F5-3), respectively. The transformed three conditions are exclusive here. For instance, the branch condition [*execute1(data)*] is actually [*execute1*(*data*) ∧ ¬*execute2*(*data*) ∧ ¬*execute3*(*data*)]. For simplification, we omit expressions of the branch conditions in Fig. 10.

4.2.2 SAT for Deriving the Truth Assignments of Branch Conditions

For deriving the branch conditions consistent with the requirements specifications, subsets of the branch conditions which should be true simultaneously have to be identified. For instance, the [¬*isValid*(*data*)] and the [¬*changeAddr*(*data*)] in Fig. 8 are not required to be true simultaneously.

The termination states are useful to identify the subsets since the termination states represent the combinations of branch conditions which exist on a path to the termination state. The branch conditions on the path should be true simultaneously. For instance, [*isValid*(*data*) ∧ *changeAddr*(*data*)] should be true to reach a final state *R*6 in Fig. 8. If the branch conditions represented by a termination state of the requirements specification model and ones represented by a termination state of the framework model can be true simultaneously, the termination state of the requirements specifications model can be related to the termination state of the framework model consistently.

Table 1 Assumed relation (1).

Framework	F4	F5-1	F5-2	F5-3
Requirements Specification	R5	R6	R7	(N/A)

Table 2 A truth assignment in the requirements specification model.

form	d_0	d_1	d_2	d_3
isValid	True	True	False	False
changeAddr	True	False	True	False

Therefore, by assuming relations between the termination states, the consistency can be examined by checking whether there exists a truth assignment under which all the branch conditions represented by the termination states are true.

Assumptions of relations between the termination states correspond to configuration of control structures of the framework. Table 1 shows the assumption which corresponds to a configuration in which the order input page is set to the `input` attribute and the other pages (confirmation and address input) are set to the `forward` elements.

In Table 1, the termination state F4 of the framework model relates to the final state R5 of the requirements specification model. F4 and R5 represent conditions [¬*isValid*(*data*)] and [¬*validate*(*data*)], respectively. Therefore, a following formula has to be true for consistency of the relation between F4 and R5.

$$\forall data.(\neg isValid(data) \leftrightarrow \neg validate(data))$$

The requirements specification shown in Fig. 8 includes two predicate (*isValid*(*data*) and *changeAddr*(*data*)), therefore, the truth assignments of these predicates can be defined as shown in Table 2 where d_0, d_1, d_2, d_3 are sample of input data. Based on this definition, the above constraint can be interpreted as a following formula.

$$validate(d_0) \wedge validate(d_1) \wedge \neg validate(d_2) \wedge \neg validate(d_3)$$

Another two constraints have to be satisfied for consistency of the relation between F5-1 (F5-2) and R6 (R7). By interpreting these constraints in the same way, checking the three constraints can be regarded as SAT for a following formula,

$$\begin{aligned}
&validate(d_0) \wedge validate(d_1) \wedge \neg validate(d_2) \wedge \neg validate(d_3) \\
&\wedge (validate(d_0) \wedge E1(d_0)) \wedge \neg (validate(d_1) \wedge E1(d_1)) \\
&\wedge \neg (validate(d_2) \wedge E1(d_2)) \wedge \neg (validate(d_3) \wedge E1(d_3)) \\
&\wedge \neg (validate(d_0) \wedge E2(d_0)) \wedge (validate(d_1) \wedge E2(d_1)) \\
&\wedge \neg (validate(d_2) \wedge E2(d_2)) \wedge \neg (validate(d_3) \wedge E2(d_3))
\end{aligned}$$

where
$E1(d)$ is $execute1(d) \wedge \neg execute2(d) \wedge \neg execute3(d)$
$E2(d)$ is $\neg execute1(d) \wedge execute2(d) \wedge \neg execute3(d)$.
This SAT has a solution and we can derive the truth assignment shown in Table 3.

