October 2006

An Approach to Provisioning E-Commerce Applications with Commercial Components

Lei Wang
Bond University, Lei_Wang@bond.edu.au

Padmanabhan Krishnan
Bond University, Padmanabhan_Krishnan@bond.edu.au

An Approach to Provisioning E-Commerce Applications with Commercial Components

Lei Wang and Padmanabhan Krishnan
Centre for Software Assurance
Bond University
Gold Coast, Queensland 4229
Australia
Email: {lwang, pkrishna}@staff.bond.edu.au

Abstract— **Component-based development is a trend towards building e-commerce applications. However, commercial components are rarely used during the development. The reason is that existing approaches to selecting and composing components suffer from the problem that the components retrieved usually do not exactly fit with other components in the system being developed. While formal methods can be used to describe and check semantic characteristics to better match components, there are practical limitations which restrict their adoption.**

We have proposed a framework to support a semantic description and selection of components. We used Simple Component Interface Language (SCIL) to describe user requirements and pre-built components from the current component sources. Specifications in SCIL can be translated to a variety of models including those that have a formal basis.

In this paper, we preform a case study of searching commercial components for a generic e-commerce application. We specify the commercial components in SCIL and use two specific tools: jMocha and Alloy Analyser to identify the correct components that suit a particular task.

I. Introduction

The main advantage of Component-Based Development (CBD) is the reuse of existing components when building systems, so that time and effort can be saved while productivity can be enhanced. This makes CBD the ideal starting point for today's e-commerce applications.

The idealised CBD process assumes that the components selected are sufficiently close to the units identified when decomposing the system that is being developed, so that the component adaptation requires less efforts than the unit implementations. However, this is not always the case. People often cannot get the required components, or the components obtained need a lot of work to be adapted into the system. This additional development and maintenance cost [1] may trade off the benefits of CBD. Moreover, completed systems may malfunction due to incompatibilities of the components selected. All these potential problems indicate that component selection should be considered throughout the entire life cycle of CBD. Even in the very early stages of requirement analysis and architecture design, the requirement engineers and system architects must be aware of the availability of the components.

These problems are decided by the nature of CBD, which involves two separate development processes: develop components and use components. These two development processes normally do not have enough communication to each other. That is to say, there are a large variety of components and user requirements for components, but those components are not tailor made to the requirements. Thus in reality it is very rare that component users can get an exactly wanted component without any adaptation when integrating it into the system. The goal of our research is to increase the possibility of finding and retrieving components that meet user requirements, while minimising the component adaptation cost, since this cost is inevitable.

We have shown that collaboration between component users and developers based on formal methods can achieve our goal [2]. The collaboration is based on component developer's interest in assisting component users to find and retrieve required components. Recently we conducted an informal survey of seven developers from different organisations that develop components. It showed that most developers and their organisations are willing to help by customising their products based on user requirements. Component customisation is easier for component developers because they have direct access to the internal details of their products.

Tools are needed to facilitate the collaboration. Since component selection involves the activities of specifying components, specifying requirements, matching and retrieving components, and evaluating components, the tools need to aid all these activities. We have developed a web-based component selection prototype system, in which component users and developers are able to exchange information through a pre-defined language SCIL (Simple Component Interface Language). SCIL is an interface specification language used to capture component behaviours. We have built a framework using SCIL for checking behavioural compatibility for component selection [3].

In this paper we focus on applying this framework in finding the pre-built components for a generic e-commerce application. This application is from the student project offered by Software Engineering subject. In the original project plan, all the components identified are built by students in order to minimise cost and gain a learning experience. While in this paper, we will search and use those pre-built components from real-life component sources, such as ComponentSource [4], TopCoder [5], etc.

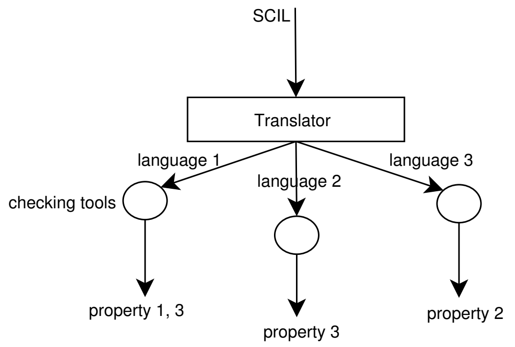


Fig. 1. Using SCIL as the Bridge to Checking Various Properties

The remainder of this paper is organised as follows: Section II briefly introduces background information of our approach to checking component behavioural compatibility, as well as an overview of SCIL; the implementation of our prototype selection system is described in Section III; Section IV gives the details of the case study; some problems are discussed in Section V; Section VI presents the related works to our approach; and the paper ends with a conclusion and potential future extensions.

