DOCTORAL THESIS

A Collaboration Framework of Selecting Software Components Based on Behavioural Compatibility with User Requirements.

Wang, Lei

Award date:
2006

BOND UNIVERSITY

A Collaboration Framework of Selecting Software Components based on Behavioural Compatibility with User Requirements

by

Lei Wang (BE, MSc)

A Thesis Submitted
in Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Gold Coast, Queensland, Australia

November, 2006

Statement of Originality

This dissertation represents the author's own work and contains no material which has been previously submitted for a higher degree at this University or any other institution, except where due acknowledgement is made.

Signature:

Date:

Abstract

A Collaboration Framework of Selecting Software Components based on Behavioural Compatibility with User Requirements

by

Lei Wang (BE, MSc)

Building software systems from previously existing components can save time and effort while increasing productivity. The key to a successful Component-Based Development (CBD) is to get the required components. However, components obtained from other developers often show different behaviours than what are required. Thus adapting the components into the system being developed becomes an extra development and maintenance cost. This cost often offsets the benefits of CBD. Our research goal is to maximise the possibility of finding components that have the required behaviours, so that the component adaptation cost can be minimised.

Imprecise component specifications and user requirements are the main reasons that cause the difficulty of finding the required components. Furthermore, there is little support for component users and developers to collaborate and clear the misunderstanding when selecting components, as CBD has two separate development processes for them. In this thesis, we aim at building a framework in which component users and developers can collaborate to select components with tools support, by exchanging component and requirement specifications. These specifications should be precise enough so that behavioural mismatches can be detected.

We have defined Simple Component Interface Language (SCIL) as the communication and specification language to capture component behaviours. A combined SCIL specification of component and requirement can be translated to various existing modelling languages. Thus various properties that are supported by those languages can be checked by the related model checking tools. If all the user-required properties are satisfied, then the component is compatible to the user requirement at the behavioural level. Thus the component can be selected. Based on SCIL, we have developed a prototype component selection system and used it in two case studies: finding a spell checker component and searching for the components for a generic e-commerce application.

The results of the case studies indicate that our approach can indeed find components that have the required behaviours. Compared to the traditional way of searching by keywords, our approach is able to get more relevant results, so the cost of component adaptation can be reduced. Furthermore, with a collaborative selection process this cost can be minimised. However, our approach has not achieved complete automation due to the modelling inconsistency from different people. Some manual work to adjust user requirements is needed when using our system. The future work will focus on solving this remaining problem of inconsistent modelling, providing an automatic trigger to select proper tools, etc.

Acknowledgements

First and foremost, I am indebted to my supervisor, Dr. Padmanabhan Krishnan. I would never have accomplished this work without his continuous support and guidance. He has always been available and patient to answer my questions, and committed himself to offering his assistance whenever needed. His enthusiasm and scholarly insights made the process of pursuing my Ph.D. a joyful and fruitful journey.

I would also thank Dr. Michael Rees and Mrs. Stephanie Patching for their kind support during my study at Bond. Michael has presented me with many opportunities to improve myself, while Stephanie has helped me with all the academic issues. Without them I would never be able to focus on my research, and my life in Australia would be harder.

I am grateful to Dr. Daniela Mehandjiska-Stavreva, who took me to Bond University to fulfil my dream. I respect her spirit of never giving up even when she is fighting against serious illness. Her optimistic life attitude can never be learned from books.

Thanks also go to Dr. Luca de Alfaro from UC Santa Cruz for discussing the use of TICC and Mocha, and Xiaowan Huang from Stony Brook University for kindly providing the source code of Mocha. I also want to say "you have done a great job!" to the Alloy development group from MIT. Without these tools my research would never be practical. Another important contribution to my research comes from Yahoo Alloy discussion group, in which I have learned a lot from other Alloy users. They are always willing to answer my questions no matter how naive they are.

I would like to thank the School of Information Technology, Business Faculty for providing me with a four-year scholarship to complete my study, and offering me opportunities to improve my teaching skills. I would also appreciate the financial support and training opportunities from BURCS (Bond University Research and

Consultancy Services). I really enjoy the discussion with my friendly colleagues: Dr. Zhaohao Sun, Dr. Jun Han, Ping and Tan. I appreciate them for creating such a pleasant office atmosphere.

I would also like to thank those people whose names I have not mentioned, but have spent time with me. Thanks all!

Special thanks to Pat's family for their prayers and support both financially and mentally. Even though Seattle is far away, I still can feel their care and love. Thanks to Dr. Ningping Yu's family for their friendship, which always makes my heart warm.

My mother and father and my wife have always been sources of great love and encouragement. Their love shaped me into the person I am today. I am grateful to all the rest of my family members, especially my sister, uncle, cousin, for their unconditional support.

Finally, I want to thank Megumi Tanaka, a special person to me, and her whole family, for their love and understanding.

Contents

List of Figures

List of Tables

List of Programs

Chapter 1

Introduction

As size and complexity of software systems are dramatically growing, software components have been proposed as the main technology to address this challenge. By enabling reuse, components permit one to rely on the subsystems developed by other developers to simplify system design and implementation. This reduces the time and effort required to develop the entire system. That is to say, by using components the building of complex systems can be simplified, so that productivity can be increased. A typical Component-Based Development (CBD) has the steps including analysis of requirement, selection of component, adaptation of component, composition with the existing system, and verification and validation of the enlarged system.

The key to a successful CBD is to get the required components. As there are a large variety of different components from different sources, users of components must be able to identify and choose components that suit their needs. This is *component selection*.

Component selection is crucial because the other development steps will depend on it. A wrong selection of the components can cause the failure of the entire system [128], as the wrongly selected components are not compatible to the target system, thus they cannot work properly. If the incompatibilities are found at a later stage than component selection, one needs to go back to find and select the components

again. As a consequence, the completion of the whole project will be delayed. Most of the top risks in CBD projects mentioned in [20] come from the step of selecting components.

In order to reduce the risks, component selection should be considered throughout the entire life cycle of the 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 [45]. Thus they are able to adjust the requirements accordingly. This would help system developers to easily locate the required components in the later development stages.

1.1 Motivation

The motivation to conduct this research initially comes from the problem of the current widely applied practice of selecting components.

1.1.1 The Current Widely Applied Practice

Of the current search engines for software components, whether a general search engine such as Google, or a particular search engine for a component repository such as the one for ComponentSource [42], only a free text-based search is supported. We consider a free text-based search as the current practice of searching components because these search engines are the most widely used. For example, ComponentSource [42] is the most popular component repository in the world, providing a large variety of components from different developers.

By this approach, component users first can receive a list of options by entering free text in the search engine. However, in order to retrieve the one that suits their requirements, users have to examine these components one by one.

The examination of each component normally follows these steps: first, one has to look at the textual description of the component to see whether the component is

relevant to the requirement; if it is relevant, the user can download the component manual to understand how to use the component; the user can also download the trial version of the component and test it in the user's working environment to check whether it really can meet user requirement.

The results obtained by this approach have exposed its weaknesses. First, except for the free text-based searching, most selection work is manual. For example, one often receives a long list of candidate components, so it takes a lot of time to decide their relevance. Testing components is also time-consuming. Second, even with so much time spent, the component retrieved still may not meet the user requirement. This is because manual work easily causes mistakes. If this happens, adapting the component at a later stage would be more difficult.

1.1.2 Problem Statement

The idealised CBD process assumes that the components obtained from other developers are sufficiently close to the units identified when decomposing the system that is being developed, so that the component adaptation requires less effort than the unit implementations. However, this is rarely the case in reality, especially when the components are not in-house developed, for example, Commercial Off-The-Shelf (COTS) components. People often cannot get the required components. Thus adapting the components into the system becomes an extra development and maintenance cost [2]. This additional cost may trade off the benefits of CBD. This is the problem of component selection, especially when selecting from COTS components.

In the current practice of searching components by free text, component specifications and the user requirements for the components are imprecise. For example, in ComponentSource, components are organised in categories, but presented in textual descriptions. Thus it is difficult to get the user-required components with imprecise specifications. Furthermore, this problem is difficult to solve due to the nature of CBD, which involves two separate development processes: develop components and

use components. These two development processes do not have an obvious connection to each other. That is to say, even if the misunderstandings on components or requirements exist, it is hard to get them clarified.

Meanwhile, there are a large variety of components, but these components are not tailor made to the particular 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. However, if component developers can get involved in component selection by customising their products based on user requirements, the possibility of finding and retrieving the required components can be increased. This requires an unambiguous understanding of components and user requirements, as well as the collaboration of component users and developers, in which components can be customised.

1.1.3 Different Approaches

Researchers have addressed the problem of component selection from different perspectives. Some focus on specification techniques because component retrieval based on matching needs an understanding of component specification, which could be formal or informal. Informal specification is easy to write and read, thus it is commonly used in current practices, such as ComponentSource [42] and TopCoder [142]. However, the search based on informal specifications often has irrelevant results that cannot be integrated into the targeting systems. This is because informal specifications are ambiguous, and only contain syntactic characteristics of component interfaces; they can match many related items.

Some work has been done to extend the syntactic approaches by adding invariants and pre/post condition pairs to constrain component behaviours, such as JML [100], Spec# [15]. Unified Modelling Language (UML) [21] is a semi-formal modelling language, widely used in design and documentation. Integrated with Object Constraint Language (OCL) [77], UML can also describe component behaviours using the ex-

isting notations. Combining such techniques with existing programming languages often results in a fairly complicated and very detailed specification. One is then unable to use such a specification to retrieve components for a particular situation. Another problem of these approaches is that it is difficult to specify *when* an operation should be invoked. When describing the interactions among components, the order by which the operations of the components should be invoked is important. Thus a complete component specification needs to include temporal properties of component behaviour.

One can use formal descriptions to specify component behaviours, including semantics and interaction rules of component interfaces at an abstract, but sufficiently precise and complete level. Formal descriptions also enable the checking of consistency and correctness of the interface models. While formal methods have been successful for specifying behavioural properties, such as those described in [4, 7, 28, 52, 58, 91, 99, 104], and tools have been developed to check these properties, they are still not popular with practitioners. The reason is that formal methods require a strong background in mathematics. It is currently unrealistic to expect normal component users and developers to have such a background. That is why we cannot find existing component repositories providing formal specifications of components.

Other researchers focus only on detailed component storage and retrieval techniques, assuming that components and requirements have been specified. In such cases the focus involves component classification and matching.

Several classification schemes such as keywords [86], enumerated [62], faceted [130] and hypertext [44] can facilitate the user's search of components. However, component users may find it difficult to take advantages of such classification schemes if they do not know the vocabularies that are used to build the schemes [116]. One way to overcome this limitation is to define a natural language user interface [137], by which the users input queries in natural languages, such as English. The input queries will be analysed and decomposed into the previously defined classification

vocabularies. Classification-based component retrieval techniques can help to filter irrelevant components, but cannot guarantee that the components retrieved will have the expected behaviours. This is because classification schemes do not specify the behavioural properties of components. A further examination of the retrieved components is needed.

With specifications, component matching is performed upon signatures [152] and behaviours [153]. Signature matching is also mainly syntactic and thus it is hard to receive expected results. Behaviour matching relies on semantics and interaction protocol of the interface model, so it can enhance the possibility of the component retrieved being integrated and working as expected in the target system.