Table 3 A derived truth assignment.

form	d_0	d_1	d_2	d_3
validate	True	True	False	False
execute1	True	False	False	False
execute2	False	True	False	False
execute3	False	False	False	False

```
range DATA=0..3
set OUTPUT={input,regaddr,confirm}
set SYS_TASK={validateInput,checkAddr}
UC=(req[d:DATA]->validateInput->ISVALID[d]),
ISVALID[d:DATA]=
(when(d==0||d==1) checkAddr->CHECKADDR[d]
|when(d==2||d==3) res.input->UC),
CHECKADDR[d:DATA]=
(when(d==0) res.regaddr->UC
|when(d==1) res.confirm->UC).
```

Fig. 11 The requirements specification model in FSP.

```
range DATA
set OUTPUT
set SYS_TASK
STRUTS=(req[d:DATA]->VALIDATE[d]),
VALIDATE[d:DATA]=
(validateInput->VALIDATE[d]
| when(!validate) res.OUTPUT->STRUTS
| when(validate) invoke_execute->EXECUTE[d]),
EXECUTE[d:DATA]
=(SYS_TASK ->EXECUTE[d]
 |when(switch:execute) res.OUTPUT->STRUTS)
\{invoke_execute}.
```

Fig. 12 The framework model in FSP like expression.

4.3 Relating Actions and Checking Sequential Equivalence

To perform the composition based on shared actions, we have used the LTSA tool [4]. The LTSA uses LTSs described in FSP (Finite State Processes) [4] as inputs. The FSP is simple process algebra for describing LTSs.

Figure 11 shows a FSP expression of the requirements specification model shown in Fig. 8 †. The first three lines are variables and sets declarations. The "`range DATA=0..3`" expresses a domain of the argument of predicates (data). The numbers 0, 1, 2 and 3 represent d_0, d_1, d_2 and d_3, respectively. The remaining lines define the behavior. For example, the forth line expresses that the model transits to the state `ISVALID[d]` from the state `UC` if two actions (`req(d), validateInput`) is performed. The branch conditions are expressed as `when` clauses. Additionally, the initial state (R1 in Fig. 8) and the final states (R5, R6 and R7 in Fig. 8) are merged (to "`UC`" in Fig. 11) in the expression for simplifying analysis by the LTSA.

On the other hand, the framework model shown in Fig. 9 is expressed as Fig. 12 in FSP like expression ††. The indefinite framework model cannot be completely modeled

```
range DATA=0..3
set OUTPUT={input,regaddr,confirm}
set SYS_TASK={validateInput,checkAddr}
STRUTS=(req[d:DATA]->VALIDATE[d]),
VALIDATE[d:DATA]=(validateInput->VALIDATE[d]
|when(d==2||d==3) res.OUTPUT->STRUTS
|when(d==0||d==1) invoke_execute->EXECUTE[d]),
EXECUTE[d:DATA]=(SYS_TASK ->EXECUTE[d]
|when(d==0) res.OUTPUT->STRUTS
|when(d==1) res.OUTPUT->STRUTS)
\{invoke_execute}.
```

Fig. 13 A transformed framework model.

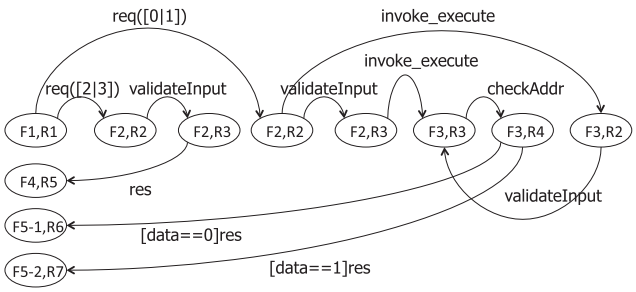

Fig. 14 A composed model.

in FSP since it includes a switch-typed branch conditions and the truth assignments of the branch conditions are not given. However, by using the truth assignment derived in Sect. 4.2, the indefinite framework model can be transformed to a definite framework model. The indefinite framework model is transformed as follows.

1. Extract switch typed branch conditions and related states.
2. Replace the branch conditions (`when` clauses) based on the derived truth assignments.
3. Substitute the "`range`" and the "`set`" definition of the requirements specification model for one of the framework model.

For instance, Fig. 13 shows a definite framework model based on the truth assignments shown in Table 3 (derived from the assumption shown in Table 1). In Fig. 13, the `when` clauses are replaced by concrete value based on Table 3.

The model which represents relations between the requirements specification and the framework can be created by the composition based on shared actions [4]. Figure 14 ††† shows the composed models of the requirements specification model and the framework model.

There is no dead lock state which has no outgoing tran-

†In this section, we omit the arguments for operation and state for brevity.