II. Background

Checking component behavioural compatibility needs formal specifications of component interfaces. The tools use the formal specification to verify if the required behavioural properties hold when components are combined.

There exist a variety of modelling languages that rely on different formalism to support specifying different behavioural properties. If these languages with their tools are used as complements to each other, more behavioural properties can be checked. One also needs a bridge to access these pre-developed tools. Simple Component Interface Language (SCIL) is introduced to act as the bridge. For example, in the Figure 1, different languages can be used to check different properties. By translating SCIL to a variety of languages, it is possible for SCIL users to access all the properties that cannot be checked by a single tool. Furthermore it allows users who do not have a mathematical background to use such tools indirectly.

A typical SCIL file consists of these definitions: TYPE, SERVICE, PROTOCOL, SCENARIO and PROPERTY. As the aim of SCIL is to support lightweight formal descriptions for interfaces, it does not support complex data types. Basic class definitions can be specified using structure types. SERVICE corresponds to a high level method in languages such as Java. That is, in the actual component, a service may be implemented by a collection of methods. However, SERVICE defines the semantics of these methods by following the idea of Design by Contracts (DBC) [6]. PROTOCOL is used to capture component behaviour on interaction, viz., the call sequences to offered services. SCENARIO is defined to determine whether the selected component meets the user requirement. Scenarios in SCIL are similar to sequence diagrams or flows of events in use-case diagrams, viz., sequence of state transitions. Another way to check whether a component is compatible to the user environment is to check various properties of the composed system by the component and its targeting environment. PROPERTY can be formed by assertions or temporal logic expressions. The more detailed discussions on SCIL can be found in our previous paper [3].

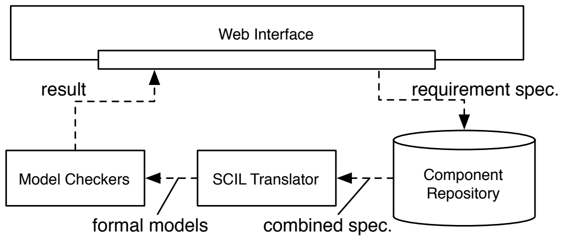


Fig. 2. The System Modules

Based on SCIL, the process of checking component behavioural compatibility can be summarised: first search components by keywords; then retrieve a component SCIL specification from the repository; combine component specification with the user requirement specification; and invoke the SCIL compiler with plug-ins to translate the combined specification to the target languages; finally send the translated specification to the relevant model checking tools to check whether the user-specified properties can be satisfied by the component.

III. Implementation

Our prototype system implementation consists of four modules (see Figure 2): the web interface, the component repository database, the SCIL translator, and the existing model checkers (currently only jMocha [7] and Alloy Analyser [8] are supported).

In our prototype system, the SCIL translator is implemented in Java. The web interface is implemented by JSP running on the Tomcat server. MySQL is used to build the sample component repository.

A. The SCIL Translator

The SCIL compiler is implemented using an open architecture. It permits different language modules to plug into the compiler. This enables the compiler to translate SCIL to the different languages without recompiling the compiler's source code. The SCIL parser is generated using SableCC [9]. The syntax checker uses the framework generated by SableCC. There is a common translation layer that recognises the different parts of the specification, and assigns a relevant grammar checking module to that part. As our framework supports a number of code generators, the compiler calls the plug-in manager to load a particular plug-in class (for example, rm.jar for Reactive Module [10]) that generates the translation to the particular modelling language.

We have implemented the plug-ins for Reactive Module (RM) [10] and Alloy [8]. Both of them can be used to describe transition systems, though Alloy needs to import the extra linear ordering module. For both tools, checking component compatibility is to check whether the composition of two components has any illegal behaviours. RM directly supports component composition by stating `component 1 || component 2`, while Alloy does not explicitly supports composing components, but is able to describe the overall transitions of the composition. Their formal details of checking theory can be found in their related papers.

A transition system in SCIL is decided by the rules of the services that can be described briefly as: `service s: a -> b` (if `a` is satisfied, then `b` is guaranteed). Being translated to RM, a rule is an update statement: `[] a -> b` (if `a` is satisfied, then `b` is executed). For Alloy, a rule is translated to a predicate: `s && a => b` (if method `s` is invoked and `a` is true, then `b` is true).