After the components have been matched and retrieved, they need to be evaluated in order to decide the best fit. Some researchers view component evaluation as software engineering discipline, starting from requirement analysis, through a whole system life cycle. Most research work in component evaluation focuses on the evaluation processes [94, 105, 123] that are driven by models, which include the product descriptions and evaluation criteria, and the particular methods such as multiple-attribute utility [47] or component ranking [108]. The lessons learned from [122] have told us that component evaluation is not easy. This is because component users lack the visibility into the internal workings of components, thus it is difficult to form an appropriate and comprehensive view on how to evaluate those components.

Some component developers can provide automatically customised components based on particular user requirements using technologies such as software product line [40], or software factories [74]. In this case, how component users express their requirements unambiguously so that component developers will not misunderstand becomes a major problem.

Iribarne et al[85] integrate a component trader into a spiral methodology for CBD by using a series of XML-based templates to document components, services and queries. This work, however, does not cover semantic trading. Select Perspective[6]

tries to establish a collaboration framework for providing and obtaining the right COTS components, in which informal repository of ComponentSource[42] is integrated. However, the framework does not address in detail how the collaboration can contribute to component selection, and no particular methods of selecting components are provided.

1.2 The Proposed Solution

In order to minimise the additional cost of component adaptation, such as writing glue code, components need to be tailor made to user requirements. The research performed in [2] has suggested that

> *writing glue code typically takes longer and requires more effort to complete than tailoring. This may be because the intellectual effort required to simply configure (or tailor) a given COTS product is usually less than that required to create code around it that is not only new, but also highly constrained – the situation that exists with glue code.*

1.2.1 A Collaboration Process

Only the in-house developed components can be tailor made to user requirements. Since the components are built by other internal developers, the component user can get access to all the component documents, including the source code. The process of selecting components becomes teamwork within the same organisation. These components can be tailor made to the requirements. Even if the components cannot meet user requirements, the developers can easily modify the component implementations to suit user needs. This is because the communication within the same organisation is easy.

If the components are built from external developers, the communication between component users and developers becomes difficult. However, the teamwork still can

be done if two conditions are met: first, it should be in the interests of the external developers in assisting component users to find and retrieve components; second, there should be technologies and tools supporting the collaboration. These two conditions are not necessary when components are tailor made to the user requirements.

Recently we conducted an informal survey of seven developers from different organisations that develop components. It showed that six of seven developers and their organisations are willing to help by customising their products based on user requirements. For example, the developer from Keyoti [93] said, "Generally we prefer to do free customisations when we feel they will improve the product and make it more marketable". Only one developer hesitated due to the high cost of maintenance for different versions of components. However, the developer said, "If that's something that might be useful for other customers, then we definitely do it, so that others can utilise the same feature, and we extend our customer base".

For communication purposes, a common language is needed by both component users and developers to exchange information. The information can be the specifications of user requirements and components. When determining whether components satisfy user requirements, one should be able to use model checking tools to check the specifications. These tools should be available for both sides of the collaboration. Additionally, a search engine that is built on the checking tools allows component users to match components by the specifications.

From the perspective of component developers, they are facing the dilemma of generality and efficiency on component design. That is to say, the components must be sufficiently general to cover the different aspects of their use, but meanwhile they must be concrete and simple enough to serve a particular requirement efficiently [46]. According to Szyperski [139], developing a reusable component requires three to four times more effort than developing a component that is for a particular purpose. It is impossible for component developers to service all the user needs, because a diverse set of user requirements exist. Thus some promising technologies can be used to facilitate

the component customisation, such as generative programming [49], software product line [40] or software factories [74]. These technologies are based on modelling software system families, and aim at providing highly customised and optimised intermediates or end-products that can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge, given a particular requirement specification [49].

The collaboration can be established when users are selecting components, but the component selected cannot completely satisfy the user requirements. Then component users may ask the component developers to customise the components for their specific requirements. The collaboration process can take these steps: first, the component user submits the requirement specification to the component developer; then the developer can use the checking tools to find which user-required properties are not met; finally, the component is modified (customised) so that all the required properties can be satisfied. This customised component will be sent back to the user.

In traditional CBD approaches, because developing components and using components are mostly separated, component developers and users do not have direct connection. Thus the additional cost of adapting components is mainly paid by component users (see Figure 1.1 – a). If components can be customised for different users, component developers will put in more effort. However, the component customisation job for component developers is easier than the component adaptation job for component users to do because the developers know the internal workings of their products. Thus for the task of selecting and adapting one component, the overall effort can be saved according to the data collected by [2] (see Figure 1.1 – b).

By allowing the exchange of requirements and component descriptions, the collaboration can help component users to make their requirements clearer, and can ensure component developers deliver suitable components to users. The collaboration can also solve the problem of being unable to find suitable components, because either component users can adjust their requirements, or component developers can provide

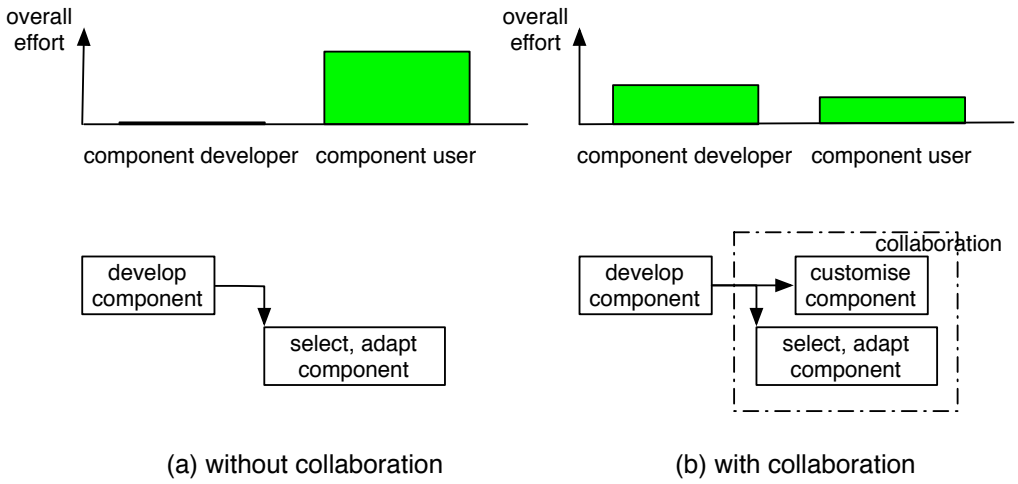

Figure 1.1: Adaptation Cost with/without Collaboration

additional choices by customising their products.

1.2.2 The Technologies to Support the Process

In order to write an unambiguous component specification, one should include semantics of the interface model and the interaction rules with its environment in the specification. Using heavyweight formal methods can serve the purpose, but it is not practical because there will not be many people who have good mathematical skills.

As a solution, we define Simple Component Interface Language (SCIL), a lightweight formal approach to achieving both precision and practicality. That is to say, SCIL has easy to understand syntax, but can precisely describe components.

It is unrealistic for a specification language to capture everything, including signatures, behaviours and other details. Thus SCIL only focuses on the high-level behaviours of components.

There exists 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.

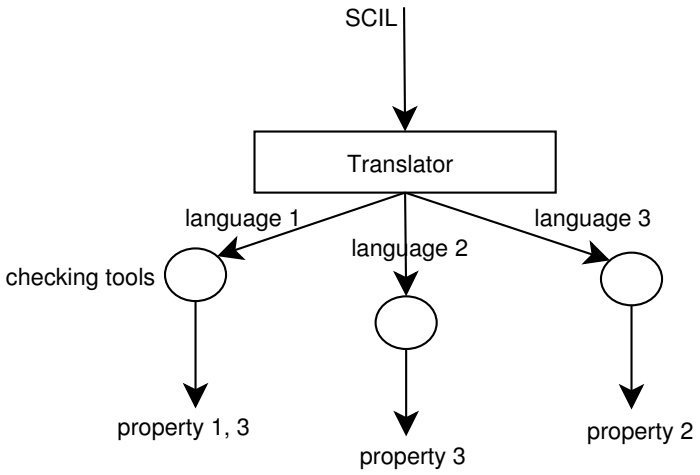

Figure 1.2: Using SCIL as the Bridge to Checking Various Properties

SCIL acts as a bridge to access these previously developed tools. For example, in Figure 1.2 different languages can be used to check different properties. By translating SCIL to a variety of modelling 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.

In the collaboration framework, SCIL is also used as the communication language. If component users and component developers use different languages to describe requirements and component products, communication would become difficult.

1.3 Overview of the Methodology

In this thesis, we focus on setting up a collaborative process of selecting components performed by both component users and component developers, and providing communication language with tools support. However, the collaboration is not forced in our framework, but enabled directly for better selecting results. In other words, it is up to users and developers to decide on the nature of collaboration. The main contribution of the thesis is to provide tools and methods to support the collaboration.

Current tools and methods do not support this directly.

We design and use SCIL as the interface specification language to capture component behaviours. SCIL views user requirements as components as well. Although other languages, such as JML, can be possibly used in the framework, we use SCIL as the communication language in the collaboration for the proof of concept purpose.

Based on SCIL, we have developed the translator to other existing languages and a web-based component selection prototype system, in which components can be matched by their behavioural compatibility with user requirements.

By applying the prototype system in three case studies, we can evaluate whether the system can indeed find components based on behaviour without exposing formal details. However, we are unable to conduct experiments on the collaboration process, because it is difficult to get industry involvement. Instead we present a recommended practice, which we believe can help in finding required components.

Component selection involves the activities of specifying components, matching and retrieving components, and evaluating components. The tools need to aid all these activities. However, we only focus on specifying and matching components. Component evaluation is not the focus of this research, because it involves a lot of details that do not belong to behavioural properties, such as implementation platform or developer's background, etc. We believe that when the behavioural properties are satisfied, components can be accepted and customised by the collaboration. Therefore, most evaluation work can be saved.

It needs to be noted that component customisation is also not forced in our collaboration framework. It is performed only when developers think it necessary and worthwhile. Thus the tools support for the collaboration process is mainly for component users, while the techniques of customising components used by component developers are outside the scope of the thesis. Even if customisation is chosen, it is still a manual process since there are no tools provided at the moment. SCIL does not automate customising components, but only tells whether a component satisfies

user requirements or not and points out the shortcomings.

1.4 Thesis Outline

The rest of this thesis is structured as follows:

`Chapter 2` briefly introduces the background knowledge of software component technology, examines the current state of the art in component selection, and highlights the advantages and shortcomings of the various approaches.

`Chapter 3` presents in detail our approach to selecting required components, including the collaborative process, the common specification language and the tools support.

`Chapter 4` shows the design of our specification language – Simple Component Interface Language (SCIL). And through an example, we demonstrate how SCIL can be used to describe user requirements and specify components.

`Chapter 5` gives the architecture and the implementation details of our tools. We explain how these tools can work together to search for required components.

`Chapter 6` elaborates on three case studies. The first continues the example from Chapter 4 to illustrate the features of the tools we have used. The second shows how a user can select a spell checker component by SCIL specifications. The third is to find components for a generic e-commerce application. Some limitations and the potential solutions are also discussed.

Finally, `Chapter 7` summarises the thesis and examines how the work can be taken further in the future.