††"\" in Fig. 12 indicates that `invoke_execute` (a method invocation) is an inner action

†††The model shown in Fig. 14 has a different view from the LTS shown by LTSA for explanation purpose. Actually, the termination states are merged to the initial state to check equivalence of actions sequences by safety property check of LTSA.

sitions except the termination states in Fig. 14. This means that the behavior described in the requirements specification model can be simulated by the framework model shown in Fig. 13.

4.4 Identifying Framework Usages

The composed model shown in Fig. 14 includes a non-deterministic choice (state (F2, R2) may transit state (F3, R3) via state (F2, R3) or state (F3, R2)). This non-deterministic choice indicates multiple relations between `validateInput` of the requirements specification model and one of the framework model. Such non-deterministic choices appear due to there are two transitions in the definite framework model which can be related to the `validateInput` of the requirements specification model ($F2 \overset{validateInput}{\rightarrow} F2$ and $F3 \overset{*}{\rightarrow} F3$ in Fig. 10).

In our framework models, the actions performed by the framework are modeled as loop action. Only the inner action can transit states except input and output actions and can cause the multiple relations between the actions. Based on this property, we can distinguish these relations based on connected components composed of inner actions (`invoke_execute` in Fig. 14) and the action of which relations are distinguished. For instance, a sub-graph composed of state (F2, R2), (F3, R3), (F2, R3), (F3, R2) and transition `validateInput` and `invoke_execute` is a connected component. The connected components include two paths from source state (F2, R2) to sink state (F3, R3), and therefore, we can identify two relations between actions labeled with `validateInput`. These relations indicate that two alternatives to implement `validateInput` (use `validate` method or override `execute` method).

On the other hand, relations between branch conditions also have to be identified. We can identify the relations by comparing truth assignments in the requirements specification model and ones of the framework model. For instance, by comparing the truth assignments shown in Table 2 and Table 3, we can identify that the `isValid` corresponds to the `validate` and the `changeAddr` relates to the `execute1` and the `execute2`.

The usages obtained by the proposed technique can be summarized as follows.

- Relations between actions
  These indicate the hot spots with which the developer can implement actions of the requirements specifications.
- Relations between branch conditions
  These indicate the situations where the branch conditions of the framework have to be true.

4.5 Considerations for Efficient Derivation

4.5.1 Avoiding Redundant Usages

In case assumptions between termination states are exhaustively checked, redundant usages can be derived. Figure 15 illustrates such redundant assumptions in the illustrative example. In framework model shown in Fig 10, the states F5-1, F5-2 and F5-3 are originated from same state F5. Therefore, the assumption (1) and (2) in Fig. 15 are essentially same and deriving usages form both of these assumptions is redundant.

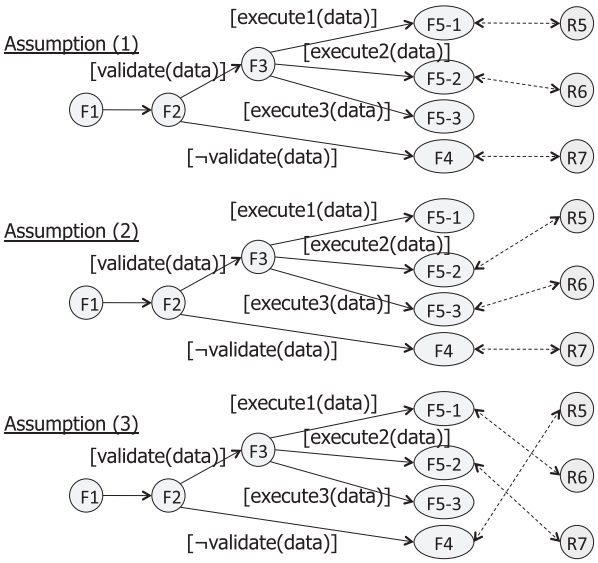

Fig. 15 Assumptions between termination states.