B. Component Repository

The component repository contains two databases. One is the user database that has all the information about the registered users of the repository, such as their usernames, passwords, contact details, etc.. The users are categorised as component users and component developers.

The other database stores the information of components, including the component name, its category, the keywords used to describe the component, SCIL specification, etc.. Each component is also connected with a component developer.

C. Web Interface

For component developers and component users, different user interfaces are implemented. Through the web interface, the system allows component developers to add components with SCIL specifications. Meanwhile the system allows component users to upload requirements and match components based on the requirement specifications.

Component users start to search for components by using keywords to reduce the number of candidate components. Then the SCIL compiler parses the candidate component specification and the requirement specification, and retrieves all the names of types, enumerate values, variables, services, etc. from two specifications. The users will be asked to map the names between the two specifications given the number of the names is the same.

If the requirement specification needs modifying, manual work is involved. The users are allowed to view candidate component specifications, and modify the requirement specification accordingly. The requirement specification is submitted to the repository, and combined with each candidate component specification. The combined specifications will be checked by the underlying tools. Finally the users are notified which components have the required behaviours.

The system decides which language (RM or Alloy) should be translated to by testing if there are scenarios defined. If so, the combined specification will be translated to RM. But if there are also properties defined in the specification, it will be translated to both RM and Alloy. If translating to Alloy, the scenario definition part is simply ignored.

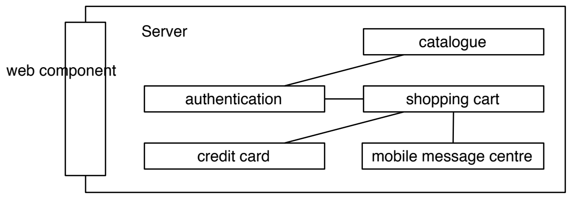


Fig. 3. The e-Commerce Application Architecture

The system will run the model checkers at background. Thus the model checking is transparent to the users. In order to achieve that, SLang scripts [7] for checking RM models are generated. For Alloy, the command interface of Alloy Analyser is invoked.

The system will display whether a candidate component has required behaviour by the results from the model checkers. For scenarios, the component is acceptable if jMocha can find the trace of the transition defined. For properties, both jMocha and Alloy Analyser should have the same checking results.

In next section, we will show how our system can be used in searching the pre-built components (normally they are commercial components) for the e-commerce application. We deliberately try to hide the formal details – the formal details can be found in papers associated with the tools. We aim to convince the reader, via the case study, that the framework we propose can indeed support formal methods without necessarily exposing the details.

IV. Case Study: Searching Components for the e-Commerce Application

The e-commerce application mainly consists of three components: authentication, catalogue, and shopping cart. Users interact with these three components through the web based user interface (web component). After the user checks out the shopping cart, if the products purchased are available, the application will invoke the credit card verifier component to verify the user's credit card; or if the products are not available, they will be registered in the mobile message centre, so that the user will be notified when the products become available. This architecture is highlighted in the Figure 3.

Since the operations of the other components are based on user authentication, we first search for the authentication component. Based on the retrieval of the authentication component, we can search for the catalogue and shopping cart components. The shopping cart component is also connected to the credit card verifier component and the mobile message centre (a subsystem developed by the students that can send messages to mobile devices).

Our component repository contains 132 components mainly taken from ComponentSource [4] and TopCoder [5]. The distribution of these components to different categories is displayed in the Table I. The category names are taken from the website of ComponentSource.

Category	Number of Components
3D Modelling	3
Addressing, Postcode and ZipCode	3
Business Rules	4
Calendar / Schedule	3
Charting and Graphing	7
Credit Card Authorisation	3
Data Validation	3
Database Reporting	3
Drawing	7
eCommerce	28
Email	3
Financial	4
Imaging	5
Maths and Stats	5
PDF	4
Reporting	4
Security and Administration	9
Spelling	7
Spreadsheet	7
Toolbar	7
User Interface	7
Zip and Compression	6

TABLE I
COMPONENT DISTRIBUTION IN THE SAMPLE REPOSITORY

A. *Searching for the Authentication Component*

We require the authentication component to have these basic functions:

- Users can log in and out to the system.
- New users are able to sign up to the system by filling in the necessary details and creating a unique username and password pair to allow login.
- System administrators can manage all the user profiles.

1) Types and Variables: We first define two types: `UserLogStatus` is the user status to the system and `UserRecStatus` is the status of the user profile.

```
UserLogStatus is {new, logged_in, logged_out,
  admin_logged_in};
UserRecStatus is {na, added, modified, managed};
```

Here `new` represents a new user; `logged_in` is used when the user has logged into the system, while `logged_out` is used when the user has logged out of the system; if the system administrator has logged into the system, the status will become `admin_logged_in`.

