PAPER *Special Section on Knowledge-Based Software Engineering*

Supporting Application Framework Selection Based on Labeled Transition Systems

Teruyoshi ZENMYO†*, ***Member***, **Takashi KOBAYASHI**††, ***Nonmember***, ***and*** **Motoshi SAEKI**†a), ***Member***

SUMMARY Framework technology is one of the promising approaches to reuse effectively software and its key issues are 1) to select the suitable frameworks for the software requirements specification, and 2) to fill the suitable hot spots with application-specific codes (customization). This paper proposes a new technique for automated support of the above issues by using labeled transition systems (LTSs) together with metrics technique. We model the behavior of the frameworks and the system behavior specified in the requirements specification by using two LTSs respectively. By establishing bisimilar relationship between the two LTSs, we check whether the behavior of the framework can match to the requirements and explore how to fill its hot spots. This process is done by means of constructing a graph to extract the bisimilar relationships, and each path of the graph denotes one of the implementations of the requirements by the framework. We attach some measures to the LTS of the framework, such as the number of the hot spots to be filled and the number of the parameters to be set up when filling the corresponding hot spot. These measures are used to estimate developer's efforts in filling the hot spots for each implementation, i.e. for each path of the graph. The result of estimating the efforts guides the developers to select the implementation, and the structure of the application-specific codes to be filled in can be automatically generated from the selected implementation. Furthermore we discuss case studies in the area of Web application, where Struts and Turbine can be used.

key words: *application framework, labeled transition system*

1. Introduction

Framework technology [1] based on object-oriented concepts is one of the promising approaches to reuse effectively software and becomes very popular even for practitioners. We have got various kinds of practical frameworks nowadays, especially Jakarta Struts and Turbine [2] are well known and frequently used in the area of Web application development.

A framework can be considered as a incomplete source code, and has specific "blank" places, so called hot spots, which the developers fill in with application-specific codes, so that it becomes a complete implementation satisfying the requirements specification. We call this task of filling the code *customization*. Software development processes using frameworks are different from conventional ones; first of all, developers select a set of the frameworks suitable for implementing the requirements specification, and then they identify which hot spots in the selected frameworks should be used for the development because the frameworks usually have more than one hot spot. By filling application-specific codes in the identified hot spots, i.e. customizing the frameworks, the developers can get the final product.

To use effectively frameworks, the key tasks are 1) to select the suitable frameworks and 2) to identify the suitable hot spots and to fill them with application-specific codes.

1. Selecting the suitable frameworks: The developers should select the frameworks by which they can implement the requirements and they can do with least development efforts. To fill application-specific codes in the hot spot so that the framework can work well as a program, the developers should write the application-specific codes and set up parameters specific to the hot spot. These efforts to do greatly depend on the selected frameworks and their hot spots.
2. Identifying the suitable hot spots and fill them with application-specific codes: Since usually more than one hot spot are available to achieve the application-specific requirements, the developers should decide which hot spots should be actually used. They select the hot spots so that they can achieve the requirement with least development efforts. What code the hot spot should be filled with greatly depends on the hot spots.

The above tasks to use frameworks require of the developers much knowledge about a family of frameworks of a problem domain. Automated support for the tasks to select the frameworks and to fill the application-specific codes in their hot spots is necessary for the developers, even non-experts to the frameworks.

This paper proposes a new technique to support the above tasks by using computer, i.e. by using this technique, non-experts of the frameworks can be guided to select the suitable frameworks for the requirements specification and to decide which hot spots are filled in with what application specific codes.

We focus on the behavioral aspects of requirements specifications and of frameworks, and check whether the behavior of the framework can *match* to the system behavior specified with the requirements specification. If the behavior of the selected framework matches to the requirements, we consider that it can implement the requirements. We model the behavior of frameworks and the system behavior of the requirements specification by using two labeled transition systems (LTSs) respectively. By establishing bisimilar

Manuscript received June 30, 2005.
Manuscript revised October 12, 2005.

†The authors are with the Dept. of Computer Science, Tokyo Institute of Technology, Tokyo, 152–8552 Japan.
††The author is with the Global Scientific Information and Computing Center, Tokyo Institute of Technology, Tokyo, 152–8552 Japan.
*Presently, with Toshiba Corporation.
a) E-mail: saeki@se.cs.titech.ac.jp

DOI: 10.1093/ietisy/e89–d.4.1378

relationship between the two LTSs, we can check whether the behavior of the framework can match to the requirements and explore how to fill its hot spots. The difficulty in this technique is how to deal with the behavior of hot spots. In our approach, hot spot parts are also modeled as "internal actions" in the LTS, which can be matched to any system behavior of the requirements specification. We construct a graph to extract the bisimilar relationships and in the graph, each path denotes one of the implementations of the requirements by the framework.

Furthermore, we attach some measures to the LTS of the framework, such as the number of the hot spots to be filled and the number of the parameters to be set up when filling the corresponding hot spot. These measures are used to estimate developer's efforts in filling the hot spots for each implementation, i.e. each path of the graph. The result of estimating the efforts guides the developers to select the implementation, and the structure of the application-specific codes to be filled can be automatically generated from the selected implementation. If we can have a data base system which stores frameworks in an application area together with their measures, we can have a useful support tool for non-experts of the frameworks.

The rest of the paper is organized as follows. In the next section, we clarify the requirements to automated support for framework based software development. Section 3 addresses how to model the behavior of the requirements specification and of the framework by using LTS technique. The mapping from the requirements to the framework on LTSs is also defined so as to investigate the satisfiability of the frameworks to the requirements specification. In Sect. 4, we illustrate some metrics to estimate the development efforts to complete the source code by using the selected framework and the implementation method. Section 5 presents how to generate the skeleton of the application-specific codes to be filled with following the selected implementation method. In Sect. 6, we discuss case studies in the area of Web application, where Struts and Turbine can be used. Sections 7 and 8 are for related work and for concluding remarks respectively.

2. Requirements to Automated Support

2.1 Customizing Frameworks

Figure 1 illustrates the principles of customizing a framework. It shows the varieties of the relationships between the requirements specification and the framework SimpleFW. The requirements specification is written as a set of scenarios. The framework SimpleFW contains three hook methods, two of which, mA() and mC() are implemented. On the other hand, the hook method mB() is not implemented yet, i.e. it's an abstract method, and the developer fills it with the application-specific code which is defined in the subclass of SimpleFW, e.g. mB()-imp#1. The instance object of SimpleFW cannot be created because it includes the method mB() having no implementation, while we can have the instance of subclass mB()-imp#1 having the specific implementation of mB(). When the message "template()" is sent to the instance of mB()-imp#1, the implementation of mB() specified in mB()-imp#1 is adopted for the execution of the "template()". If the developer adds another subclass, say mB()-imp#2, he can get the different method "template()" whose behavior follows mB() of mB()-imp#2. It means that he can flexibly change the part of the method mB() in the "template()" by supplying the subclasses including the various implementations of mB(). This technique is called template method and mB() of "template" is a hot spot. To fill a hot spot with a specific code, we typically use the mechanism of overriding a method from a subclass.