To avoid such redundant usages, the framework model to which terminations states of a requirements specification are attached can be regarded as a mixture of a sorted tree and an unsorted tree and the isomorphism is checked. By checking the isomorphism, the redundant assumptions (e.g. the assumption (2) when usages have been derived based on the assumption (1)) can be detected. By omitting such redundant assumptions, we can avoid to derive redundant usages. The tree is treated as the mixture to distinguish assumption (3) from the other assumptions in Fig. 15. The assumption (3) differs from the other assumptions since the termination state of the requirement specification model attached to the state F4 is different from the others. In the isomorphic checking of the mixture tree, edges relating to predicates originated from branch conditions associated with choices of which the number is customizable (e.g. (F3, F5-1)) are treated as in an unsorted tree and other edges (e.g. (F2, F3)) are treated as in a sorted tree.

The isomorphism check is performed by encoding the tree [6] and the codes of checked assumptions are recorded. Assumptions of which code has already been recorded are omitted, and therefore, the redundant usages are not derived. In addition, by omitting assumptions, computation costs can be reduced.

4.5.2 Limiting Branch Size

In case of multiple operations in requirements specifications, the number of action sequences becomes huge, and

then, switch-typed branch condition could be transformed to a huge number of choices. To alleviate such effects, the proposed technique adopt smaller one from the below alternatives as the size of maximum branch conditions used to transform the switch-typed branch conditions.

- The number of action sequences in the requirements specifications
- The domain size of argument variables of the switch-typed branch condition. In case of multiple arguments, the product of domain sizes is used.

4.5.3 Constraints on Framework Models

Frameworks configuration often have properties which have to be assured for customizations being appropriate. For instance, if the predicate $\neg validate(data)$ in Fig. 9 is always true, the framework becomes useless since all the input is processed as invalid input.

To cope with this issue, framework-specific constraints on branch condition can be attached to framework models. The constraints are additional conditions in the truth assignments derivations based on SAT. For instance, the constraint that the $\neg validate(data)$ is not a tautology can be attached to the framework model to avoid above-mentioned situation. By attaching the constraint, for instance, we can avoid deriving truth assignments where $\forall data.\neg validate(data)$ is true.

5. Evaluation

5.1 Setup

We have applied our approach for identifying the usages of Ruby on Rails [7] with act_as_authenticated plug-in to implement a requirements specification shown in Fig. 16. The arguments value used in Fig. 16 are explained in Table 4. The requirements specification is composed of two typical use cases which are common in e-commerce system like JPetStore [5] and osCommerce [8]. One is for login where users request authentication with id and password. The other is for proceeding checkout where only the authenticated user can proceed.

We have implemented a prototype which generates definite framework models in FSP [4]. To perform composition based on shared actions, the LTSA tool [4] has been used where the generated FSPs and a FSP describing the LTSs shown in Fig. 16 are input.

The framework model is summarized below although the detail cannot be described in this paper due to space limitation.

- The model includes below three types of Action Controllers. The controller which processes requests is selected based on the value of operation.

  An authentication controller provides functionalities for authentication and user registration. This controller cannot be customized any more.

  Secure controllers check the users are authenticated by the authentication controller before proceeding to request processing. We can define any size of the secure controllers.

  Normal controllers are standard controllers in Ruby on Rails (ones which do not check the authentication state). We can define any size of the normal controllers.

- The controller is normalized to avoid mixture of switch-typed branch conditions for avoiding redundancy (see Sect. 4.5.1) as shown Fig. 17.
- In each secure and normal controller, any size of action methods can be implemented, where developers can implement application-specific codes. The action method which processes requests is selected based on the value of operation.
- Each action method in secure and normal controllers can make a various types of response according to input data.
- The model has 19 predicates where 6 are switch typed.

In addition, three types of constraints on predicates are attached to the framework model to prevent derivations of

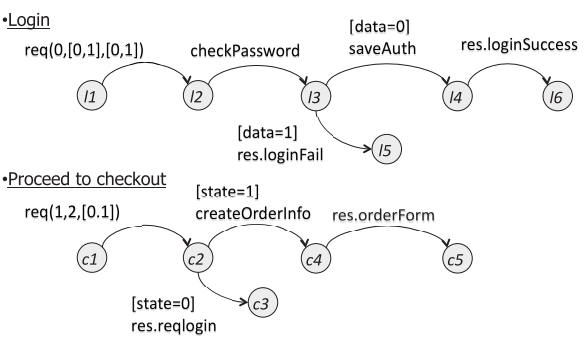

Fig. 16 Requirements specification.

Table 4 Descriptions of arguments in Fig. 16.