After a new user has signed up, a new profile is `added`; also the user profile can be `modified` after the user logs in; `managed` is used when the system administrator has updated (including add, delete, modify) the profiles of users. Finally `na` means not applicable or no changes to the user profiles.

The variables `uls` and `urs` are the instances of the types: `UserLogStatus` and `UserRecStatus`.

2) Specifying Requirements: The following SCIL description tells when users can log in and out to the system. If a user is new, signing up is needed.

```
uls = logged_out -> Login;
uls = logged_in -> Logout;
uls = admin_logged_in ->Logout;
uls = new -> Signup;
```

Here `Login`, `Logout`, and `Singup` are the services expected from the potential authentication component.

When a normal user logs in to the system, the user can only modify the individual profile. The administrator can manage all the user profiles.

```
uls = logged_in -> ChangeProfile;
uls = admin_logged_in -> ManageUsers;
```

Scenarios are defined to check whether the component can have expected behaviour. For example, a new user successfully signs up to the system, so that the user profile is added. Now the user is able to log in to the system, and then change the individual profile. After it is done, the user logs out. This scenario can be described by SCIL as this:

```
new_user_signup_change {
  uls = new;
  urs = added && uls = logged_in;
  urs = modified && uls = logged_in;
  urs = na && uls = logged_out;
}
```

We also want to check two properties if the authentication component is integrated into our application. The first, when a new user is being added or the user profile is being modified, the user cannot be logged out of the system:

```
always !(uls = logged_out &&
  (urs = added || urs = modified));
```

The second, only the administrator can manage all the user profiles:

```
always !(urs = managed && uls = logged_in);
```

3) Searching the Repository: We initially search using keywords `authentication` and `authenticate` and the system yields 19 components. This is because many components having security features that are described by these keywords. But most of them are irrelevant to our requirements, such as RSA BSAFE Cert-C component [4] and Rebex Secure FTP component [4].

Before combining the component specification and the requirement specification, we need to check the modelling consistency between the two specifications. This is because they are developed by different people. A typical example is that one may use `{admin, normal}` to mean administrator and normal user, while another may use a boolean type to mean the same (true means administrator and false means normal user). Other examples of inconsistent modelling include using the different numbers of states, states having different meanings, etc.. These problems cannot be solved simply by mapping names. The user has to view the component specification and modify the requirement specification according to the component specification, making sure they are in consistency. For example, if the component specification uses `{admin, normal}`, in the requirement specification the user has to change to the same type definition.

We think it not practical to review and modify a large number of specifications during the search. So we need to reduce the number of candidates by adding one more keyword `password`, then we get only 5 candidates (see Figure 4). .

4) Translating and Model Checking: The combined specifications are translated to RM and Alloy models. RM supports scenario checking, so scenarios can only be translated to RM. But the required properties can be translated and checked by both tools:

```
RM:
 predicate pred_p10_ is ~(uls = logged_out
  & (urs = added | urs = modified))
 judgment J_p10_ is
  authentication_requirement |= pred_p10_
 predicate pred_p11_ is ~(urs = managed
  & uls = logged_in)
 judgment J_p11_ is
  authentication_requirement |= pred_p11_

Alloy:
 assert as_p10_ {
  no s: State | (s.uls = logged_out
   && (s.urs = added || s.urs = modified)
 }
 assert as_p11_ {
  no s: State | s.urs = managed
   && s.uls = logged_in
 }
```

Both model checking tools advice that in our sample repository there are 3 components which have the expected behaviour (see Figure 4).

B. Searching for the Catalogue Component

The operations of the catalogue component can only be performed by the system administrator at the logged-in status. Thus the pre-condition of the services that the component provides is: `uls = admin_logged_in`. Other functions include adding/removing product groups, adding/removing products, users browsing catalogue. Due to space limitation, we do not put specification details here. The way to describe requirements is similar to the other components.

There are not many pre-built catalogue components available for use. When searching our repository by keywords `catalog catalogue`, we only get 2 candidates. After viewing their specifications, we find that one component only provides a catalogue presentation from the catalogue database while adding/removing records needs separate operations on the database. We cannot adjust our requirements for such a component. The other component satisfies our requirements, however, it offers more functions that we do not need, such as creating rules, calculating discount price, etc.. In order to use this component in our application, glue code is still needed [2].

C. Searching for the Shopping Cart Component

The shopping cart component requires users to log in to the system first. Meanwhile it connects to the credit card verifier and mobile message centre components that also should be specified in the requirement specification.