Chapter 2

Related Works

2.1 Introduction

As early as 1968, software components was suggested by McIlorys [110] as a way of tackling the software crisis. Yet only in the past decade or so was the idea of Component-Based Development (CBD) proposed. Nowadays the component-based approach has already shown considerable success in many application domains.

CBD focuses on building software systems by assembling previously existing software components. Borrowing ideas from hardware components, software components are written in such a way that they provide functions common to many different systems. Allowing components to be reused, CBD has the potential to reduce development time and cost while increasing development productivity. Meanwhile, it becomes possible to replace parts (components) of software systems with newer and functionality equivalent components. Thus component systems are flexible and easy to maintain.

Recently there has been increasing interest of Commercial Off-The-Shelf (COTS) components that embody a *"buy, don't build"* [26] approach to constructing software systems. Selecting the suitable components against user requirements is especially difficult for such an approach, because the source code of COTS components cannot

be accessed by the users. Thus how to describe components precisely is especially crucial in COTS component selection.

Technically, component selection involves the steps of specifying components and requirements, matching and retrieving components, and evaluating components. Some non-technical aspects, such as the component supplier's reputation and market share, sometimes also affect the selection of components.

This chapter first briefly gives the background knowledge of component technology, and explains the key concepts of CBD that are related to component selection. It then examines how the existing approaches have addressed the different problems in component selection. We highlight the strengths and shortcomings of these approaches.

2.2 Background Knowledge

2.2.1 Software Component

According to Szyperski [139], a software component is defined as:

> *a unit of composition with contractually specified interfaces and explicit context dependencies only. A component can be deployed independently and is subject to composition by third parties.*

According to this definition, a software component consists of a set of interfaces, and the encapsulated implementations of these interfaces, which cannot be directly accessed from its environment. The separation of interfaces and their implementations makes it possible to either add new interfaces (with new implementations) without changing existing implementations, or replace old implementations with new ones without affecting their interfaces. A software component is produced to be composed with other components and deployed in a container. This is done by any person other than its developer through the component interfaces and the contracts attached to

these interfaces without the knowledge of the component's internal workings.

In this thesis, any kind of reusable units with specified interfaces, such as classes or libraries, can be regarded as components. The size of components is not important. For example, an *assembly*, which is an aggregation of components that provides integrated behavior, can be regarded as a component. A *framework* is also a component.

2.2.2 Concept of Interface and Contract

Interfaces are the means to using software components. An interface (see Figure 2.1) may contain one or more methods or operations by which users interact with the component. Methods are divided as inputs and outputs. Input methods receive requests while output methods generate results. Abstractly, all input methods can be grouped into required ports while all output methods can be grouped into provided ports. Both required ports and provided ports are dependent on working environment. A possible solution is to gather all the environment-dependencies of one component into one location, so that the component can easily adapt to new environments without any unexpected dependencies [12]. This is useful when quality properties are concerned. This approach requires that interfaces and their implementations are completely independent, which in reality is difficult to achieve [12].

Using interfaces usually involves a number of tacit agreements between the component and its users. For example, a component depends on another component to provide a service. In return, the latter component may rely on the former to provide data arguments within certain bounds, or to have properly initialised the component service. These are called *usage contracts* [33]. In CBD, there exists another type of contract, specifying the rules about how components should be implemented. These are called *realisation contracts* [33]. Corresponding to these two types of contracts (see Figure 2.1), we can write *interface specifications* and *component model specifications*.

However, in some literature component specification, interface specification and

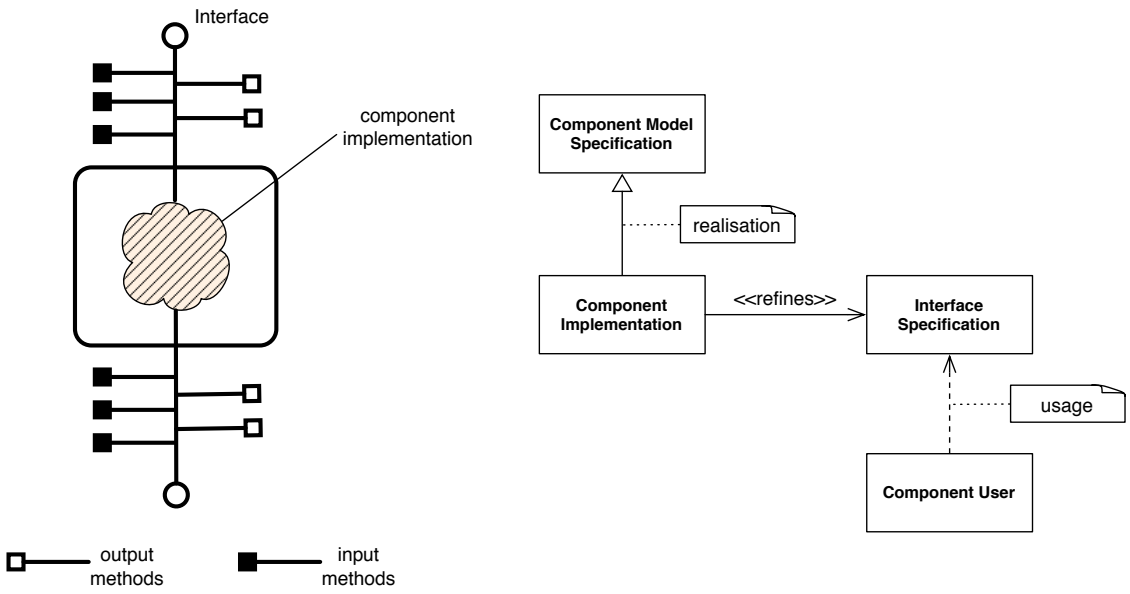

Figure 2.1: Interfaces, Methods, Contracts and Specifications

component model specification have been confusingly used. Interface specifications explain how components can be used, while component specifications describe what components can do. Interface specifications are simpler, but contain enough information for determining how the underlying components are composed. Component specifications do not make assumptions on their working environment. The detailed discussion on the differences can be found in [53]. Despite the differences, in this thesis sometimes we also *use component specification to mean interface specification*.

Component specification is also confusingly used in some literature to mean component model specification, which consists of component model, support infrastructure, implementation framework, as well as packaging and deployment models. The typical examples of component model specification include Microsoft's Component Object Model (COM) [43], Sun's JavaBean [113], Enterprise JavaBean (EJB) [114], and OMG's CORBA Component Model (CCM) [75].

2.2.3 Component-Based Development Life Cycle Model

Compared to traditional software development, component-based software development shifts development emphasis from programming systems to composing existing components to build systems.

2.2.3.1 Development Process

CBD is characterised by two separate but parallel development processes throughout the whole system life cycle. The first process is the component development undertaken by component developers. This process can follow an arbitrary development process model. As components are for reuse purposes, managing requirements is more difficult. Meanwhile a precise component specification is always required.

The second process is the system development performed by system developers (they are the component users). This process follows the steps of [47]: requirement analysis, architectural design, component selection, component adaptation, system integration, verification and validation, system maintenance (see Figure 2.2).

Requirement Analysis: This step is to analyse whether the requirements can be fulfilled by available components. This means that requirement engineers must be aware of the components that can be potentially reused. Since it is likely that no appropriate components can be found, there is a risk that the components will have to be implemented from scratch. In order to minimise this risk, system developers need to keep negotiating and adjusting the requirements to be able to reuse the existing components.

Architectural Design: This step should pay attention to architectural patterns, key architectural design principles including abstraction and separation of concerns, and system decomposition principles. A good decomposition satisfies the principle of loose coupling between components.

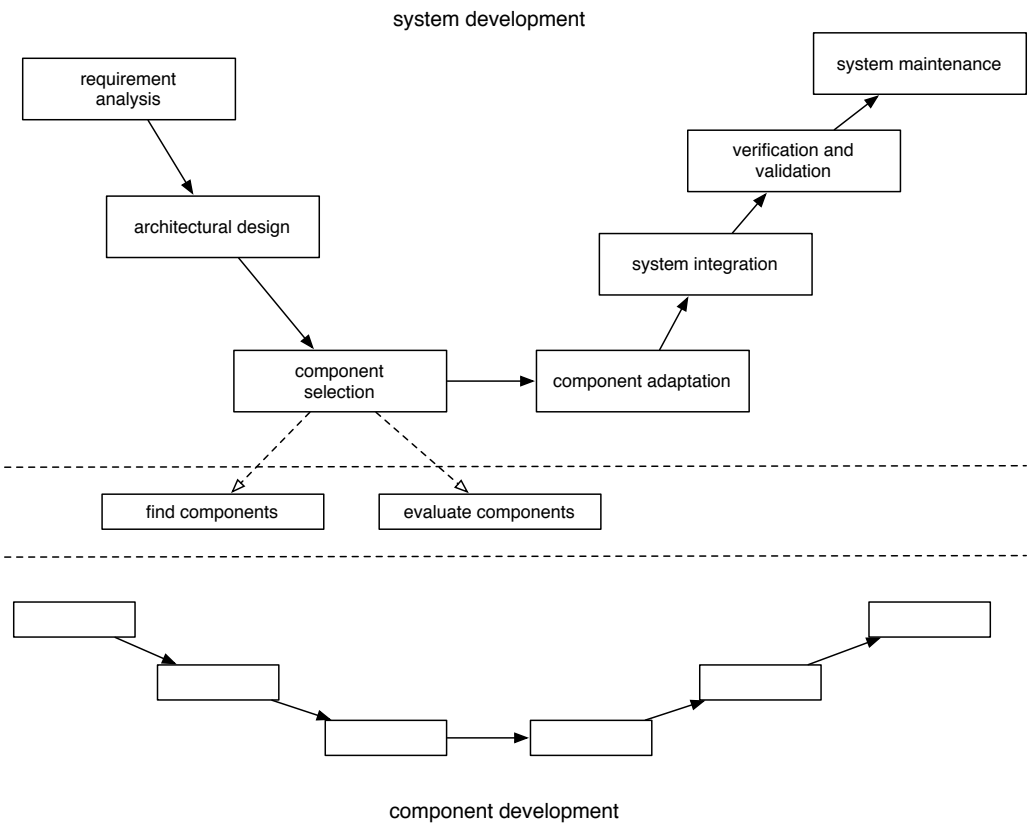

Figure 2.2: Two Development Processes of CBD

This step is also tightly connected to the availability of the components. Components are complying with a particular component model that requires a particular architectural support infrastructure. Thus the component model directly has impact on architectural designs. It is possible to use components that are implemented following different component model specifications. Vienna Component Framework (VCF) [121] is an approach to supporting the interoperability and composability of components from different component models. The advantage of VCF is that system developers do not have to understand the details of all the available component models when selecting components, instead they focus on the tasks that components can fulfil. However, at the moment VCF does not support converting the support

infrastructures of different component models, for example, transaction and security services.

Software architecture includes abstraction, rationale (which is related to requirements and implementation), architectural style [135], constraints, multiple models, multiple views [96], etc. Software architecture is increasing in importance as systems are becoming larger and more complex. It has been widely agreed that software architecture is the only way to guarantee quality attributes of the system that we try to build [13]. This is especially true when the system is built from commercial components of which the internal workings are completely invisible to component users. However, when many system qualities are expected, some of these qualities may conflict, so that trade-offs have to be made taking into account the relative priorities of these system qualities.