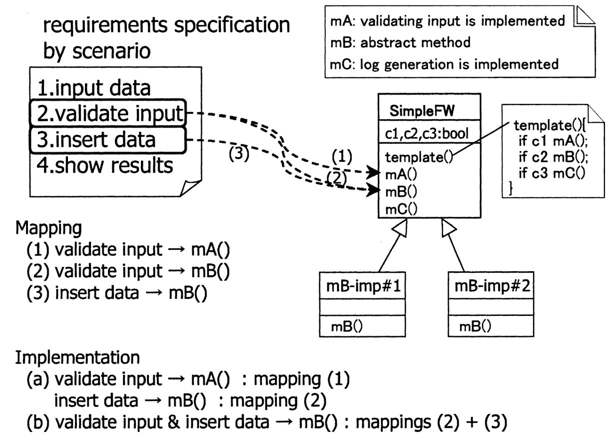


Fig. 1 Framework and customization.

The template method of SimpleFW specifies the selective sequential execution of mA(), mB() and mC(), and the boolean variables c1, c2 and c3 are flags to decide if these methods are executed or not. The developer should set their values before using this framework as well as implement the mB() by defining its subclass. It leads to varieties on how to use the hot spot, i.e. on several mappings of the requirements specification to the framework.

See Fig. 1 again and consider the mappings. As shown in the left side of the figure, the requirements specification is provided in the form of scenarios, i.e. an execution sequence of actions. If the developer likes to use the mA() of SimpleFW as it is to implement the action "validate input", all that he should do are to make the code for "insert data" as mB() and to add a subclass having the code. On the other hand, if he does not use the mA(), he is required to set the flag c1 to be false, to make both codes for "validate input" and for "insert data", and to override the hot spot mB() of SimpleFW with them. Thus he can have two alternatives of implementing the requirements specification as shown in the figure; one is a mapping from "validate input" to mA() and from "insert data" to mB() (alternative (a) in the figure), and another is the mapping both of them to mB() (alternative (b)). He should consider which implementation of them is suitable from the viewpoint of the development efforts that he makes. The former one seems to be more suitable than the latter, because "validate input" has already been implemented as mA() in SimpleFW and he should implement "insert data" only.

This example suggests to us the followings: 1) Although we have selected only one framework that is the most suitable, we can have several implementations, and 2) we can select the framework and its implementation by measuring developer's efforts. The variables c1, c2 and c3 in the figure are specific to the framework and that the developer is required to set up their values before using the framework. These kinds of parameters, more concretely their number and complexity to be initially set up, can affect the developer's efforts in using the framework.

Note that there are other significant factors to select a frameworks and an implementation such as performance, maintainability etc. These factors are closely related to non-functional requirements (NFRs) to the system being developed. In this paper, we focus on functional requirements (FRs) of the system and as a result narrow selection criteria to satisfiability with the FRs and to the measures of development efforts only. As for NFRs, we should develop the other types of the measures and they are one of the future work.

2.2 Requirements to Modeling Frameworks

As mentioned in the Sect. 1, software development processes based on framework technology includes 2 separate phases of selecting the suitable frameworks and filling the hot spots with application-specific codes.

To automate these phases, the essential point is a formal treatment of the mapping of the requirements specifications to the frameworks. Considering the illustration of the previous sub section, the following requirements to the formal model can be listed up.

1. The model should be the unified one in the sense that it can specify the behavior both of frameworks and of requirements specifications. Our consideration should be put on the issue how to model "hot spots" included in frameworks. The hot spots are the parts where the behavior cannot be specified or defined in advance.
2. By using a framework description and a requirements specification based on the model, their correspondence, i.e. mapping of the requirements specification to the hot spots should be automatically calculated. As illustrated in the previous sub section, this mapping cannot be uniquely defined because we can have several implementation methods. Our calculation technique should find out all of the possible mappings.
3. Some measurement techniques can be embedded into the formal model. To select the most suitable implementation method, we estimate the development efforts of the extracted mappings. Some measures used to estimate the efforts can be formally specified and defined.

In our approach, satisfying the above requirements, we adopt a technique of labeled transition systems (LTSs). We specify the behavior of the frameworks by using a LTS and do the requirements specification in the same way. By using extracting bisimilar relationships between the two LTS [3], we can get the mappings of the requirements specification to the hot spots. We improve the algorithm for detecting bisimilar relationships so that all of the mappings can be exhaustively detected. These mappings are presented with a directed graph, called *relational graph*, whose node is a mapping instance from an element of the requirements specification into that of the framework.

The behavior of a hot spot is considered as an "internal action" so that any behavior can match to it. We investigate and collect the framework-specific measures related to development efforts such as the number of set-up parameters and embed them to the LTS of the framework. The structural characteristics of a relational graph can be used to estimate the development efforts.

To establish automatically and exhaustively mappings, we assume that we can correspond the atomic actions appearing in the requirements specifications to the hook methods of the frameworks. More concretely, we use the same vocabularies to describe them in natural language. In the example of Fig. 1, it is assumed that the atomic action "validate input" appearing in the requirements specifications can easily correspond to the hook method mA() because the explanation of the mA() in the document of framework SimpleFW includes the same word. We call these words function names. Of course, we recognize that it is difficult for stakeholders or analysts to unify function names appearing in requirements specifications and in frameworks in practical situation. To solve this issue, we consider establishing domain-specific thesaurus is one of the promising approaches [4].

3. Relating Requirements Specifications to Frameworks

3.1 Labeled Transition Systems and Bisimilarity

LTS is commonly used to describe system behavior and defined as follows:

Definition 3.1 (Labeled Transition Systems (LTS)): A labeled transition system is a structure (S, L, T) where:

S is finite set of states.
L is a set of labels that denotes actions in the system or environment in which the system exists.
$T \subseteq (S \times L \times S)$ defines the labeled transitions between states.

If $(q, \alpha, q') \in T$, we write $q \xrightarrow{\alpha} q'$ and call q the source and q' the target of the transitions respectively. Intuitively, when the system is in the state q and the action α occurs, then it is changed to the state q'. If a series of transitions $q_i \xrightarrow{l_1} q_{i+1}, \xrightarrow{l_2} \dots \xrightarrow{l_n} q_{i+n}$, is possible in a LTS , we write this series $q_i \overset{l}{\Rightarrow} q_{i+n}$ where $l = l_1 l_2 \dots l_n$. □

In a LTS, the actions that are not observable from the outside of the system (i.e. environment) are modeled as an internal action and it is represented with the label τ.