1) Types and Variables: Other than the type `UserLogStatus`, we need other types:

```
Quantity is (0..capacity);
ProductStatus is {not_available, available};
ReturnMsg is {na, succeed, fail, registered};
```

`Quantity` represents the number of the items that have already been put in the shopping cart, and `capacity` is the maximum number allowed. We give it a value of 5 as an example. `ProductStatus` denotes the status of a product, it could be in stock (`available`), or out of stock (`not_available`). `ReturnMsg` is the list of possible messages sent by the credit card verifier and mobile message centre component: `succeed` means the credit card is verified correct, `fail` means failing in verifying the credit card, `registered` is sent when the out-of-stock product has been registered in the mobile message centre, and `na` simply means no messages or not applicable.

The variables `q`, `ps`, and `rm` are the instances of the types: `Quantity`, `ProductStatus`, and `ReturnMsg` respectively.

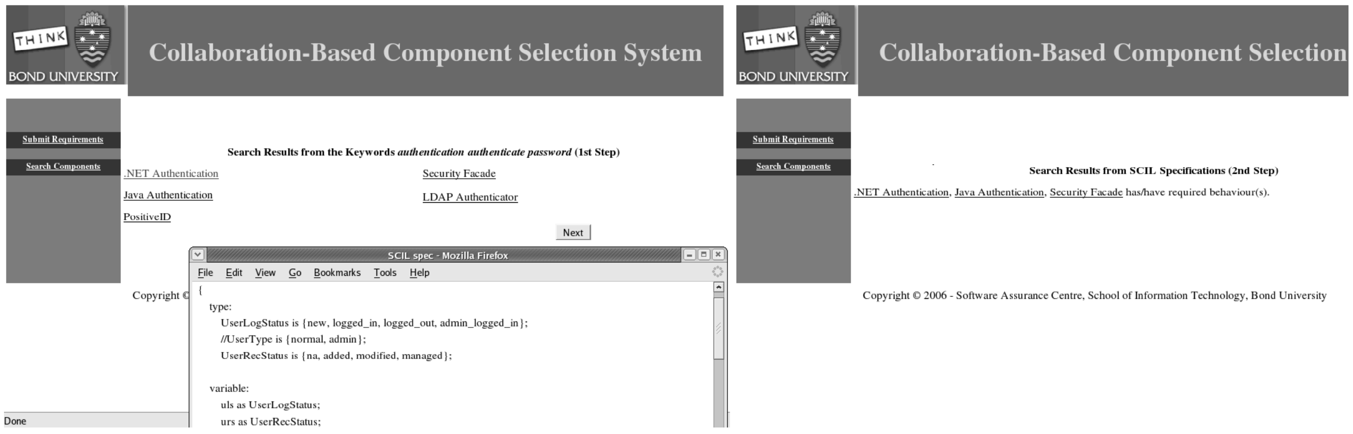
2) Specifying Requirements: We only allow normal users to add/remove products or checkout after they have been logged in:

```
us = logged_in -> AddItem || RemoveItem
  || Clear || Checkout;
```

where `AddItem`, `RemoveItem` will add or remove a product item in the shopping cart. `Clear` will empty the cart while `Checkout` is called after the user is ready to pay. They are required services from the shopping cart component.

This time the user environment includes the authentication, the catalogue, the credit card verifier, and the mobile message centre component. We import the `UserLogStatus` type as the interface of the authentication component. Since browsing catalogue (not key behaviour to this application) does not need to be modelled, and catalogue operations are only allowed by the system administrator, there is no interface between the catalogue component and the shopping cart component. In order to describe the interface for the shopping cart to access the other two components, we need to specify the services that these two components provide. Thus we need to define a new combined component interface in this requirement specification:

```
component CC_MC connects shopping_cart {
  ...
  CC_Verify {
    input: rm
    output: rm
    rule:
```


THINK
BOND UNIVERSITY
Collaboration-Based Component Selection System
Submit Requirements
Search Components
Next
SCIL spec - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Done
THINK
BOND UNIVERSITY
Collaboration-Based Component Selection
Submit Requirements
Search Components

Search Results from the Keywords *authentication authenticate password* (1st Step)

.NET Authentication | Security Facade
Java Authentication | LDAP Authenticator
PositiveID

Copyright ©

```
{
  type:
    UserLogStatus is {new, logged_in, logged_out, admin_logged_in};
    //UserType is {normal, admin};
    UserRecStatus is {na, added, modified, managed};

  variable:
    uls as UserLogStatus;
    urs as UserRecStatus;
```

Search Results from SCIL Specifications (2nd Step)

.NET Authentication, Java Authentication, Security Facade has/have required behaviour(s).