Component Selection: This step is to find components to fit into an underlying architecture. Since software architecture puts the constraints on component selection and the rationale for choosing a specific component in a given situation, one can identify components from the system architecture, generating the requirements for each potential component. Many methods and techniques of component identification have been introduced in [146].

Patterns also can help in identifying potential components [124]. A pattern is a recurring solution to a standard problem in a certain context [134]. Patterns enable people to exchange architectural knowledge and design paradigm.

Components in the repository will then be matched against the user requirements for the identified components. Several Architecture Description Languages (ADL) [41] have been developed to describe such requirements, e.g., architectural roles that components play. Components can be retrieved through a process of filling roles with the components within the architecture. In one word, architectural decisions have a large effect on the components selection [106].

The selection process should ensure that appropriate components have been selected with respect to their functional and non-functional properties. Some guidelines [88] have been given to choose components that satisfy specific performance metrics. This would require verification of the component specification, or testing of some of the component's properties that are important but not documented. Furthermore, even if isolated components function correctly, an assembly of them may fail, due to invisible dependencies and relationships between them [47]. This requires that components integrated in assemblies are tested before being integrated into the system.

Component Adaptation: This step is required to avoid architectural mismatches[65]. Since each component is developed targeting different requirements, making different assumptions about the working environment, the purpose of adaptation is to ensure that conflicts among components are minimised, or to ensure particular properties of the components or the system. There are several known adaptation techniques: using parameterised interface makes it possible to change the component properties by specifying parameters that are the parts of the component interface; writing wrappers to encapsulate components and provide new interfaces that either restrict or extend the original interfaces, or to ensure or add particular properties; writing adapters to modify component interfaces to make it compatible with the interfaces of other components. These different approaches depend on the accessibility of the internal structure of a component.

System Integration: This step includes integration of standard infrastructure components and the application components. The integration is done at two levels: particular components and the entire system. The integration of a particular component into the system is also called component deployment.

Verification and Validation: This step uses the standard test and verification techniques [89]. Since components are black-box types and delivered from different vendors, the errors are difficult to locate. Contractual interface specifications play an important role in checking the proper input and output from components.

System Maintenance: This step is mainly about replacing old components by new components or adding new components into the systems. The paradigm of the maintenance process is similar to this for the development: find a proper component, evaluate it, adapt it if necessary, and integrate it into the system.

2.2.3.2 Some Approaches to CBD

There have emerged some approaches that provide frameworks and best practices to achieve CBD. Following the guidelines provided by these approaches, various methods and tools can be used within these frameworks.

RUP (Rational Unified Process): RUP [97] is created by the Rational Software Corporation, now a division of IBM. RUP is not a single concrete prescriptive process, but rather an adaptable process framework. It follows an iterative and incremental way to construct software systems.

When handling component-based software systems, RUP focuses on producing the basic architecture in early iterations. This architecture then becomes a prototype in the initial development cycle. The architecture evolves with each iteration to become the final system architecture. RUP also asserts design rules and constraints to capture architectural rules. By developing iteratively it is possible to gradually identify components that can then be developed, bought or reused. These components are often assembled within existing infrastructures such as CORBA [76], COM [43] or J2EE [115].

RUP is characterised by a common use of Unified Modelling Language (UML) [21], as well as object-oriented concepts and constructs. But RUP has not addressed in

detail how to select COTS components in CBD. Evolutionary Process for Integrated COTS-Based Systems (EPIC) [5] extends RUP to accommodate COTS-based system development. It provides a framework that redefines acquisition, management and engineering practices to more effectively leverage the COTS marketplace and other sources of previously existing components.

Catalysis: Catalysis [55] has been developed and applied in many fields since 1992. It is a framework specially designed for component-based development. Catalysis makes separate development cycles for component kit architecture, component design and component assembly, and a cycle for development of the component repository. When creating a kit architecture, a domain model is built independent of any component's design. The domain model includes not only entities and relations, but invariants and dynamic constraints as well.

When designing components, detailed specifications are required. Component specifications can be formed by using UML plus Object Constraint Language (OCL) [77]. When a candidate implementation is presented, it should be tested against the specification model.

Connectors are defined separately from the components. Connectors are interfaces that can encompass the ideas of dialogues, protocols and transactions. A variety of techniques can be applied to defining connectors precisely. The central idea is the post-conditions defined on abstract models of the components' states.

However, Catalysis is specially suitable for using in-house developed components; it has not given details on how to cope with the situation if the required component specifications cannot be satisfied.

Select Perspective: Select Perspective [6], from Select Business Solutions, has evolved over the past 10 years to become a component-based development process. Select Perspective is unique in offering processes and techniques that fully support the creation of service-oriented architectures.

The main theme behind the Select Perspective process is of three workflows: Supply, Consume and Manage. The Supply workflow presents the steps for delivering components or services. The Consume workflow is an overall process for the activities that focus on the project-based delivery of the business solutions. In the Manage workflow there are two distinct streams of activity. One stream is concerned with the acquisition, certification and publication of components. The other stream focuses on the location and retrieval of candidate components for reuse.

Select Perspective picks components through a community of component suppliers, consumers, and the brokerage role between them. Searching ComponentSource [42] repository has been integrated into the Select Perspective toolset. When candidate components arrive, they undergo formal testing and certification. If the components are certified, then after classification and storage in the component repository, they are published ready for subsequent reuse. When component users have specified the components or services required for their construction work, they then search the component repository for matches to their requirements. Whenever candidates are discovered they can be retrieved and examined for their suitability before they are finally reused.

Selective Perspective supports building systems from COTS components. 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.

2.2.3.3 Focusing on Component Selection

As we mentioned before, component selection is the key step when constructing systems from previously existing components. Researchers are interested in how to effectively and efficiently find and retrieve the components that can satisfy user requirements. This problem has been addressed from different perspectives. Thus many concepts related to component selection have been introduced. We drew a concept map (see Figure 2.3) to show how these concepts can support the selection of com-

ponents.

Among these concepts, we focus more on the existing techniques of specifying component interfaces, storing and retrieving components, as well as evaluating components.

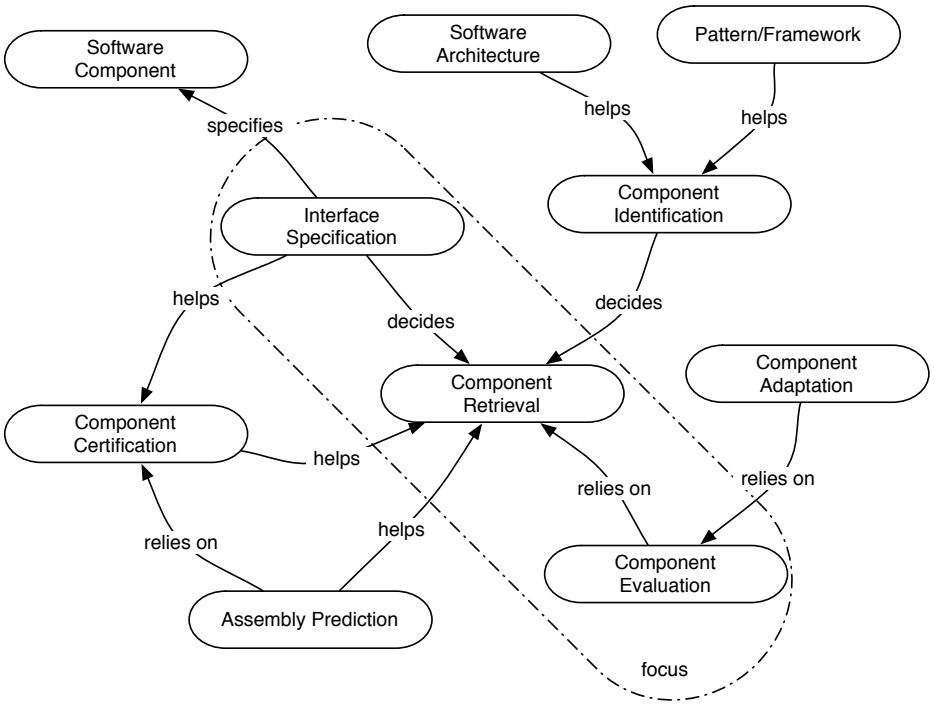

Figure 2.3: The Concept Map of Component Selection

A typical component selecting process involves some roles, which will be referred to throughout this thesis. We define them as below:

- A *component user* is a person or an organisation who selects and purchases the component and uses it in the system development. We use the term *component purchaser* to avoid confusing with *system user*. The synonyms to mean a component user in some literature include *component buyer*, *component evaluator*, *component integrator* or *system developer*.

- A *component vendor* is a person or an organisation who designs and implements the component, then makes it available to be reused. Others may use different terms to mean a component vendor, including *component provider*, *component supplier*, *component developer* or *component writer*.

- A *component broker* is a middleman between users and vendors, bridging their requirements and offerings. A component broker takes care of the trading place and negotiates contracts of purchase and sale. ComponentSource [42] is a typical online component broker.

2.3 Specifying Component Interfaces

Finding components needs an understanding of component interface specifications, which explains how a component can be used. The result of component selection depends on how precisely components can be specified. In general, three levels of interface specification can be identified: the syntactic level, the semantic level and the protocol level. Apart from that, the non-functional properties or quality attributes of components may also be written into the interface specification [19].

2.3.1 Syntactic Level

Interface specifications at syntactic level express only the syntactic characteristics of component interfaces, such as what services the component provides and the signatures of these services. Conventional API (Application Programming Interface) based on some programming languages, such as Java and IDL, is a typical approach at this level.

2.3.1.1 Interface Definition Languages

Interface Definition Languages (IDL) were originally designed to express object interfaces in client/server applications. When clients and servers are implemented in different languages and do not share the common call syntax, for instance, Java and Pascal have different ways of calling routines, IDLs are used to describe interfaces in a language-independent way. Program 2.1 shows a CORBA IDL [75] fragment of the auctioneer component interface.

Program 2.1 Auctioneer Interface in IDL

```
interface Auctioneer {
   void login(in Bidder b);
   void logout(in Bidder b);
   boolean bid(in Item item, in float offerPrice);
   string purchase(in Item item, in float finalPrice);
   void sell(in Item item, in float minPrice, in float maxPrice);
}
```

However, when used to describe component interfaces, IDLs exhibit a number of limitations [29]. The example (see Program 2.1) describes only the services the auctioneer 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.

2.3.2 Semantic Level

In order to overcome the limitations of Conventional API approaches, the idea of Design by Contract (DBC) [112] is used to augment interface definitions with semantic information. The semantic information includes the contract on interface operations, viz., the obligations of both the consumer of an operation and the provider of the operation.

In Design by Contract, the provider of the operation makes certain assumptions (the pre-condition of the operation) about how the operation is called. In return for the consumer meeting the terms of the pre-condition, the provider guarantees that the operation will result in certain properties (the post-condition of the operation) being met.

DBC also makes these contracts executable. The contracts are written attached to program code in the programming language itself, and then are translated into executable code by the compiler. Thus, any violation of the contracts that occur while the program is running can be detected immediately. The typical approaches following the DBC principle include Java Modelling Language (JML) [100], Spec# [15] and OCL [77].

2.3.2.1 Java Modelling Language

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 [100]. JML uses Java's expression syntax to write the predicates used in assertions, such as pre/post-conditions and invariants. For example, in an auction system, when a bidder pays for an item, the item has to be existing in the product repository, and also the buying

price should be greater than or at least equal to the minimal price set by the item seller. Thus when specifying the auctioneer component in JML, the method *purchase* would be written as in Program 2.2. If the pre-conditions are met (the *requires* part), the method guarantees that the result string would be the name of the item purchased.