Bisimulation is a binary relation on LTSs and represents that the two LTSs are indistinguishable from their environments. For instance, suppose there are two LTSs $p \overset{\tau\alpha\tau}{\Rightarrow} q$ and $p' \overset{\alpha}{\rightarrow} q'$. Since only action α is observable from the environment, these LTSs cannot be distinguished and we say that they are bisimilar. Figure 2 illustrates three LTSs that are bisimilar, and an only occurrence of action α is observed from the external environment in all of them.

Furthermore, in the similar way to [5], we attach state variables to states of a LTS and their values are used to specify complicated behavior with a simpler LTS. For simplicity, in this paper we omit this topic because the state variables do not contribute the framework selection and customization in the current version of our approach. Even if two states have the same label, we consider them as the different states in the case that their state values of two states are different.

3.2 Measuring Development Efforts

After deciding a mapping, we can estimate its development efforts by considering how many tasks the developer should do, e.g. how large he should construct an application-specific code, how many external components he should provide, how many parameters he should initially set up to use the framework, etc., as mentioned in the previous section. We express these factors numerically as measures of the development efforts. The hot spots of a framework are filled with application-specific codes through its hook methods [6], [7] as shown in Fig. 1. Therefore, we consider these measures should be put on the hook methods of the framework. We can summarize them as follows:

- The number of the hook methods that are not implemented yet.
  Some hook methods are implemented, but the others are not implemented in the framework. In the example of Fig. 1, the hook method mA() has been implemented in the framework SimpleFW, and on the other hand mB() has not been done. If the developer uses not mA() but mB(), he should implement it and make more development efforts. The number of the non-implemented hook methods has the effect on the development efforts.
- The number of configuration parameters about a hook method.
  As illustrated in the parameters c1, c2 and c3 in Fig. 1, the more configuration parameters to be set up, the more efforts the developer makes to use the hook method.
- The number of the external components that a hook method depends on.
  Some of implemented hook methods use external software components. For example, a hook method sends a message to the other class, and in this case it uses the external method defined in the other class. The number of external components that the hook method uses has an effect on the development efforts.

3.3 Modeling Framework Behavior by a LTS

Let *Func* be a set of function names specific to an application domain. In the case of Web application, we can have the set *Func* = {*input waiting, input validation, log generation,...*}. We consider that each state in a LTS models a set of the functions that are included in *Func*. The set *Func* plays an important role on establishing the mapping of requirements specifications into hot spots.

LTSs for modeling frameworks can be defined as follows and call this framework LTS.

Definition 3.2 (Framework LTS): A Framework LTS is a structure $(S, L, T, F, I, C_i, C_e)$ where S, L and T are states, labels and transitions of a LTS respectively and:

$F : S \rightarrow Func$
  $F(s) = n$ iff the function whose name is n is executed in the state s.
$I : S \rightarrow \{true, false\}$
  $I(s) = true$ iff the hook method represented by the state s has an implementation.
$C_i : S \rightarrow \mathbb{N}$
  $C_i(s) = n$ iff n is the number of configuration parameters to be set up at the state s.
$C_e : S \rightarrow \mathbb{N}$
  $C_e(s) = m$ iff m is the number of the external components used at the state s.

where $\mathbb{N}$ means a set of natural numbers. □

Some of hot spots do not have specific role in the sense that they are not implemented, and the developer can implement any code in such hot spots. Each of them is modeled as a state with a function $*$ (its meaning is a "don't care" function) and with iteration of an internal action.

For example, the model of SimpleFW is shown in Fig. 3. Label α represents the inputs from users and β represents the outputs from the system. f_0 is the state representing waiting users' input ($F(f_0)$ = *input waiting*). f_{mA}, f_{mB} and

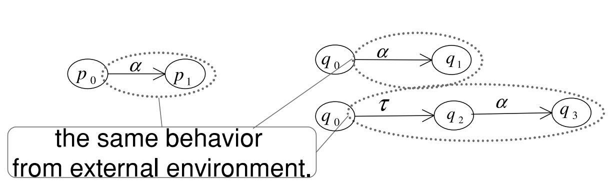


Fig. 2 Bisimulation.

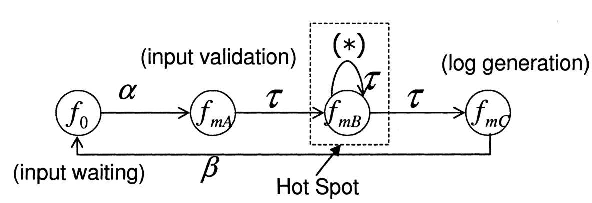


Fig. 3 A Model of SimpleFW.

f_{mC} are the states that represent corresponding hook methods are executed ($F(f_{mA})$ = *input validation*, $F(f_{mB}) = *$, $F(f_{mC})$ = *log generation*). State f_{mB} has the function $*$, and the iteration of an internal action since method mB() is an abstract method and does not have a specific role.

3.4 Modeling a Requirements Specification by a LTS

In the same way as the frameworks, we can model the behavior specified by a requirements specification as follows. We call this requirements specification LTS.

Definition 3.3 (Requirements specification LTS): A Requirements is a structure (S, L, T, F, IS, FS) where S, L and T are states, labels and transitions of a LTS respectively and:

$F : S \rightarrow Func$
  $F(s) = n$ iff n is the name of the function executed at the state s.

□

IS and FS are subsets of S and represent initial states and final ones respectively. We assume that requirements specifications are given as scenarios that include neither loops nor forks.

For example, the requirements specification shown in Fig. 1 is modeled as shown in Fig. 4. In this example, we have $F(r_{input})$ = *input waiting* and $F(r_{validate})$ = *input validation*. And the initial state and final one are r_{input} and r_{show} respectively.

3.5 Detecting the Mappings

As already mentioned, the mappings of the requirements specification LTS into the framework LTS should be detected exhaustively. The mapping makes each state of requirements specification LTS correspond to a state of framework LTS and it holds the following criteria:

- The function executed at the state of the requirements specification LTS is the same as that of the corresponding state of the framework LTS.
- For the mapping, the requirements specification LTS and the framework LTS are bisimilar.

The first criterion guarantees that the function of a state in requirements specification LTS is implementable or implemented in the hook method represented by the state of framework LTS by using the mapping. In the second criterion, both of the requirements specification LTS transition and the framework LTS one can occur with the same observable action.

These criteria are formally defined as follows.

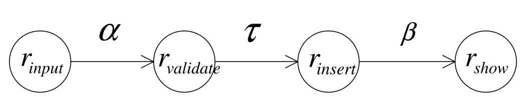


Fig. 4 The model of the example scenario.