Copyright © 2006 - Software Assurance Centre, School of Information Technology, Bond University

Fig. 4. Searching for Authentication Component

```
      rm = na -> rm = succeed;
      rm = na -> rm = fail;
  }

  MC_Register {
    input: rm
    output: rm
    rule:
      rm = na -> rm = registered;
  }
}
```

Here CC_Verify is the service provided by the credit card verifier component. The service will check the validity of credit cards, and send message indicating operation success or failure. MC_Register is provided by the mobile message centre component to notify its caller that the product information has been registered.

We specify two scenarios about how the shopping cart component is expected to perform. The first scenario shows: the user first puts an item in the shopping cart, then adds another two items. But the user probably has made some mistakes, so the shopping cart is emptied. Finally the user puts one product item in the cart, and then checks out by credit card. Since the product is available, the credit card is verified. After the transaction is finished, the cart is emptied.

```
successful_story {
  q = 0;
  q = 1;
  q = 2;
  q = 3;
  q = 0;
  q = 1 && ps = available;
  rm = succeed && q = 0;
}
```

The second scenario shows that the user adds one item and then checks out. But the product is not available at the moment, so the mobile message centre registers the product.

```
need_notification {
  q = 0;
  q = 1 && ps = not_available;
  rm = registered;
}
```

One basic property of the joint system is to ensure the number of the items put in the shopping cart can never be less than zero, or exceed the maximum number allowed, which is 5 in our case:

```
always !(q < 0 || q > 5);
```

3) Translating and Model Checking: The scenarios can only be translated to RM monitors, while properties can be translated to both RM and Alloy. Here is the judgement on the scenario successful_story that needs to be checked:

```
module matching_successful_story_ is
 webshop || successful_story predicate
pred_successful_story_ is
 (final_successful_story_ = true)
judgment J_successful_story_ is
 matching_successful_story_ |= pred_successful_story_
```

4) Results: The first search by keywords shopping cart ecommerce has found 5 candidates, but none of them can be passed by both jMocha and Alloy Analyser. This is because no pre-built shopping cart components that have an interface to our mobile message centre. In order to use those pre-built components without modification, we need to change our requirements by removing the message centre component, or write a wrapper for the selected component. We can also ask the component developer to customise the component by adding such an interface. The shopping cart component .netCART [4] supports unlimited number of items. This would remove the constraint that the number of items should be within the range. Another candidate CartWIZ [4] is actually a combination of the catalogue and shopping cart components. It provides much richer functions than we expected. In order to use such a component in our application, glue code that can consume the extra functions is needed [2].

V. Discussion

The major difficulty of our approach we found in this case study is how to ensure the consistent modelling between requirement specification and component specification that are written separately by component users and component developers. In order to overcome this difficulty, we propose three solutions:

1) Map names between specifications developed by component users and developers. This method can only handle simple naming differences, and the precondition is that the number of the mapping names from two specifications should be the same, and two different mapping names should have the same type and same meaning. We have implemented this in our prototype system. But sometimes this method is too strict to get satisfactory results.
2) Adjust the user requirement specification manually. By informal explanations attached with the component specification, this method can increase the possibilities of getting the required components. We have used this method in this case study, combined with the name mapping method. We believe that at the moment this combination can get the best results. However, some manual work is needed, and since it is not completely automatic, the selecting process takes longer time.
3) Formally define how to transform from one model (a transition system) to another. Thus more automation can be achieved. However this method needs future exploration on how to use in our framework.

Comparing with the traditional keywords-based searching method, our approach can save system development time by following a semi-automatic process with tools support. The only effort is to write SCIL specifications. We record that approximately it takes a person one hour to write the SCIL specification for a shopping cart component. This does not include the time spending on reading the component's user manual or other documents, because we assume that the person who develops the SCIL specification should be the person who develops the component, thus has enough knowledge about the component. It takes about half an hour to specify the user requirements in SCIL. Searching by keywords is fast, for our sample repository, it usually takes less than 20 seconds. However, it takes much longer time to review candidate component specifications and modify the requirement specification. On average 15 minutes for each modification is necessary in this case study, although the component specification is fully commented. Thus in order to make the component selection faster, we need to reduce the number of candidate components filtered by keywords. Thus picking appropriate keywords is important. We think that it is impractical for users to have more than 15 components to view and modify their requirement specifications accordingly.