Program 2.2 Code Fragment of Auctioneer Purchase Method in JML

```
/*@ public invariant !repository.isEmpty() &&
  @ (\forall Item item; repository.contains(item);
  @ item.minPrice >= 0 && item.minPrice <= item.maxPrice);
  @*/
  ...
/*@ requires item != null && repository.contains(item)
  @ && finalPrice >= item.minPrice;
  @ ensures \result.equals(item.name);
  @*/
public String purchase(Item item, double finalPrice) {
   ...
}
```

One can also write an invariant as in Program 2.2, specifying that the product repository is always available, and for all the items in the repository the minimum price should be greater than or equal to zero, but less than or equal to the maximum price set by its seller. In JML, invariants are class invariants, viz., they should hold for all instances of the class at any time. JML specifications are written in special annotation comments , which start with an at-sign(@).

The advantage of using Java's notation in assertions is that it is easier for programmers to learn and less intimidating than languages that use special purpose mathematical notations. Meanwhile JML extends Java's expressions with various specification constructs, such as quantifiers [27]. JML speculations are more precise than specifications in IDLs. However, JML cannot be used on non-Java code.

From the view of component selection, JML has two major problems. The first, with the obvious development-centric focus, is that JML specifications depend heavily on the internal elements of the component. Constraints are often expressed in terms of a property that is internal to the component [66]. Thus when selecting components, JML specifications are not always helpful.

In the second it is difficult to use JML to specify when a method should be invoked. Temporal properties are especially important when the component interacts with other components. For example, the *purchase* method is allowed to execute only when the bidder has logged into the system, while the user authentication function is delivered by another component. Moveover, verifying joint behaviour of components expressed in Java and JML requires the use of sophisticated theorem provers [127].

2.3.2.2 Spec#

Spec# [15] is very similar to JML. The Spec# language extends C# with contract specifications, analogous to the way JML extends Java. The Spec# compiler emits run-time checks that enforce the contracts and the Spec# program verifier uses theorem-proving technology to statically check the consistency between a program and its contracts. One difference between Spec# and JML is that Spec# builds in a new methodology for object invariants [14], trading restrictions on the kinds of programs that can be written for a sound modular reasoning technique.

Using Spec# to write assertions is similar to using JML. Invariants and pre/post-conditions are introduced with the keywords *invariant*, *requires* and *ensures* respectively. However, Spec# can only be used on C# code.

Spec# shares the similar advantages and disadvantages with JML when used to specify components for selection.

2.3.2.3 Object Constraint Language

Object Constraint Language (OCL) [77] is a declarative language for describing rules that apply to Unified Modelling Language (UML) [21] models. OCL supplements UML by specifying constraints on objects defined by UML.

OCL has the power of the Lower-order Predicate Calculus (LPC) plus simple set theory. OCL statements are constructed in four parts: a context that defines the limited situation in which the statement is valid, a property that represents some characteristics of the context, an operation that manipulates or qualifies a property, and keywords that are used to specify conditional expressions.

Four types of constraints can be specified by OCL. They are: invariants, pre/post-conditions and guards. Guard is a constraint that must be true before a state transition fires.

Program 2.3 shows the code fragment of the *purchase* method and invariants specified in OCL.

Program 2.3 Code Fragment of Auctioneer Purchase Method in OCL

```
context: Auctioneer
inv: repository->notEmpty
inv: forAll(item | repository->includes(item) implies (item.minPrice >= 0 and item.minPrice <=
item.maxPrice)
 ...
context: Auctioneer::purchase(item: Item, finalPrice: Real): String
pre: item <> null
pre: self.repository->includes(item)
pre: finalPrice >= item.minPrice
post: result = item.name
```

Similar to JML and Spec#, OCL provides expressions that have neither the ambiguities of natural language nor the inherent difficulty of using complex mathematics. By only allowing assertions to use pure methods, JML and OCL have no side-effects,

because they describe *what* rather than *how*. Spec#, however, seeks to alter the underlying programming language. For example, Spec# has introduced field initialisers and *expose* blocks.

Different from JML and Spec#, OCL is language-independent. OCL is also a navigation language for UML graph-based models. However, OCL is more complicated and its presentation is not modular.

2.3.2.4 Alloy

Alloy [87] is a first-order declarative language based on sets and relations. It is strongly typed and assumes a universe of atoms partitioned into subsets, each of which is associated with a basic type. An Alloy specification is a sequence of paragraphs of two types: signatures used for constructing new types, and a variety of formula paragraphs used to record constraints.

One can use Alloy modules to specify component interfaces, in which *fact* statements can be used to write invariants describing properties of states, and pre/post-conditions can be expressed as states before/after state transitions. An example is shown in Program 2.4: the pre-condition is that the bidder's state changes to *win*, and the product becomes *engaged*; and the post-condition is the product is *sold*.

Program 2.4 Code Fragment of Auctioneer Purchase Method in Alloy

```
pred purchase (ps, ps': ProductStatus, bs, bs': BidderStatus) {
  bs = win && ps = engaged => ps' = sold && bs' = bs
}
```

Compared to OCL, Alloy is more succinct and expressive. OCL supports complex data types, but lacks tools support. Research tools such as USE [70] exist, but most only support a subset of the full OCL language, while Alloy Analyser implements the complete Alloy language. Alloy has structuring mechanisms to allow reuse of model

fragments, but OCL does not. Furthermore, Alloy has the power to specify temporal properties of software systems by defining states.

The Alloy Analyser creates a boolean satisfaction formula from an Alloy model, and assigns a scope to the formula as the first order logic is undecidable. The analysis done by SAT solvers determines if an instance exists for the formula within the scope (the number of elements in each domain set). If one does exist, then the Alloy model is consistent. The tool can also be used to look for theorem (assertion) counterexamples that indicate model inconsistency. Failing to find an instance of a formula does not necessarily indicate that the model is inconsistent; it may simply have an instance at a larger scope. Similarly, failing to find a counterexample to an assertion does not mean that the assertion is consistent; a counterexample may exist at a larger scope.

2.3.2.5 The Other Approaches

Other formal specification techniques, including those based on general mathematical syntax and semantics, such as Z [92], VDM [71] and Larch [39], are also able to present a sufficiently precise and complete understanding of components. However, these specification languages are too difficult for normal practitioners to use.

2.3.3 Protocol Level

Interface specifications at the protocol level specify the contracts on the interactions among components, viz., the ordering between exchanged messages and blocking conditions. Thus the temporal properties of component interfaces, which the semantic level specifications cannot describe, need to be captured.

There have been a number of efforts in introducing temporal aspects into component interface specifications. These approaches are based on Finite State Machines (FSM) [36, 52, 120, 131, 150, 151], temporal logic [4, 11, 56, 79, 90, 98], process algebras [8, 28], Message Sequence Chart (MSC) [58, 73], Petri-net [17], etc.

2.3.3.1 Finite State Machines

Using Finite State Machines (FSM) to model the protocol information has been suggested by [120] and [151]. Protocol specifications can be used to generate adaptors between components [150]. Cho [36] presents a specification technique that can identify component interactions and serve as the basis for automatically generating test cases for the integration of components. In such an approach, one needs to write two levels of specifications for one component: interface specification (semantics) and protocol specification (interaction). The interface specification is based on OCL [77], while the protocol specification is built on FSM. Thus how the other components would interact with the component can be captured. Reussner [131] defines the interface of a component as consisting of two protocols: the protocol defining the call sequences to offered services and the protocol describing the call sequences to external component services. These two protocols are defined by two different automata. The main benefit of this enhancement is the relatively low complexity of algorithms for checking the equivalence and substitutability, and for computing the adaptation [131].

Interface automata [52] is also able to capture both input assumptions about the call sequences to offered services, and output guarantees about the call sequences to external component services. The formalism supports automatic compatibility checks between interface models, and thus constitutes a type system for component interaction. Unlike the traditional uses of automata, interface automata is based on an optimistic approach to composition, and on an alternating approach to design refinement. CHIC [32] is a modular verifier for behavioural compatibility checking built on interface automata.

Generally speaking, this type of approach writes a protocol specification as a set of abstract states, and the execution of the component services is modelled as the transitions among the states. Since the abstract states do not belong to the signatures of any component interfaces, this style of specification is difficult to understand from the user's perspective. For the component developers, it is also impractical to ask

them to write full descriptions of the component interaction protocols [80].

2.3.3.2 Temporal Logic

Temporal Logic has been used to describe the temporal aspects of component behaviour. The languages [4, 98] based on temporal logic can specify both intra-component and inter-component call ordering constraints. This allows one to combine component properties and architectural properties to reason about the system. Some constructs provided by these temporal logic languages can be used to organise specifications in a hierarchical way, which is more suitable for the reasoning. The proof calculus associated with the languages allows people to prove properties effectively, taking advantage of the structure of the specifications.

Reactive modules [11] is a formal model for concurrent systems. The model represents synchronous and asynchronous components in a uniform framework that supports compositional (assume/guarantee) and hierarchical (stepwise refinement) design and verification. The abstraction operator, which may turn an asynchronous system into a synchronous one by collapsing consecutive steps into a single step, allows one to describe systems at various levels of temporal detail. The given below is the example showing that the method *purchase* should be called after the method *bid* is called and the bidder wins the bidding (see Program 2.5).

Program 2.5 Code Fragment in Reactive Modules

```
  ...
update
  [] bs = logged_in & ps = available & bid? -> bs' := win; ps' := engaged
  [] bs = win & ps = engaged & purchase? -> ps' := sold
  ...
```

In [79, 90], temporal operators, such as *before*, *until*, etc., are defined to specify

constraints on component interactions. In these approaches, the temporal operators are about timing relationship of actions, rather than truth relationship of temporal logic formulas. In order to reduce the difficulty of using formal specification languages, the specification patterns [56] are introduced to enable the transfer of experience between practitioners by providing a set of commonly occurring properties and examples of how these properties map into specific specification languages. The temporal operators are used in writing these specification patterns.

2.3.3.3 Process Algebras

Another class of approaches bases the specification of component interaction protocols on the use of Process Algebras, such as *Communicating Sequential Processes (CSP)* and *π-calculus*. Canal et al. [28] propose the use of the π-calculus for the specification of software architectures. This permits the analysis of the specifications for bisimilarity, deadlock and other interesting properties. Canal et al. [29] also extend CORBA IDL with π-calculus for describing object service protocols, aimed at the automated checking of protocol interoperability between CORBA objects. Thus formal specifications in π-calculus are incorporated into the component descriptions. However, the π-calculus is a low-level notation. When specifying large systems, it would be difficult to use.

Some Architectural Description Languages (ADL) include the descriptions of the protocols that determine access to the components. The protocol descriptions derive from process algebras. For example, Allen and Garlan define architectural connectors as explicit semantic entities in Wright [8]. These formal connectors are specified as a collection of protocols that characterise each of the participant roles in an interaction and how these roles interact. The underlying formal semantics based on CSP makes it possible to check architectural compatibility in a way analogous to checking types in programming languages.