Definition 3.4 (Criteria for mapping): Let the states r_i and $r_j \in$ requirements specification LTS be mapped into the states f_k and $f_l \in$ framework LTS respectively. Suppose the transitions $(r_i \xrightarrow{a} r_j) \in$ requirements specification LTS and $f_k \overset{\alpha}{\Rightarrow} f_l \in$ framework LTS. For these transitions, the constraints

$$(F(r_i) = F(f_k) \vee F(f_k) = *)$$
$$\wedge(F(r_j) = F(f_l) \vee F(f_l) = *)$$
$$\wedge(\alpha = \tau^* a \tau^*))$$

should hold. In this case, we write $(r_i \xrightarrow{a} r_j) \approx (f_k \overset{\alpha}{\Rightarrow} f_l)$.

□

Our algorithm is to detect the mappings so that the above constraints hold on them. The inputs of the algorithm are a requirements specification LTS and a framework LTS, and the output is a direct graph called relational graph. Each node in the graph is labeled with a pair of the states, e.g. (r, f) where r is mapped into f. Each path through the nodes is constructed following the possible transitions.

The overview of the algorithm can be illustrated as follows. Figure 5 shows the relational graph generated from the LTS shown in Fig. 3 and Fig. 4. Suppose that the state r_{input} has the same function name as f_0 ($F(r_{input}) = F(f_0)$ = *input waiting*), so we can get the pair (r_{input}, f_0) as a root of the rational graph. The pair means that r_{input} is mapped into f_0 in a mapping. Since the transition of the requirements specification LTS (Fig. 4) $r_{input} \xrightarrow{\alpha} r_{validate}$ and $f_0 \overset{\alpha}{\Rightarrow} f_{mA}$ (Fig. 3) are bisimilar, and $r_{input} \xrightarrow{\alpha} r_{validate}$ and $f_0 \overset{\alpha}{\Rightarrow} f_{mB}$ are also bisimilar, we can get the two nodes $(r_{validate}, f_{mA})$ and $(r_{validate}, f_{mB})$), both of which can be direct children of the root node. The meaning that we have the two nodes is that we can have the two implementations for the requirements specification; one is that the action "validate input" is implemented with the hook method mA() and another is that "validate input" is implemented with mB().

We can summarize the above procedure to construct a relational graph as follows:

Step 1 Setting up a root node: We make a pair $p(= (r, f))$ where r is one of the initial states of requirements specification LTS and f is one of the nodes that have the same function as r. The node p denotes a root node of a relational graph.

Step 2 Extending a graph: We make a pair $p'(= (r', f'))$ that satisfies $(r \xrightarrow{a} r') \approx (f \overset{\alpha}{\Rightarrow} f')$ and connect p to p' with an edge. The extension should be done for all pairs (r', f') that satisfy $(r \xrightarrow{a} r') \approx (f \overset{\alpha}{\Rightarrow} f')$

Step 3 Repeating step 2 until visiting the final state: If r' is a final state of requirements specification LTS the procedure ends. Otherwise, let $p := p'$ and return to Step 2.

Figure 6 illustrates the process to construct the relational graph of Fig. 3 and Fig. 4 following the above procedure. In the step 1, we have two pairs (r_{input}, f_0) and (r_{input}, f_{mB}) as root nodes of relational graphs, because f_0

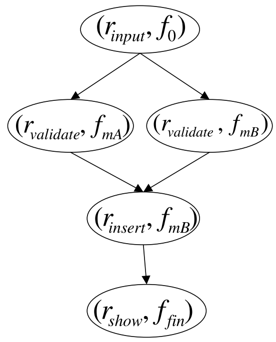


Fig. 5 A relational graph.

and f_{mB} have the same function name as the initial state r_{input} of the requirements. The two roots means that we will have two relational graphs, and from the subsequent steps, for each root node, we extend nodes and edges to construct a relational graph. On account of limited space, our example focuses on a root node (r_{input}, f_0) in Fig. 6. During executing step 2, the graph is incrementally extended by adding a new node and an edge. For example, in the first execution of step 2 in the figure, we find two pairs $(r_{validate}, f_{mA})$ and $(r_{validate}, f_{mB})$), and connect them to the root node, because $(r_{input} \overset{\alpha}{\to} r_{validate}) \approx (f_0 \overset{\alpha}{\Rightarrow} f_{mA})$ and $(r_{input} \overset{\alpha}{\to} r_{validate}) \approx (f_0 \overset{\alpha}{\Rightarrow} f_{mB})$ hold. In the last step, step 3, we find (r_{show}, f_0) as a leaf, because r_{show} is a final state and $(r_{insert} \overset{\beta}{\to} r_{show}) \approx (f_{mB} \overset{\beta}{\Rightarrow} f_0)$ holds. Finally we can get a relational graph as shown in Fig. 5.

The paths from an initial node of requirements specification LTS to a final node of requirements specification LTS denote the patterns of the implementations of the requirements specification by using the framework. We call these paths *realization paths*. As mentioned before, Fig. 5 includes two patterns of the implementations as follows:

- Using the "input validation" function provided by SimpleFW, the developer implements the code for "insert data" at the hot spot mB().
- The developer implements codes for both "validate input" and "insert data" at mB().

If we can have at least one realization path, we can implement the requirements specification by using the framework.

We can formally define our algorithm and it is shown in Appendix.

4. Measuring Mappings

As shown in the previous section, we select the frameworks that can have a realization path to implement the requirements specification. The next problem is how to select the suitable framework and the suitable mapping on it, because there can be more than one framework having realization paths and can be more than one realization path for the framework. In this section, we provide the technique to evaluate the frameworks and their mappings for selection. Our measures are defined based on using a relational graph.