Operation (first argument)	
0	login request
1	proceed checkout request
Data (second argument)	
0	a correct pair of id and password
1	an incorrect pair of id and password
2	content of cart
State (third argument)	
0	the user is not authenticated
1	the user is authenticated

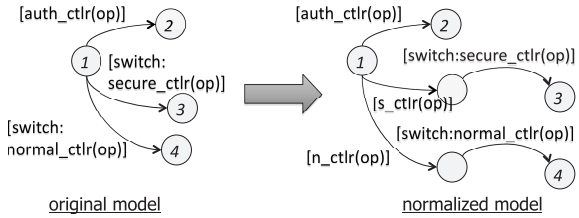

Fig. 17 Normalizing the framework model.

Table 5 Derived usages. (applied to each use case separately)

Login

No.	Usage
L_0	use the authentication controller
L_1	customize a normal controller

Proceed Checkout

No.	Usage
P_0	customize a secure controller
P_1	customize a normal controller

Table 6 Derivation statistics. (applied to each use case separately)

all relations between termination states	90
relations passed the redundancy check	58
definite framework models	5

inappropriate usages.

- **constraint attached to the predicate which evaluates users authentication states**
  This constraint declares that at least one state must treated as an unauthenticated state. If all states are treated as authenticated, the authentication function is meaningless. This constraint is to avoid the derivation of such usages.
- **constraint attached to the predicate which evaluates correctness of pairs of id and password**
  This constraint declares that at least one pair must make this predicate false for avoiding the authentication functionality becomes useless. If all pairs are treated as valid, the login function is meaningless. This constraint is to avoid the derivation of such usages.
- **Constraints attached to the predicates associating with the action methods, login and registration functionalities**
  These constraints declare that at most one value of operation make this predicate true for assuring one-to-one relations between operations and action methods. These constraint is to be compliant with the framework specification.

5.2 Results

Table 5 and Table 6 show derived usages and the statistics of the derivation when the proposed technique is applied to each use case separately[†]. The derived usages indicate that secure controllers and the authentication controller cannot be used for implementing the login and the proceed checkout use case, respectively. For the login use case, the secure controller is not appropriate since the state of the user is not checked in the login use case. On the other hand, the authentication controller cannot be customized any more and cannot perform functionalities except login and user registration. Therefore, the derived usage shown in Table 5 can be considered appropriate.

Table 7 and Table 8 show derived usages and the statistics of the derivation when the derivation of both use cases is performed simultaneously. Table 7 indicates three appropriate usages (A_0, A_3, A_4) can be derived with two false positives(A_1, A_2).

Table 7 Derived usages. (applied to both use cases simultaneously)

No.	Usage
A_0	the auth_ctr for login and a sec_ctr for proceed_co
A_1	the auth_ctr for login and a nor_ctr for proceed_co
A_2	a nor_ctr for login and a sec_ctr for proceed_co
A_3	a nor_ctr (two methods for login and for proceed_co)
A_4	two nor_ctrs for login and for proceed_co

(auth_ctr, sec_ctr and nor_ctr are abbreviated forms of the authentication controller, the secure controller and the normal controller respectively. proceed_co is an abbreviated form of the proceed checkout (use case).)

Table 8 Derivation statistics. (applied to both use cases simultaneously)

all relations between termination states	1974024
relations passed the redundancy check	4908
definite framework models	22

In case the authentication controller is used to achieve the login use case, the secure controller should be used for the proceed checkout use case. The normal controller is not appropriate in such case since the normal controller does not provide a function for checking whether users are authenticated by the authentication controller. The usage A_1 is inappropriate since it does not obey this rule. The usage A_2 is also inappropriate since the authentication function is not achieved consistently.

The appropriate usages have been identified in case that the proposed technique has been applied to each use case separately. Therefore, we consider the proposed technique is meaningful as a starting point for understanding framework usages. Framework users with insufficient knowledge can identify usages for implementing simple requirements by the proposed technique.

The result shown in Table 7 has a variation on the use of the normal controllers (A_3 uses two methods in a normal controller and A_4 uses two normal controllers). Although the inappropriate usages are included, the result shown in Table 7 indicates a different merit because the variation cannot be identified by simply combining the results of the separate applications (Table 5).