If components cannot meet user requirements, according to our collaboration model [2], communication between users and developers are required. Developers can adjust or customise the SCIL description of each component. Such SCIL descriptions are used to search for components and will be sent back and forth between users and developers till user requirements are fulfilled.

VI. Related Works

Works related to our approach mainly focus on component specification and component retrieval techniques.

CORBA IDL (Interface Definition Language) [11] is a typical approach to informal specifications. It is easy to understand, however, it describes only the services the component provides but not the services it requires to accomplish its tasks. Operation descriptions are syntactic. The constraints on how and when the operations may be invoked cannot be expressed.

The Java Modelling Language (JML) is a behavioural interface specification language for Java. JML combines the practicality of DBC language like Eiffel with the expressiveness and formality of model-oriented specification languages [12]. JML uses Java's expression syntax to write the predicates used in assertions, such as pre/post-conditions and invariants. Spec# [13] is a similar approach for C# language. However they use specific analysis tools while SCIL can support a number of complementary analyses tools. Both JML and Spec# are much richer languages and verifying joint behaviour of components expressed in them requires the use of sophisticated theorem provers [14]. Also, it is difficult to use JML and Spec# to specify when a method should be invoked. Temporal properties are especially important when the component interacts with other components.

One can use formal descriptions to specify component interactions at an abstract, but sufficiently precise, complete, and unambiguous level. There have been a number of efforts in introducing temporal aspects into component interface specifications. These approaches are based on finite state machines [15], [16], temporal logic [17], [18], [19], process algebras [20], [21], message sequence charts [22], etc. While formal methods have been successful for specifying behavioural properties of components, and tools are developed to check these properties, they are still not popular with practitioners. The usual reason is that formal methods require a strong background in mathematics.

Efficient component retrieval depends on the way in which components are classified, specified and stored. Mili et al. divide component retrieval methods into three major categories [23]: text-based, lexical descriptor-based, and formal specification-based encoding and retrieval [24]. Some classification techniques that have been used include enumerated [25], keyword [26], faceted [27]. The main problem of these classification methods, however, is that an agreed vocabulary has to be developed and component users have to be familiar with this vocabulary [23].

Current CBD approaches, such as RUP [28] and Catalysis [29] are characterised by a common use of UML, as well as object-oriented concepts and constructs. But none of them has addressed in details how to select commercial components in CBD. Thus they are specially suitable for using in-house developed components. Selective Perspective [30] supports building systems from commercial components (COTS). It provides a model of collaboration by component suppliers, consumers, and brokers. However, it does not address how the collaboration can contribute to component selection. All these approaches only provide frameworks and guidelines to select components, and various methods and tools can be used within these frameworks.

VII. Conclusion and Future Work

In this paper, we have reviewed our approach on how to leverage formal methods of checking behavioural compatibility of components by defining an intermediate language that resembles a programming language which can be translated to existing modelling languages. We have also applied this framework in searching components for a generic e-commerce application. By the experiment and comparison, we think our approach can decrease the system development time but increase the precision of component selection.

Our future work includes adding support to other tools such as interface automata [34] and TICC [35]. We will also explore how to transform from one transition system to another in order to achieve full automation of SCIL specification matching.

References

[1] C. Abst, B. Boehm, and E. Clark, "Empirical Observations on COTS Software Integration Effort Based on the Initial COCOTS Calibration Database," in *ICSE 2000 COTS Workshop*, Limerick, Ireland, 2000.

[2] P. Krishnan and L. Wang, "Supporting Partial Component Matching," in *Distributed Computing and Internet Technology: First International Conference, ICDCIT 2004*, Bhubaneswar, India, 2004, pp. 294–303.

[3] L. Wang and P. Krishnan, "A framework for checking behavioral compatibility for component selection." in *17th Australian Software Engineering Conference (ASWEC 2006)*, 2006, pp. 49–60.

[4] ComponentSource, "http://www.componentsource.com," 1996-2006.

[5] TopCoder, "http://software.topcoder.com/," 2000-2003.

[6] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. London, UK: Prentice-Hall International, 1997.

[7] R. Alur, L. de Alfaro, R. Grosuz, T. Henzinger, M. Kangy, R. Majumdar, F. Mang, C. Kirsch, and B. Wangy, *Mocha Manual*. [Online]. Available: http://www.cs.sunysb.edu/ mocha/doc/j-doc/j-manual.ps

[8] D. Jackson, *Alloy 3.0 Reference Manual*, 2004. [Online]. Available: http://alloy.mit.edu/reference-manual.pdf

[9] E. Gagnon, "SableCC, an Object-Oriented Compiler Framework," Master Thesis, McGill University, Montreal, 1998.