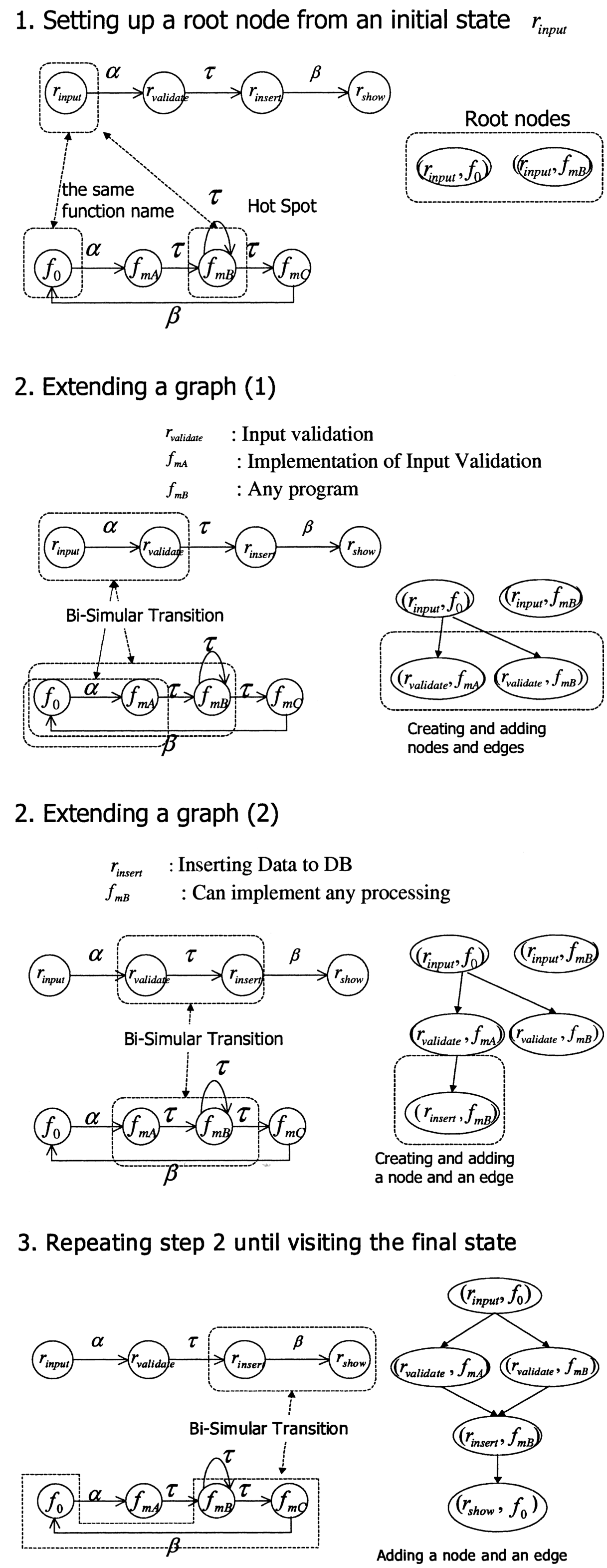


Fig. 6 Construction process.

Note that the aim of proposed metrics is the evaluation of mappings between behavior specified by a requirements specification and implemented in the framework.

4.1 Evaluating Availability and Realization Paths

The existence of realization paths can be a criterion whether a framework is available to realize the requirements specification since each realization path represents an implementation. Thus if the number of the realization paths is more than 0, the framework is available. However, since the number of the realization paths expresses the variations of the implementations, the more the number of the realization paths is, the more difficult it is for the developer to decide which implementation he should adopt. Thus we use the number of realization paths as a measurement and call it POM(Patterns Of Mappings).

$$POM = \text{the number of realization paths in a relational graph}$$

In our approach, a POM of value "1" is the best for above reason. The smaller the POM is, the better the framework is with respect to the difficulty in choosing one implementation from the candidates.

4.2 Estimating Development Efforts

The development efforts that the developer should make in the framework-based developments are categorized as follows:

- the total size of the code that the developer should develop. (WMI)
- the total number of parameters to be configured. (NCP)
- the total number of the mandatory external components. (CSC)

The total size of the application-specific codes that the developer should develop and fill the hot spot with can be estimated by counting the states of the requirements specification LTS that are not mapped to the implemented hook methods. In the example of Fig. 1, mB() is not implemented yet, i.e. $I(f_{mB}) = false$. If the developer adopts the realization path (r_{input}, f_0), $(r_{validate}, f_{mA})$, (r_{insert}, f_{mB}), (r_{show}, f_{fin}) in Fig. 3 (the left path of the figure), he should develop one application-specific code for "insert data" (r_{insert}). On the other hand, if he uses the right path of the figure, he should develop two codes; one is for "validate input" and another is for "insert data" because the path includes the two nodes $(r_{validate}, f_{mB})$ and (r_{insert}, f_{mB}), which are related to the non-implemented method mB(). It means the implementation denoted by the right path of the figure seems to require twice volumes of the code as the left path to fill the hot spot mB(). We can use this counting number for one of the estimations of development efforts, and write it as WMI (Weighted Methods to be Implemented). We can formally define WMI by using a relational graph and I in Definition 3.2.

The other two metrics to estimate the development efforts, the total number of the variables to be configured and the total number of the external components, can be calculated by using the functions C_i and C_e in Definition 3.2. We name these values NCP (Number of Configuration Parameters) and CSC (Costs for Setting up external Components). Note that C_e is defined on the states corresponding to the implemented methods, i.e. $C_e(f)$ is defined if and only if $I(f) = true$.

4.3 Discussions on Measuring Mappings

In this section, we discuss the limits on application of our approach.

Since our approach is based on LTS which models behavior of a requirements and a framework, it is inapplicable when a requirements specification and/or a framework cannot be represented by behavioral model. Moreover, our approach may produce less effects when most states of requirements map to hot spots of some frameworks because our approach cannot clarify differences between the frameworks.

However, we can deal with even non functional requirements and characteristic for a system if we can represent the non functional ones as behavioral aspects. For example, a consumption time and dependability of a system can be handled with simple extention for the transition of LTS. On the other hand, we cannot treat non functional ones which are irrelevant to behavioral aspects such as reusability, extensibility.

5. Tool for Automated Support

The prototype of the supporting tool based on our technique for framework selection and customization is developed. The tool includes the support for automatically generating the skeleton codes with which the hot spots are to be filled. As shown in Fig. 7, our tool contains the following three components:

1. Requirements Specification LTS Editor
   The tool users use this editor to input and edit the requirements specification with a LTS. To attach the unified function names to the requirements specification

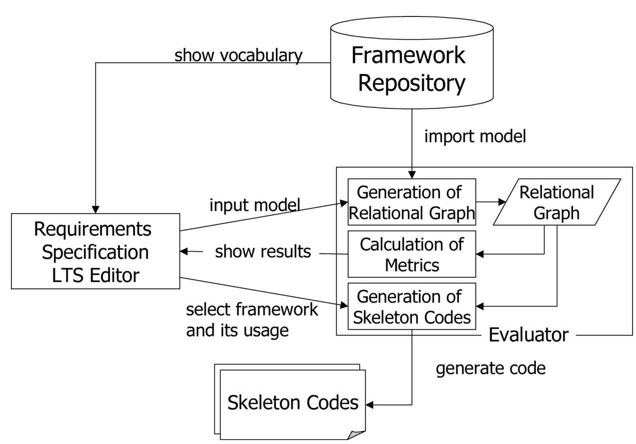


Fig. 7 Overview of the tool.

LTS for establishing the correspondence on atomic actions, the function names attached to the framework LTSs, which are stored in the framework repository are shown for the tool users. The results of measuring the frameworks and the mappings are also shown through this editor.

2. Framework Repository
The repository holds the existing framework LTSs in an application domain. The framework LTSs are retrieved so that they can hold the availability to the requirements specification LTS.
3. Evaluator
The evaluator has three roles; the first one is to evaluate the availability of the stored framework LTSs by generating their relational graphs and the second is to calculate the measures POM, WMI, NCS and CSC, mentioned in Sect. 4. After selecting the framework and the mapping, it generates the skeleton of the codes to be filled with, and that is the third role.