2.3.3.4 Message Sequence Chart

Message Sequence Chart (MSC) has become popular in software development by its visual representation, depicting the involved processes as vertical lines, and each message as an arrow between the source and the target processes, according to their occurrence order. MSCs can also serve as a specification and reasoning technique for the composition of systems from components. In [58], MSCs express global coordination properties as well as requirements on individual components for their correct participation in an interaction pattern. The paper defines a decompositional proof rule based on MSCs, and it suggests a composition operator for MSC specifications of the retrieved components that are designed independently of each other. In the paper [73], the MSC connector concept is introduced. The MSC connector concept makes it possible to model component-based systems by means of MSCs, in which the MSC connector concept has been applied to a protocol specification. However, one needs to separately study the expressiveness of MSC languages, and adapt the validation algorithms [67].

2.3.3.5 The Other Approaches

Bastide et al. [17] use *Petri Nets* to specify the behaviour of CORBA objects, providing full operational semantics. However, since the semantics of the behavior of operations and the interaction protocols are defined altogether, the specification does not distinguish the internal semantics and the external behaviour of the component, thus it is difficult for the user to understand how the component will interact with others.

Plasil and Visnovsky write component behavioural protocols using a notation similar to regular expressions [126] that is easy to read. Based on such a protocol definition, the introduction of bounded component behaviour and protocol conformance relation makes it possible to verify the adherence of a component's implementation to its specification at run time, while the correctness of refining the specification can

be verified at design time.

One can also use UML collaboration, sequence and state diagrams [21] to semi-formally describe the component interactions. However, collaboration and sequence diagrams can describe only traces of execution, they cannot be used for a complex description of component behaviour. Although state diagrams have the same expressive power as regular expressions, there is no support in UML for combining state diagrams. Moreover, UML lacks tools support to check consistency of interaction models.

Using ADLs is another way to specify component interactions [41], and ADLs are normally easy to understand and use. However, ADLs have been mainly focusing on early stages of development, they are not suitable for specifying components, even though they share some of the component concepts and their scope is complimentary to component models [47]. ADLs differ from the above approaches by their explicit focus on connectors and architectural configurations. When describing component interactions, ADLs cannot specify the protocol by which those operations must be invoked, thus some extension work is needed [8, 28] to overcome this difficulty.

2.3.4 Specifying and Predicting Quality Properties

Quality property is also called Quality of Service (QoS) or non-functional property. ISO reference model for QoS [60] defines the concepts of QoS characteristics, QoS contracts and QoS capabilities, as well as a basic architecture that are basic elements of QoS specification.

Examples of QoS-enabled modeling languages have QoS Modelling Language (QML) [63] and Component Quality Modeling Language (CQML) [1]. These languages support describing user-defined QoS categories and characteristics, quality contracts and quality bindings. But they do not provide support to optimise the resource allocation, or evaluate the levels of quality provided.

Some work has extended IDL to support QoS, such as Contract Description Lan-

guage (CDL) and Quality Interface Description Language (QIDL) [103]. CDL and QIDL can be used for the automatic generation of stubs and skeletons that support the management of some basic QoS functions and specification of QoS attributes. However, they do not provide support for the description of user-guided QoS attributes.

Recent research interest in assembly prediction focuses on predicting quality properties of component assemblies prior to actually acquiring the components. Some work has been done on performance [34], latency [84], reliability [136], etc.

The prediction has two prerequisites: first, all the components integrated in the assembly should be certified by trusted agents or organisations [145] using some techniques to generate component trustworthiness. The motivation for component certification is that there is a causal link between a component's properties and the properties of the assembly including the component [48]. If enough is known about the components selected for assembly then it may be possible to predict the properties of the final assembled system.

Second, a reasoning framework is required to make a determination if the assembly of those components is well formed with respect to the rules dictated by the reasoning framework. If the assembly is well formed, then the reasoning framework generates a prediction.

2.4 Component Storage and Retrieval

As the complexity of components and the size of component repositories increase, a clear classification scheme of components and a well-designed structure of component repositories can make components easier to locate and retrieve. Therefore, 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 [116]: text-based, lexical descriptor-based, and formal specification-based

encoding and retrieval.

2.4.1 Text-based Encoding and Retrieval

Systems applying the text-based encoding and retrieval method [61] describe the functionality of components in a natural language. The retrieval is based on the words or strings appearing in the description, which usually do not carry much semantic information. As indicated in [116], the text-based method is easy to use, but imprecise.

2.4.2 Lexical Descriptor-based Encoding and Retrieval

Retrieval systems based on lexical descriptor encoding assign a set of previously defined key phrases (lexical descriptors) to classify software components. Domain analysis must be performed first to identify and to determine the key phrases [129]. Some classification techniques that have been used include enumerated [62], keyword [86], faceted [51, 111, 130, 144] and hypertext [44].

An *enumerated* classification scheme generates a hierarchical structure of software components, while a *faceted* scheme uses several facets, and each facet contains a keyword to describe a software component. The faceted scheme, drawn from library science, can describe the attributes of a component more precisely and more flexibly than the enumerated scheme. *Hypertext* provides a means to link together all of the related work products, such as design models, source code, tests, manual and other documentation, into a conceptual entity [44]. This link provides ready access to all work products within the entity, as well as to related or similar entities. Multiple classification schemes are supported to provide a number of different ways to browse and search through a repository. The main problem of the lexical descriptor-based encoding and retrieval method, however, is that an agreed vocabulary has to be developed and component users have to be familiar with this vocabulary [116].

One way to overcome this limitation is to define a natural language user interface [68, 137], by which both user queries and software component descriptions can be expressed in a natural language, such as English. Then both user queries and software component descriptions can be analysed and formalised into the internal representation, in canonical forms. Then the matching is based on computing the closeness of the query and the software component description, which is the distance of the two canonical forms. A public domain lexicon or domain ontology is used to get lexical information for both the query classification and the component classifications.

The retrieval techniques based on similarity analysis can provide good retrieval effectiveness through partial matching of descriptions, processing of synonyms, generalisations and specialisations of terms and considering the syntactic and semantic information available in the descriptors of software artefacts [69]. One can retrieve components by applying fuzzy logic to compute proximity between a user query and a component specification, which are tree-structured models constructed from their respective XML files [107].

However, the component semantics provided by these approaches does not contain precise behavioural properties of components. A further examination of the retrieved components is needed.

2.4.3 Formal Specification-based Encoding and Retrieval

In order to obtain more precise component-retrieval results, component behaviour must be considered. The behaviours of components are described by both semantic and protocol aspects of interface specifications. Hence, formal methods are used to capture not only the terms appearing in the interface operation signatures [152], but the meanings of these operations and the orders of these operations being invoked. Any desired relations between a user-expected component and a previously existing component in the repository, such as refinement and matching, is expressed by a logical formula composed from the behaviour specifications of both [59]. Component

retrieval becomes possible by checking the validity of the formula by an automated theorem prover, and only if the prover succeeds, the relation is considered to be established.

The majority of such work uses first-order logic [35, 117, 125, 153] as the underlying formal notation to write the specifications of component interfaces and user queries. The matching is presented on the basis of interface operations, checking how exactly the pre-conditions and post-conditions of the operations and queries must match. The matching criteria can range from exact match to relaxed match [125, 153]. For relaxed matches, it may be possible to identify the type of adaptation necessary for retrieved components as is the case in the REBOUND project [125].

Modules are specified by grouping individual operations and a query matches a module if all query requirements are matched against the module specification [82, 153]. When modules are specified by state transitions, the matching algorithm has been given by [81].

The limitation of formal specification-based encoding and retrieval is that one has to write formal specifications that are difficult for normal users who do not have good mathematical knowledge, and will become much more difficult as the size and complexity of components increase.

2.5 Component Evaluation

After the components have been matched and retrieved, they need to be evaluated in order to decide the best fit. Thus, component evaluation is a decision aid. Some researchers view component evaluation as software engineering discipline that starts from requirement analysis and is driven by models that include the product descriptions and evaluation criteria, while others focus on the particular methods used to make decisions, such as multiple-attribute utility [47] or component rank [108].

2.5.1 Evaluation Processes

Evaluating components needs to follow a process that at the most abstract level involves three large-scale tasks:

1. Plan the evaluation: define the problem, define the outcomes of the evaluation, assess the decision risk, identify the decision-maker, identify resources, identify the stakeholders, identify the alternatives, and assess the nature of the evaluation context.

2. Design the evaluation instrument: specify the evaluation criteria, build a priority structure, define the assessment approach, select an aggregation technique, and select assessment techniques.

3. Apply the evaluation instrument: obtain products, build a measurement infrastructure, perform assessment, aggregate data and form recommendations.

Most existing approaches, such as [37, 94, 105, 118, 123], follow the above process. Meanwhile the process is driven by different models: OTSO [94] builds the model with definition of evaluation criteria from various sources, such as initial requirements, analysis report, architectural design, etc., and the evaluation is performed upon these criteria. CAP [123] by Siemens has a similar model. In IusWare [118], an evaluation model is formalised to facilitate verification and validation activities by checking the consistency of the model and its components. When applying the evaluation model, attributes of products are measured. These measures are transformed into values on criteria, and these values are then aggregated to form a recommendation. PORE [105] defines three models (requirement, product and compliance) to achieve a compromise between customer requirements and product features. Similarly, CARE [37] introduces the world model (describing the environment), the system model (representing the system capabilities: functional and nonfunctional), and the interface model (mapping system goals and requirements to component goals and specifications). Based on these three models, component evaluation is conducted.

All the above evaluation approaches set requirement analysis as the starting point, and with the identification of components, the system requirements and architecture are refined. For example, OTSO [94] is conducted during the requirements specification phase; PORE [105] particularly emphasises the iterative and parallel process of requirement acquisition and component evaluation. CARE project [38] uses component searching and matching process to support the reciprocal refinements of the stakeholders requirements.

Despite the similarities, the above approaches also have their own different features. PORE [105] and CARE [37] stress that the requirement should be "component-aware", that is to say, requirement specifications must be sufficient to enable effective product selection rather than complete with respect to the user's needs. IusWare [118] adds more formalism flavor to check different evaluation models created with more freedom in various situations. CAP [123] packages all data that evolved from performing the CAP activities into a repository for reuse purposes in future projects. CARE [37] is designed as a knowledge-based approach, in which agents (either human or software) are created to fulfil different sub-goals, based on which final goal can be achieved. Another agent-based component evaluation method [147] models different players as either cooperating or competing expert agents. The administrator agent collects and combines the knowledge and decisions from those expert agents in different areas to support component selection.

2.5.2 Particular Methods

The particular methods for evaluating components can help to identify alternatives among discrete choices based on some criteria. Multiple-attribute utility is a kind of multiple-criteria evaluation method [47]. It is based on a preference structure that includes the factors that govern the decision and judgements about these factors. Meanwhile it uses an aggregation technique to generate interpretation of a classifying or ranking model.

Analytic Hierarchy Process (AHP) is a multiple-criteria decision-making method that uses hierarchic or network structures to represent a decision problem and then develops priorities for the alternatives [133]. AHP attempts to resolve conflicts and analyse judgements through a process of determining the relative importance of a set of criteria. BAREMO [141] is a typical application of the AHP model to help software engineers choose the appropriate components for a project. However, AHP assumes that criteria are independent. This will result in compensations in scores and getting unworkable combinations of values, such as .NET with Linux. If considering criteria dependencies to be important, one may look to the field of Artificial Intelligence for applicable techniques, for example, fuzzy logic [50, 109].