5.3 Discussion

The difference between the separate applications and the simultaneous application is whether relations between usages for each use cases are considered. The proposed technique transforms framework models based on the size of requirements specifications (described in Sects. 4.2.1 and 4.5.2). The relations between the termination states of the transformed framework model and the termination states of the requirements specification model are checked exhaustively. In the simultaneous application, the framework model is transformed into the model including two normal controllers and two secure controllers since the whole requirements specification contains two types of operation. The usages

[†]The statistics are same for both use cases.

derivation is performed taking the two normal controllers into account, and therefore, the variation shown in Table 7 can be identified. On the other hand, the framework model is transformed into the model which has one normal controller and one secure controller in the separate applications since each use case has only one type of operation. Therefore, the results of the separate applications cannot indicate, for instance, whether the normal controllers in L_1 and P_1 are the same or different.

The reason why the inappropriate usages have been derived is that the dependency among the framework functions is not considered. Information on the dependency (e.g. "In case that the secure controllers are used, the authentication controllers should be used.") are needed to judge the appropriateness of the usages. Such dependency information is just one perspective of frameworks design. We consider the judgement of usages appropriateness is easier than identifying frameworks usages from scratch since less information is needed, and therefore, we believe the proposed technique is meaningful even in the case of multiple use cases.

Although we consider the inappropriate usages derived in this example can be detected easily, avoiding such inappropriate usages is an important future work. We are investigating the refinement of the proposed technique based on the separate application approach. To avoid inappropriate usages, we consider to separate the analysis of relations among usages from the usage derivation for each use case.

Separating the usage derivation for each use case from checking and identifying relation among the usages has benefit for computation cost. The size of all possible relations between termination states increases exponentially with the number of terminations states of requirements specification model. The time to take to generate definite framework model increases in the same way. For instance, generating definite models has taken less than one second when the prototype applied to each use case separately[†]. On the other hand, it has taken about three and half minutes for generating definite models when applied to both use cases simultaneously. This indicates that the proposed technique is not suit to batch processing of multiple use cases.

The other future work is evaluation of usages. For instance, the usage A_0 is preferred rather than A_3 and A_4 since the authentication function can be reused with the usage A_0. To this end, we have proposed some metrics [1]. Evaluating effectiveness of these metrics in case of requirements specifications with branches and, if necessary, establishing new metrics are important future works.

6. Related Work

To the best of our knowledge, the proposed technique is a first attempt for relating requirements specifications to frameworks with considering branch conditions. Recent studies focus on support for maintenance of framework-based software [9], [10]. These works aim to maintain the framework-based software to adapt changes of the frameworks, and the relation to requirements specification is not considered. The other kind of framework-related study is on generation of framework models [11], [12]. We considered these work complementary to our work.

There are several researches on verification for software reuse [13]–[15]. Additionally, the techniques to select software components or architectures [16], [17] have been proposed. However, these works are not sufficient for our purpose because they did not consider the framework specific concepts such as the hot spots.

We consider applications to software product lines (SPL) are a part of the future work. Variability is important to implement a range of products of a product family. Framework technology is a promising approach to achieve the variability. In SPL community, there are various works on the analysis of the variability among products and product families [18]–[20]. On the other hand, we believe the proposed technique can contribute to implement products of a product family. By modeling behavior of the product family with LTSs, the usages to implement each product can be derived by the proposed technique.

7. Conclusion

This paper has proposed a technique to derive usages of frameworks based on satisfiability problems (SAT) and process algebra. The usages are derived based on consistency of branch conditions and equivalence of action sequences.

In our approach, requirements specifications are modeled based on Labeled Transition Systems (LTS). On the other hand, frameworks are modeled in indefinite form called indefinite framework models where branch conditions and control structures are not defined completely. The indefinite framework model can be transformed into a definite framework model by deriving truth assignments of the branch conditions consistent with the given requirements specification. The derivation is regarded as finding a solution of SAT on the branch conditions. If the definite framework model can be obtained, a model which represents relations between the requirements specification and the framework is created by composition based on shared actions. The usages can be identified based on the composed model.

We have prototyped a tool which has been applied to typical use cases in e-commerce systems to assess the proposed technique. A variety of appropriate usages have been derived by the proposed technique and some inappropriate usages which can be distinguished with simple information on dependency between framework functionalities also have been derived. Although we consider the proposed technique is effective since inappropriate usage can be easily distinguished, avoiding inappropriate usages is one of the future work. The other future work includes evaluation of usages and support from aspects other than behavior.