Figure 8 shows the screen shot of the LTS editor to input the requirements specification. The editor is a kind of diagram editor and is for inputting and editing LTSs as graphs. To attach to the LTS the attributes such as function names and state variables, as shown in Fig. 9 (a), the specific dialog appears when the user clicks the state in the LTS diagram. Figure 9 (b) shows how to input the function names by using a pop up menu. The entries of this menu are automatically generated from the names used in framework LTSs.

The result of measuring each realization path in the graph is displayed in the form of a table, as shown in Fig. 10. Each row represents the metrics values for a realization path, i.e. an implementation. In the figure, the path of the identification number 4 has the values 2, 3 and 0 as WMI, NCP, and CSC respectively, and they help the developer in selecting the implementation. In this case, he seems to select the path 4 because WMI is the lowest.

The skeletons of the codes are generated based on the information attached to the LTSs. Figure 11 shows how to generate the skeletons of the codes. For each node having no implementation method on the selected realization path, a class is generated to implement the method. In this example, two classes are generated to implement the requirements "validate input" and "insert data". In Fig. 11, the skeleton of the class for implementing "validate input" is generated by using a predefined template. The requirements specification LTS and the framework LTS have the attributes such as state, function name, method name and argument types (argType and return Type in the figure), and the slots of the template are filled with their values. For example, className and methodName slots in the figure are replaced with "SimpleFW" and "mB" respectively and as a result, we can have a subclass of SimpleFW having the method mB for implementing the function name "input validation". The develop will make a code in the part of "input validation" shown as a comment parenthesized with /* and */.

Currently our tool supports Java Skeleton code generation only. However, almost all language can be supported with simple modifications, given that the file requiered for customizing hot spots can be represented as template.

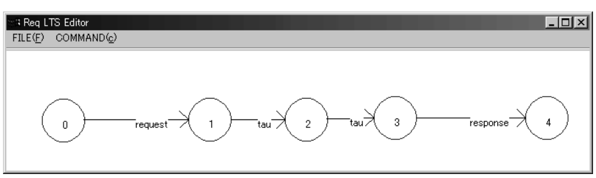
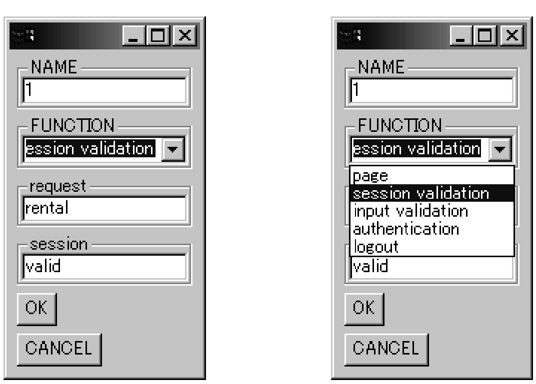


Fig. 8 Inputting the requirements specification.



(a) (b)

Fig. 9 Editing the states of LTSs.

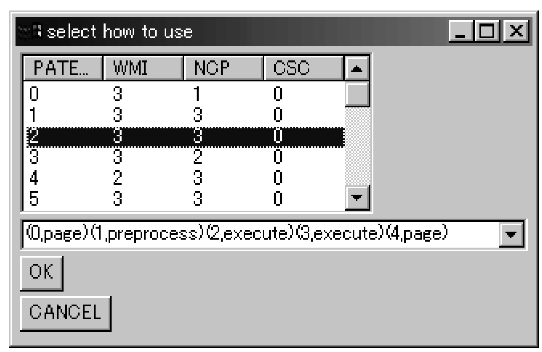


Fig. 10 Measurement results.

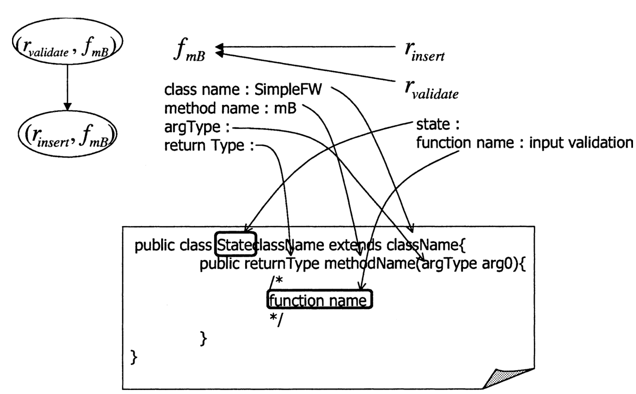


Fig. 11 Generating skeleton codes.

6. Case Studies

6.1 Examples and Results

We have applied our approach to the developments of library business support systems using web application frameworks in order to assess it. The users of the systems can perform library business such as borrowing books on Web pages. We picked up two type of systems for case studies:

System 1 Only administrators of the library can use the system, and the system validates sessions in each Web-page transition in order to assure that the user is authenticated as an administrator. The system supports the authentication of administrators, the procedures of adding new books to the library, of removing books from it, of borrowing books and of returning books.

System 2 Everyone can use the system and the system validates the inputs of users in order to check if all items of the input information are completely and correctly filled in. The system supports the procedures of adding new books to the library, of removing books from it, of borrowing books and of returning books.

We have evaluated two Web application frameworks, Struts and Turbine, that have been developed by Jakarta Project [2]. We have defined their LTSs as shown in Fig. 12 and Fig. 13 respectively, and the attached information to them is shown in Table 1. For instance, in Struts, the function name "input waiting" is assigned to state s_0 and the corresponding method has already been implemented ($I(s_0) = true$). Neither configuration parameters to be initially set up nor external components are necessary for the method ($C_i(s_0) = C_e(s_0) = 0$).

We have calculated the measures by using our supporting tool and the results are shown in Table 2. Note that we use the value of POM per scenario. It means how many implementations are possible to achieve a scenario included in the requirements specification, because the requirements specification is composed of scenarios.

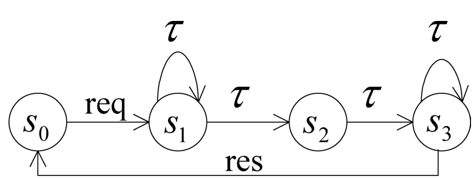


Fig. 12 The LTS model of Struts.

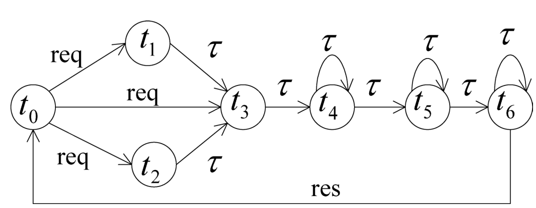


Fig. 13 The LTS model of Turbine.

6.2 Discussion

To assess the soundness of our measures, we have developed both System 1 and System 2. Each system has been developed twice by using the different frameworks. The first one and the second have been developed using Struts and Turbine respectively. Both system have been developed by a person who has 4 years of Java programming experience and 2 years of web application development experience. For generality of the data, Each implementation has been checked whether the system has full function indicated by requirements and whether the implementation is suitable by three developers who have from 3 to 10 years of Java programming experience.