From a non-technical perspective, components are evaluated based on such criteria as user's familiarity, vendor's reputation, project budget, etc. In this case, component selection can be viewed as general product selection that is driven by non-technical models, for example, microeconomic model [25]. These models are often calculated relying on some intelligent techniques [24], such as data mining, knowledge discovery, etc.

Sun [138] discusses how Case-Based Reasoning (CBR) can support electronic commerce product selection in his Ph.D. thesis. CBR is an artificial intelligence technology that uses past occurrences to locate problem solutions. In CBR, the primary knowledge source is not generalised rules, but a memory of stored cases recording specific prior episodes. Product selection using CBR also needs dialogue, retrieve, customisation and product representation for four phases [18]. Another example of applying CBR in finding the best case is WordNet [72], in which UML class diagrams are retrieved. Chung [38] proposed to use a hybrid of CBR and AHP in order to gain advantages from both techniques: AHP is good for prioritising the importance of the components within each of the component sets, CBR is useful for clustering the various evaluation criteria and similarity measures collectively for each component set.

In the component ranking model [108], a collection of software components are represented as a weighted directed graph, in which nodes correspond to the components and edges correspond to the usage relations. By analysing actual use relations among the components and propagating the significance through the use relations, component ranks can be calculated. According to the authors, high ranks are given to those generic components that are used by most applications.

Components can also be ranked by some non-functional properties, such as performance. A systematic approach to find the feasible combinations of alternatives and to rank them based on predicted performance is described in [16]. The paper defines components in layered queuing models for software performance.

2.6 Existing Component Selection Systems

Some component selection systems have been implemented for public use. For example, ComponentSource [42] and Topcoder [142] provide component repositories and search engines. However, as mentioned before, both of them only support searching by free text. Thus the results received often contain many irrelevant items.

RetrievalJ is a Javabean component-retrieval system based on Directed Replaceability Distance (DRD) theory [148]. When retrieving components, three types of similarities are compared: structure, behaviour and granuality. However, the behaviour in DRD is expressed by the return value of method, type of value changed and readable properties. Since it does not put constraints on the methods, components retrieved may still contain the behaviour that is incompatible to user requirement. RetrievalJ only supports searching Javabean components.

Knowledge Based Automated Component Ensemble Evaluation (K-BACEE) is an expert system for component selection, which uses keyword search from manifest criteria to yield working set components to form ensembles [22]. K-BACEE is built on the belief that systems are not built from individual components, but compo-

nent ensembles. A component ensemble [83] is a set of technologies, products and components that interact to provide some useful behavior. In K-BACEE, ensemble evaluation is supported by the cooperation of component specifications, integration rules and patterns. The user inputs a manifest (SRS, System Requirements Specification) and obtains an ensembles list with ranked value. K-BACEE is implemented towards EJB only, and the authors think that it would be expensive to make it fully support all types of components due to the difficulty of extending its knowledge base.

IBM has built a Reusable Asset Specification (RAS) repository for workgroups [23]. The repository supports searching and browsing of assets using the RAS standard repository service interface. A variety of ways of retrieving information about the assets can be used, including viewing of the documentation, viewing of feedback and importation of the complete asset. Meanwhile the asset authors can publish assets, create and organise the logical view of the assets in the repository. Reusable Asset Specification is now an open standard that can be used to package any reusable software assets. RAS describes assets as part of asset-based development (ABD), which complements the Model Driven Architecture (MDA) by describing asset production, asset consumption and asset management. A RAS of an asset may contain the following parts [78]: the overview part contains a collection of human-readable artefacts such as documents describing the problem that the asset solves as well as the intent and motivation; the classification part contains metatags (or descriptors, or qualifiers) that describe the assets as a unit. They are used to group, store, search and retrieve assets as units. This section also has a description of the Context(s) in which, or for which, the asset may be applied; the usage part contains key information about how to apply the asset, such as the problem context (the reuse intention) for which the asset was created and the variability points through which the asset can be customised for a particular reuse situation; the solution part contains the artefacts that make up the solution. These artefacts include requirements, designs, models, code, tests, deployment scripts, and so on. Finally, the structure of the asset is defined

using UML within a separate UML description section. However, using RAS cannot yet precisely specify component behaviour, so the users have to test the components retrieved from the repository by themselves.

2.7 Component Trader's Involvement

Recently researchers realise that selecting components is not only the component user's business, but also involves other roles such as component developers and brokers. As one of the mainstream CBD frameworks, Select Perspective [6] has created a collaborative model to select COTS components.

ComponentXchange [143], a web-based software component exchange, acts as an online broker between component users and component developers, allowing component storage and retrieval through a licensing service. The limitation of this approach is that components should be licence-aware, and developers have to be involved in the transaction. Iribarne et al. [85] integrate a component trader into a spiral methodology for CBSD by using a series of XML-based templates to document components, services and queries. This work, however, does not cover semantic trading.

2.8 Summary

Most literature on component selection is written from a user's perspective. Therefore, even though components and user requirements can be precisely specified, it is likely that users cannot find any suitable components if the components are from external sources.

Meanwhile, for component selection, even though people have understood the importance of the collaboration between the users and the developers, the recent approaches have only presented general frameworks with an informal way to establish the communication. Thus the misunderstandings on components and user require-

ments remain existing. Moreover, there are no tools to support the collaboration process.

In the next chapter, our approach is proposed mainly towards overcoming these two problems.

Chapter 3

The Framework of Selecting Components based on Collaboration

3.1 Introduction

In Component-Based Development (CBD), component selection is a critical activity that should be started from the phase of requirement analysis and continue throughout the whole system development life cycle. The software process for users to identify, match, retrieve, evaluate and finally choose components is driven by the models that include component descriptions and user requirements for the components.

However, many lessons learned from [122] have told us that component selection is still hard. Although there are many different developers providing a large variety of different types of components, it is difficult to identify the required one simply based on the information the developers provide. This is because the descriptions of components are mostly imprecise, while the internal workings of the components are invisible to the users. On the other side, the user requirements are also expressed ambiguously and changeably. The gap between the components provided and the

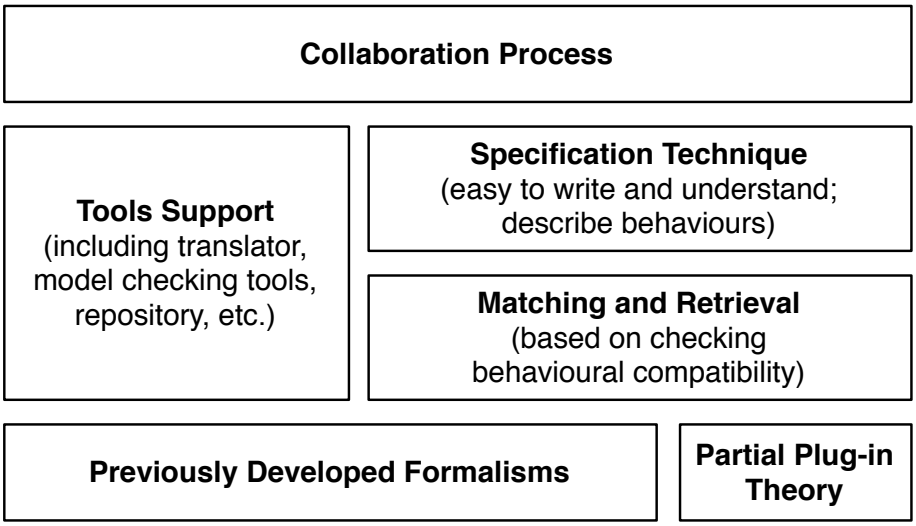

Figure 3.1: Structure of the Proposed Framework

components required almost always exists when the components from external sources are targeted. This gap is difficult to reduce due to the nature of CBD, which has two separate processes of developing components and systems. However, there is little connection between these two processes. This increases the misunderstanding between component users and component developers.

Enhancing the communication between component users and developers can reduce the gap and remove the misunderstanding, so that the required components can be found effectively. This is the purpose of building our collaboration framework.

This chapter first overviews the proposed framework, then it explains in detail each module of the framework.

3.2 The Proposed Framework

The structure of the proposed framework can be depicted as shown in Figure 3.1.

In the framework, for collaboration purposes, a common language is used to specify

both components and the user requirements for the components. Thus component users and developers can communicate with each other by exchanging requirement and component specifications. One can use formal methods to specify components and user requirements at an abstract, but sufficiently precise and complete level. However, it is currently unrealistic to expect normal component users to write formal specifications directly, especially when components are becoming larger and more complex.

Thus we need to keep the specifications easy to write and understand. Since component behaviour is the most important aspect when considering to use the component in the target system, our specification language should be able to describe component behaviours rather than interface syntax. Meanwhile there is underlying formalism to make the specifications precise, and this formalism is hidden from normal component users. In our solution, we reuse those previously developed formalisms from other research projects to check component behavioural compatibility. The advantages of doing this is their accompanying tools can be reused as well in our implementation. We only need to build a bridge from our specification language to those formalisms.

Furthermore, we have developed the theory of component partial plug-in based on Interface Automata [52]. This formalism allows the components to be selected even when they only partially fulfil the user requirements. The situation when a component can completely satisfy user requirement can be regarded as a special case of the theory. However, it is not included in our implementation due to the absence of its model checking tools. The detailed elaboration of the component partial plug-in theory can be found in [95].

If an agreement on how to use the component can be made between the user and the component developer, the component is regarded as meeting the requirements and can be selected. Tools are provided to make the selecting process easy and automatic. Since user requirements are also viewed as components in our approach,

the component selection is to check the component behavioural compatibility with user requirement by using those previously developed model checking tools. Other necessary tools include the translator, the organised repository, the user interface, etc.

Evaluating the retrieved components is not the emphasis of our solution, even though it is a necessary step in practice. This is because component evaluation needs to consider many non-technical aspects that are not the focus of our research.

3.3 The Collaboration Process

As there exist misunderstandings about the component descriptions and user requirements for the components, a collaboration between component users and developers is required to clarify the misunderstandings. Moreover, the collaboration can help users to get the required components if developers agree to customise their components for particular user requirements. It is easier for component developers to do that because they have direct knowledge about the internal workings of the components. Some promising technologies can be used on the developer's side to facilitate this customisation, such as software product line [40]. It is in the developer's interest to assist in the process of component selection.

As an effective software development process should describe who does what, how and when [57], borrowing the terms from RUP [97], our collaboration process has the same key concepts:

- Roles: The who
- Artefacts: The what
- Activities: The how
- Workflow: The when

3.3.1 Roles

The collaboration involves two roles: component user and component developer. A component user is the system developer, who selects components from different sources to build the system. A component developer implements the component that can be used by a component user.

3.3.2 Artefacts

The artefacts used in the collaboration process include user requirement specification, which is divided into required behaviour specification and required interface signatures, component specification, which is also divided into component behaviour specification and component interface signatures, as well as other documents. The specifications of user requirements and components can be revised in order to reach an agreement between component users and developers. The revised requirement specifications and component specifications are also artefacts in the collaboration process.

The reason why component behaviours have been separately specified from the interface signatures is that we only focus on matching components by their behaviours. Interface signatures are not important in our approach, however, we include them in the process in order to provide a complete solution of selecting the user-required components.