[†]A laptop PC with a 2.16 GHz dual core processor has been used.

References

[1] T. Zenmyo, T. Kobayashi, and M. Saeki, "Supporting application framework selection based on labeled transition systems," IEICE Trans. Inf. & Syst., vol.E89–D, no.4, pp.1378–1389, April 2006.
[2] R. Milner, Communication and Concurrency, Prentice Hall, 1989.
[3] "Apache struts." http://struts.apache.org/
[4] J. Magee and J. Kramer, Concurrency, State Models & Java Programs, Wiley, 1999.
[5] "Jpetstore." http://ibatis.apache.org/
[6] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974.
[7] "Ruby on rails." http://rubyonrails.org/
[8] http://www.oscommerce.com/
[9] B. Dagenais and M.P. Robillard, "Recommending adaptive changes for framework evolution," Proc. 30th International Conference on Software Engineering (ICSE '08), pp.481–490, 2008.
[10] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," Proc. 30th International Conference on Software Engineering (ICSE '08), pp.471–480, 2008.
[11] M. Antkiewicz, T.T. Bartolomei, and K. Czarnecki, "Automatic extraction of framework-specific models from framework-based application code," Proc. Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE' 07), pp.214–223, 2007.
[12] S. Thummalapenta and T. Xie, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web," Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08), pp.327–336, Sept. 2008.
[13] A. Betin-Can and T. Bultan, "Verifiable concurrent programming using concurrency controllers," Proc. 19th IEEE International Conference on Automated Software Engineering (ASE '04), pp.248–257, 2004.
[14] C. Blundell, K. Fisler, S. Krishnamurthi, and P.V. Hentenryck, "Parameterized interfaces for open system verification of product lines," Proc. 19th IEEE International Conference on Automated Software Engineering (ASE '04), pp.258–267, 2004.
[15] S.P. Reiss, "Specifying and checking component usage," Proc. Sixth International Symposium on Automated Analysis-Driven Debugging (AADEBUG '05), pp.13–22, 2005.
[16] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," Tech. Rep. CMU/SEI-98-TR-008, Software Engineering Institute, Carnegie Mellon University, 1998.
[17] V. Sai, X. Franch, and N.A.M. Maiden, "Driving component selection through actor-oriented models and use cases," Proc. 3rd International Conference on COTS-Based Software Systems (ICCBSS' 04), pp.63–73, 2004.
[18] D.S. Batory, "Feature models, grammars, and propositional formulas," Proc. 9th International Conference on Software Product Lines (SPLC' 05), pp.7–20, 2005.
[19] A. Metzger, K. Pohl, P. Heymans, P.Y. Schobbens, and G. Saval, "Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis," Proc. 15th International Requirements Engineering Conference (RE' 07), pp.243–253, 2007.
[20] D. Fischbein, S. Uchitel, and V. Braberman, "A foundation for behavioural conformance in software product line architectures," Proc. ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA' 06), pp.39–48, 2006.

Teruyoshi Zenmyo received a BS and an MS in computer science from Tokyo Institute of Technology in 2002 and 2004 respectively. In 2004, he joined Corporate Research and Development Center of Toshiba Corporation. He is currently a Ph. D student in computer science at Tokyo Institute of Technology. His research interests include software design method, software development automation and distributed systems. He is a member of IPSJ.

Takashi Kobayashi received B.Eng., M.Eng., and Dr.Eng. degrees in computer science from Tokyo Institute of Technology in 1997, 1999, and 2004, respectively. He is currently an associate professor of computer science at Nagoya University. His research interests include software patterns and architecture, software development method, software configuration management, Web-services compositions, workflow, multimedia information retrieval, and data mining. He is a member of IPSJ, JSSST, DBSJ, and ACM.

Motoshi Saeki received a B.Eng. degree in electrical and electronic engineering, and M.Eng. and Dr.Eng. degrees in computer science from Tokyo Institute of Technology, in 1978, 1980, and 1983, respectively. He is currently a professor of computer science at Tokyo Institute of Technology. His research interests include requirements engineering, software design methods, software process modeling, and computer supported cooperative work (CSCW).