Table 3 shows the results of the developments. LOC (Lines of Codes) expresses the total volume of the codes that were to be newly implemented by hand. By comparing WMI(min) with LOC, WMI is sound to estimate the volume of the codes to be implemented. Both of the results presented that Turbine was better than Struts in System 1, but on the contrary Struts was better in System 2, from the view of development efforts.

The identification of the hot spots is more difficult in case that the framework has many hot spots. POM of Turbine is higher than Struts in the two developments because Turbine has more hot spots that the developers can fill with any code. POM is sound to estimate the easiness of identify-

Table 1 Attached information of the LTSs.

STATE	F	I	C_i	C_e
s_0	input waiting	true	0	0
s_1	*	false	1	0
s_2	input validation	true	9	0
s_3	input waiting	false	3	0
t_0	input waiting	true	0	0
t_1	authentication	true	2	1
t_2	logout	true	2	0
t_3	session validation	true	3	0
t_4	*	false	0	0
t_5	*	false	0	0
t_6	*	false	0	0

Table 2 Calculating measures.

metrics	System 1		System 2	
	Struts	Turbine	Struts	Turbine
POM per scenario	3.87	14.95	5.32	10.19
WMI(min)	10	7	7	9
WMI(max)	13	13	11	11
NCP(min)	1	0	1	0
NCP(max)	19	5	51	0
CSC(min)	0	0	0	0
CSC(max)	0	1	0	0

Table 3 Actual development efforts.

	System 1		System 2	
	Struts	Turbine	Struts	Turbine
LOC	259	135	170	182

ing hot spots to be used. The value of NCP means the configuration parameters of Struts was more complicated than Turbine. This fact resulted from the property that Struts manages page-transitions by the configuration file "struts-config.xml", and it leads to the larger value of NCP (max) in Struts. This fact coincides with the result that in the development of System 1 using Struts the developer had to write larger code (259 LOC) than Turbine (135 LOC). On the other hand, in Turbine, the value of CSC is related to the authentication function that requires a relational database system, thus CSC (max) of System 1 was 1 and the same metrics of System2 was 0 because System2 has no autentication process. In this case study, the developer can use the relational database functions such as advanced search for large data through the interface to the relational database system. It led to the smaller code (135 LOC) than Struts (259 LOC) in the development of System 1. Moreover, if we don't need to use the relational database functions, we shouldn't use Turbine because Turbine cannot save developments cost for implementing funcitons excluding an autentication in this case study. In the development of System 2, we can get actually about same LOC. Considering learning costs of Turbine which is more difficult than Struts, we have to select Struts in the development of System 2.

To reduce development efforts to customize hot spots, some frameworks [8], [9] support Dependency Injection (DI) [10] and/or Weaving which is a key feature in Aspect oriented programming [11].

DI enables us to use new external components for customizing hot spots without any code modifications. All a developer need to do is to set the parameter in the configuration file only. In our approach, we can treat DI as one of configuration parameters. However, we need to modify definition and /or an evaluation scheme of NCP since development effort in the case of DI is much smaller.

We can treat Weaving as well as DI if the framework is designed with interceptors which are insert points for some aspects as hot spots. When we weave the framework with intersepter and some aspects, all we need to do is to set the parameter in the configration file only. However, we may need to consider how to describe specifications, since many aspects such as logging aspects which are very famous, may not be described in requirements. It is future work.

The complexity of the algorithm can be one of the critical issues when we apply our approach to much more complicated LTSs. Its theoretical and experimental analyses to check if the algorithm can be practically executed are necessary.

7. Related Work

We can find the several researches to support framework based software developments, especially focusing on the implementation of hot spots [12]–[14]. In them, the dependencies among hot spots are modeled or documented to deal with constraints on the hot spots, so that when the developer fill a hot spot, its implementation information can automatically propagate to the other relevant hot spots. However, they did not support the selection of framework or the identification of the hot spots, and the their objectives are different from ours. We have not dealt with dependencies among hot spots in a framework. As a future topic, we can add the measure on how complicated dependencies exist in the framework to estimate development efforts more accurately. In [15], the authors proposed the measures of software components for estimating adaptation efforts, by defining functional distances between the components and requirements specifications. This approach seems to be helpful to elaborate our measures to select a framework and an implementation.

For some popular frameworks (e.g. Jakarta Struts), the supporting tools have been developed and put into practice [16], [17]. However these tools concentrate on a specific framework and require that the users have knowledge on the target framework. On the other hands, our tool is based on the general property of framework (e.g. customizing hook methods). Only the knowledge on the target domain and on the LTS are required. And our approach is applicable to the wider application domain of frameworks.

The existing metrics of frameworks have been used for framework developers, not for the developers of application programs, i.e. users of frameworks [18], [19]. Although these metrics are useful to develop frameworks of high quality, by using them we cannot estimate the development efforts of the framework based software developments or whether the framework is available or not.

The techniques to select software components [20], [21] are not sufficient to select the frameworks because they did not consider the selection criteria specific to frameworks (e.g. whether all of required functions can be mapped to a framework or not). In particular, some of them did not focused on behavioral aspects of components, but static properties such as signatures of functions (data types of input and output data) [22], [23]. To deal with hot spots in a framework, we should consider its behavior.

8. Conclusion

In this paper, we have proposed the technique to support framework selection and customization based on LTS and metrics. Our technique allows the software developer to estimate framework-specific criteria (e.g. whether a certain framework is available or not) and supports the identification of hot spots.

In our approach, we focus on the formal mapping from the requirements specifications to the frameworks. To related them on LTS systems, we use the concept of bisimilarity. Requirements specifications and frameworks are modeled based on LTS, and several additional information is attached to these LTSs to define the metrics so as to estimate the development efforts. To establish and explore out the mappings, we design the algorithm to construct a directed graph whose path expresses an implementation by using the framework.

The tool has been developed based on the proposed technique. Our tool does not require the knowledge of a framework. Using it, the skeleton codes can be generated with the knowledge about LTSs and the target domain. A kind of database, called framework repository, plays an important role to the automated support.

The several researchers focus on supporting the implementation of hot spots [12]–[14]. Their contributions are complementary to our technique. The integration with them is one of the future works. The other future work is the treatment of proscribed behavior [5] to deal with the requirements specifications written with the style of misuse case [24].

References

[1] M.E. Fayad and D.C. Schmidt, "Object oriented application frameworks," Commun. ACM, vol.40, no.10, pp.32–38, 1997.

[2] Jakarta Project, http://jakarta.apache.org/

[3] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[4] J. Kato, M. Nagata, S. Yamamoto, M. Saeki, H. Kaiya, H. Horai, A. Ohnishi, S. Komiya, and K. Watahiki, "PAORE: Package oriented requirements elicitation," Proc. 10th Asia-Pacific Software Engineering Conference (APSEC2003), pp.17–26, 2003.