[10] R. Alur and T. A. Henzinger, "Reactive Modules," *Formal Methods in System Design*, vol. 15, pp. 7–48, 1999.

[11] O. M. Group, "CORBA 2.0 Specification," 1996. [Online]. Available: ftp://ftp.omg.org/pub/docs/formal/97-02-25.pdf

[12] G. Leavens and Y. Cheon, "Design by Contract with JML," 2003. [Online]. Available: http://www.jmlspecs.org

[13] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview," in *In CASSIS 2004, LNCS*, vol. 3362. Springer, 2004.

[14] E. Poll, J. van den Berg, and B. Jacobs, "Specification of the JavaCard API in JML," in *Fourth Smart Card Research and Advanced Application Conference (CARDIS)*, D. Chan and A. Watson, Eds. Kluwer Academic Press, 2000, pp. 135–154.

[15] R. Reussner, "Enhanced Component Interfaces to Support Dynamic Adaption and Extension," in *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9*. Washington, DC, USA: IEEE Computer Society, 2001, p. 9043.

[16] D. M. Yellin and R. E. Strom, "Protocol Specifications and Component Adaptors," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 2, pp. 292–333, 1997.

[17] N. Aguirre and T. Maibaum, "A Temporal Logic Approach to Component-Based System Specification and Reasoning," in *The 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly*, Orlando, Florida, 2002.

[18] J. Han, "Temporal Logic-based Specifications of Component Interaction Protocols," in *ECOOP Workshop on Object Interoperability*, 2000.

[19] Y. Jin and J. Han, "Specifying Interaction Constraints of Software Components for Better Understandability and Interoperability," in *COTS-Based Software Systems, 4th International Conference, ICCBSS 2005*, Bilbao, Spain, 2005, pp. 54–64.

[20] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 3, pp. 213–249, 1997.

[21] C. Canal, E. Pimentel, and J. M. Troya, "On the Composition and Extension of Software Components," in *Foundations of Component-Based Systems Workshop*, Zurich, Switzerland, 1997.

[22] B. Finkbeiner and I. Kreger, "Using Message Sequence Charts for Component-Based Formal Verification," in *OOPSLA 2001 Workshop on Specification and Verification of ComponentBased Systems*, Tampa, FL, USA, 2001.

[23] A. Mili, F. Mili, and A. Mili, "Reusing Software: Issues and Research Directions," *IEEE Transactions on Software Engineering*, vol. 21, no. 6, pp. 528–562, 1995.

[24] A. M. Zaremski and J. M. Wing, "Specification Matching of Software Components," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 4, pp. 333–369, 1997.

[25] W. B. Frakes and T. P. Pole, "An Empirical Study of Representation Methods for Reusable Software Components," *IEEE Transactions on Software Engineering*, vol. 20, pp. 617–631, 1994.

[26] T. Isakowitz and R. J. Kauffman, "Supporting Search for Reusable Software Objects," *IEEE Transactions on Software Engineering*, vol. 22, pp. 407–423, 1996.

[27] R. Prieto-Diaz, "Implementing Faceted Classification for Software Reuse," *Communications of the ACM*, vol. 34, no. 5, pp. 89–97, 1991.

[28] P. Kruchten, *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 2000.

[29] D. F. DSouza and A. C. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley Professional, 1998.

[30] P. Allen, *Component-Based Development for Enterprise Systems : Applying the Select Perspective*, S. Frost, Ed. Cambridge University Press, 1998.

[31] J. Oberleitner, T. Gschwind, and M. Jazayeri, "The Vienna Component Framework: Enabling Composition across Component Models," in *the 25th International Conference on Software Engineering (ICSE)*. IEEE Press, 2003.

[32] J. Hatcliff, W. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," in *the 2003 International Conference on Software Engineering*, 2003.

[33] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby, "A Language Framework for Expressing Checkable Properties of Dynamic Software," in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. London, UK: Springer-Verlag, 2000, pp. 205–223.

[34] L. de Alfaro and T. A. Henzinger, "Interface Automata," in *the Ninth Annual Symposium on Foundations of Software Engineering*. ACM Press, 2001, pp. 109–120.

[35] B. T. Adler, L. de Alfaro, L. D. D. Silva, M. Faella, A. Legay, V. Raman, and P. Roy, "Ticc: A tool for interface compatibility and composition," School of Engineering, University of California, Santa Cruz, Tech. Rep., 2006.