[5] S. Uchitel, J. Kramer, and J. Magee, "Behaviour model elaboration using partial labelled transition systems," 11th ACM SIGSOFT Syposium on the Foundations of Software Engineering (FSE03), pp.19–27, 2003.

[6] W. Pree, Design Patterns for Object-Oriented Software Development, Addison Wesley, 1995.

[7] G. Froehlich, H.J. Hoover, and P.G. Sorenson, "Choosing an object-oriented domain framework," ACM Comput. Surv., vol.32, no.1es, 2000.

[8] Seasar, http://www.seasar.org/

[9] Spring Framework, http://www.springframework.org/

[10] M. Fowler, "Inversion of control containers and the dependency injection pattern," http://www.martinfowler.com/articles/injection.html

[11] G. Kiczales, J. Irwin, J. Lamping, J.M. Loingtier, C.V. Lopes, C. Maeda, and A. Mendhekar, "Aspect-oriented programming," ACM Comput. Surv., vol.28, no.4es, p.154, 1996.

[12] T. Tourwé and T. Mens, "Automated support for framework-based software evolution," Proc. International Conference of Software Maintenance (ICSM03), 2003.

[13] A. Schappert, P. Sommerlad, and W. Pree, "Automated support for software development with frameworks," ACM SIGSOFT Symposium on Software Reusability, pp.123–127, 1995.

[14] R.E. Johnson, "Documenting framework using patterns," Proc. OOPSLA'92, pp.63–76, 1992.

[15] L.L. Jilani, J. Desharnais, M. Frappier, R. Mili, and A. Mili, "Retrieving software components that minimize adaptation effort," Proc. 12th IEEE International Conference on Automated Software Engineering (ASE'97), pp.255–313, 1997.

[16] Scioworks CAMINO, http://www.scioworks.com/scioworks_camino.html

[17] Struts Console, http://www.jamesholmes.com/struts/console/

[18] R. Kuroda, K. Ono, and Y. Fukazawa, "A metric for frameworks based on the deviation of user's understanding from the manufacturer's intention," Proc. 4th International Conference on Information Systems Modelling (ISM'01), pp.103–110, 2001.

[19] J. Bansiya, "Evaluating framework architecture structural stability," ACM Comput. Serv., vol.32, no. 1es, 2000.

[20] A. Lozano-Tello and A. Gomez-Perez, "Baremo: How to choose the appropriate software component using the analytic hierarchy process," Internatinal Conference on Software Engineering and Knowledge Engineering, 2002.

[21] C. Alves and A. Finkelstein, "Investigating conflicts in cots decision-making," Int. J. Software Engineering and Knowledge Engineering, vol.13, no.5, pp.473–493, 2003.

[22] B.H.C. Cheng and J.J. Jeng, "Reusing analogous components," IEEE Trans. Knowl. Data Eng., vol.9, no.2, pp.341–349, 1997.

[23] A.M. Zaremski and J.M. Wing, "Specification matching of software components," ACM Trans. Softw. Eng. Methodol., vol.6, no.4, pp.333–369, 1997.

[24] I. Alexander, "Intial industrial experinece of misuse cases," IEEE Joint International Requirements Engineering Conference (RE'02), pp.61–68, 2002.

Appendix: Algorithm for Detect Mappings

We start with the definitions of useful and auxiliary notation to define the algorithm.

Framework LTS

$LTS_f = (S_f, L_f, T_f, F_f, I, C_i, C_e)$

Requirements specification LTS

$LTS_r = (S_r, L_r, T_r, F_r)$

Initial States of requirements specification LTS

$S_{r_0} = \{s | s \in S_r \wedge \forall s' \in S_r, l \in L_r.(s', l, s) \notin T_r\}$

Final States of requirements specification LTS

$S_{r_{fin}} = \{s | s \in S_r \wedge \forall s' \in S_r, l \in L_r.(s, l, s') \notin T_r\}$

Relational Graph

$G = (V, E), V \subseteq S_r \times S_f, E \subseteq V \times V$

A set of series of transition of framework LTS

$$T_{f^*} = \{(f_i, l_0 l_1 \ldots l_n, f_{n+1}) \\ | \forall k \in \{0 \leq k \leq n\}.(f_{i+k}, l_k, f_{i+k+1}) \in T_f \\ \wedge \forall x, y \in \{0 \leq k \leq n\}.x \neq y \rightarrow f_x \neq f_y\}$$

The formal definition of the algorithm is provided as follows.

step 1 $P_{source} \leftarrow \{(r_0, f) | (r_0 \in S_{r_0}) \wedge (F(r_0) = F(f) \vee F(f) = *)\}, V \leftarrow P_{source}, E \leftarrow \emptyset$.

step 2 If $P_{source} = \emptyset$, then go to step 7.
Otherwise, for $\exists p \in P_{source}$, let $p_0 \leftarrow p, i \leftarrow 0, P_i \leftarrow \{p\}, P_{i+1} \leftarrow \emptyset$.

step 3 If $P_i = \emptyset$, then go to step 6. Otherwise, for $\exists p \in P_i$, let $p_p \leftarrow p$.

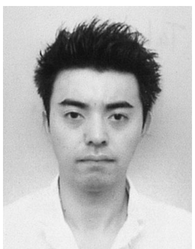
step 4 Let $P_{tmp} \leftarrow \{(r_l, f_m) | (r_p \stackrel{a}{\rightarrow} r_l \approx f_p \stackrel{\alpha}{\Rightarrow} f_m) \wedge (r_0, a, r_l) \in T_r \wedge (f_p, \alpha, f_m) \in T_{f^*}\} (where p_p = (r_p, f_p))$,
and let$P_{i+1} \leftarrow P_{i+1} \bigcup P_{tmp}, V \leftarrow V \bigcup P_{i+1}, E \leftarrow E \bigcup \{(p, p') | p' \in P_{tmp}\}$.

step 5 Let $P_i \leftarrow P_i - p_p$, and return to step 3.

step 6 If $P_{i+1} = \emptyset$, let $P_{source} \leftarrow P_{source} - p_0$, and return to step 2.
Otherwise, let $i \leftarrow i + 1, P_{i+1} \leftarrow \emptyset$, and return to step 3.

step 7 Let $G \leftarrow (V, E)$.

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



Takashi Kobayashi received a 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 assistant professor of Global Scientific Information and Computing Center, Tokyo Institute of Technology. His research interests include software patterns, software architecture and multimedia information retrieval and data mining. He is a member of IPSJ, JSSST, DBSJ and ACM.

Motoshi Saeki received a BS in electrical & electronic engineering, and an MS and a Ph.D. 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 interets include requirements engineering, software design methods, software process modeling, and computer supported cooperative work (CSCW).