If component users and developers need more information to make decisions in the collaboration, other documents may be required from each other, such as design document, user manual, etc.

3.3.3 Activities

In the collaboration, component users can perform such activities as searching components, checking behavioural compatibility, requesting modification on component

behaviour, requesting modification on interface signatures, testing components to see whether they satisfy the requirements, and adjusting requirements if necessary.

- *Searching Components* can be done by keywords, or free text. Thus a list of options can be received. The purpose of this activity is to initially screen the components from the repository.

- *Checking Behavioural Compatibility* is to check whether the component behaviour specification meets the user requirement on component behaviour. This can be done by some model checking tools. The purpose of this activity is to further reduce the number of candidate components. Thus those components having the required behaviour can be selected to use in the targeting system.

- *Requesting Modification on Component Behaviour* is to ask the developer to change the component behaviour based on the user requirement. The user will submit the required component behaviour specification to developers. This activity is to ensure that the component selected will have the required behaviour.

- *Requesting Modification on Interface Signatures* is to ask the developer to change the component interface signatures based on the user requirement. The user will submit the required component interface signatures to developers. This activity is to ensure that the component selected can be directly used in the user system, given that it has already had the required behaviour.

- *Testing Components* checks whether the revised components can satisfy the user requirements. The user may check component behavioural compatibility again with the user requirement, or try to integrate the component into the user system. This activity is to ensure that the component selected can be truly used in the targeting system.

- *Adjusting Requirements* is to modify user requirements so that some particular components can be selected.

Developers can agree to modify components to satisfy the user requests on component behaviour or interface signatures, or both.

- *Modifying Component Behaviour* is to make sure that the component can have user-required behaviour after the modification. The developer will give the user a new behavioural specification of the component.

- *Modifying Interface Signatures* is to make sure that the component can have user-required interface syntax after the modification. The developer will give the user a new interface syntactic specification of the component, such as an API (Application Programming Interface) document.

3.3.4 Workflow

The workflow describes groups of activities performed by component users and developers together to select the components that can satisfy both syntactical and behavioural requirements. The workflow in the collaboration process (see Figure 3.2) can be described by the following steps:

1. A user starts the selection of a specific component after the requirement for this components has been formed. The user may first search the component from the repository by keywords, and receive a list of candidates from different developers.

2. For each candidate component, the user checks if they have the required behaviour.

3. If a candidate component can completely meet the requirement, viz., it has both required behaviour and required interface syntax, it is the best choice.

4. If a candidate component has required behaviour but different interface syntax (signatures), the user will negotiate with the developer on modifying the syntax

of the component interface (see Figure 3.2 – a). In this case, the user can send the required interface signatures to the developer for customisation. Thus this component can be used with some modifications.

5. If a candidate component does not have the complete required behaviour, the user will negotiate with the developer on modifying the component behaviour (see Figure 3.2 – b). The developer may agree to do so depending on the cost caused by the customisation. If not, the user may need to consider adjusting the requirement.

6. If no components can be retrieved, the user now should think of adjusting the requirements for the component. The related developers can help this by providing some design documents of the components.

7. Receiving the revised component from the developer, the user will test the component in the targeting system to check whether the component can really be used.

The collaboration on selecting components can be further discussed through an example scenario (see Figure 3.3): a user specifies the requirements for a particular component assembly in which four components interact with each other. The user searches for these components and has found a few candidates for the required components. Component `a` exactly matches the user requirement `1` because it is developed by an internal developer. Thus it can be used without modification. Component `b` has the required behaviour as specified in requirement `2`, however, its interface signatures are not matching with the requirement. This component still can be used because behaviourally it is matched. Then the user sends the developer the request of customising the component with user-specified interface syntax. It would be easy for the developer to do that since the main business logic remains unchanged. Thus component `b` still can be used with customisation by its developer. The user cannot find any suitable components for requirement `3`, however, component `c` is probably the

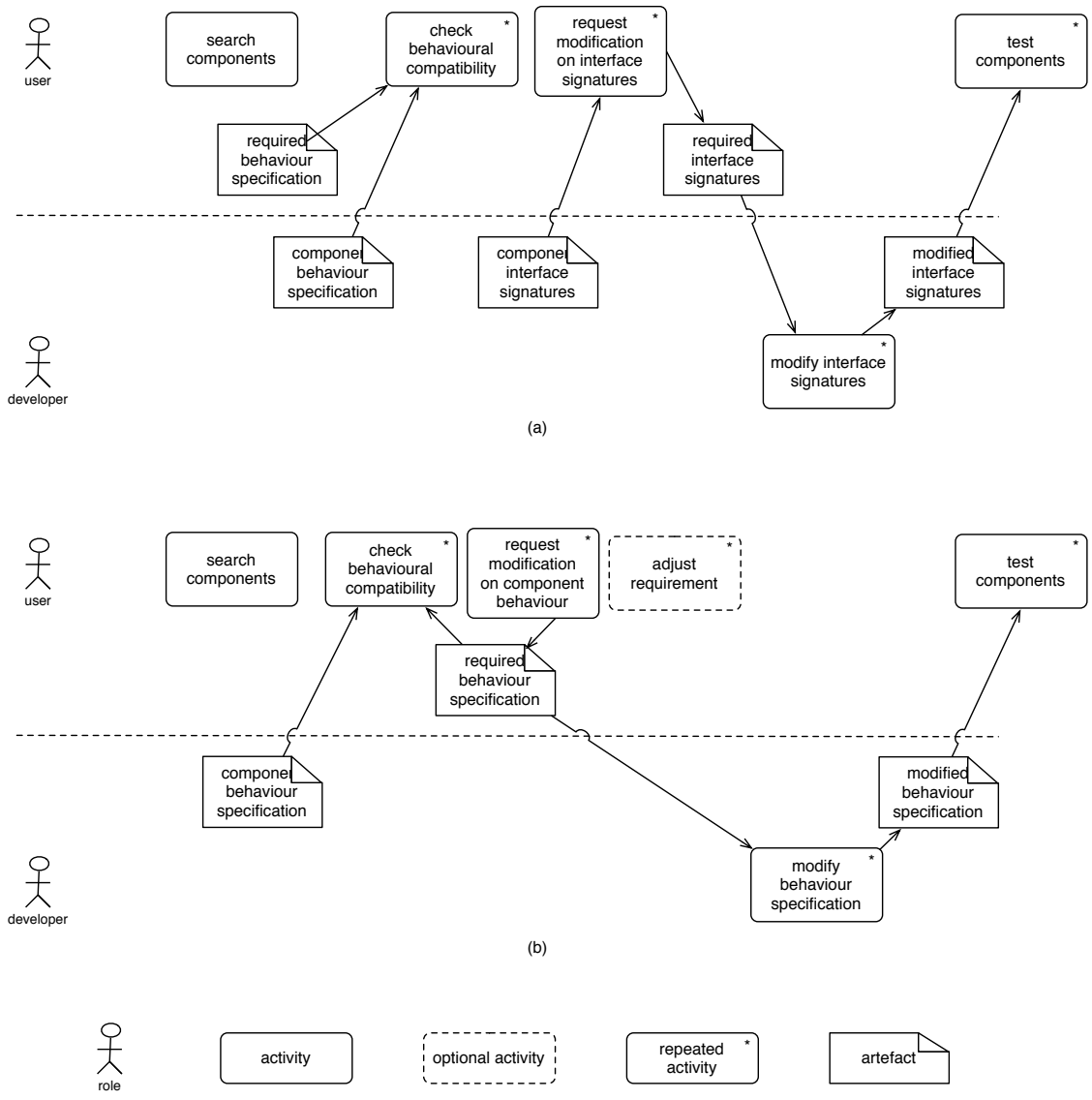

Figure 3.2: The Workflow in the Collaboration Process

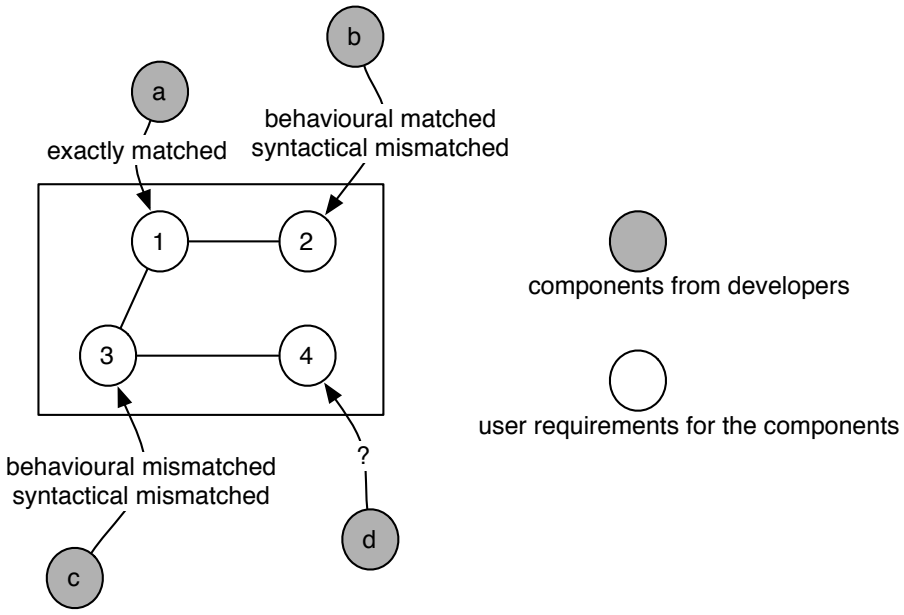

Figure 3.3: A Scenario of the Collaboration

closest one, because it has most of the required behaviour. The developer is asked to change both its internal behaviour and interface signatures. However, the developer may refuse to do that because it is difficult and costly. Thus the user may need to consider changing requirement `3` according to the available components. Component `c` may be used. This depends on the negotiation between the user and the developer. Component `d` interacts with component `c`. If requirement `3` needs changing, requirement `4` may need changing as well, or glue code should be written between them in order to keep requirement `4` unchanged. In this situation, developers of the candidate components can help the user to make decisions by getting involved in the user's project. It is helpful since the developers have a lot of knowledge and experience about the same type of components. For example, if the user needs to write glue code for component `c`, its developer can help the user by providing detailed information on the internal workings of the component.

A new search will be performed if the requirement for the component has been changed. The same workflow will be followed until the required component is found. Thus the collaboration is an iterative and incremental process.

The collaboration process provides a framework of selecting required components. Thus a variety of different methods and tools can be used in this framework. In order to prove the concept we develop our own method and tools to support the collaboration.

3.4 The Specification Language and Matching Technique

A common language is used as the means of communication for component users and developers. Both components and user requirements for the components can be described by this language. Such a language should have familiar programming language constructs, with easy to understand semantics. We developed Simple Component Interface Language (SCIL) for proof of the concept purpose. SCIL is derived from formal descriptions such as Interface Automata [52]. It can also be viewed as a cut-down version of a normal programming language that aims to support formal specification of component interfaces and requirements. SCIL only focuses on the high-level behaviours of components; it cannot give detailed information on component interface signatures.

Matching components against the user requirement is to check the behavioural compatibility between them, so component SCIL specifications are first retrieved from the repository. That is to say, in our approach, if a component SCIL specification is checked compatible with the SCIL specification of user requirement, the component is regarded as matched and having required behaviour. Then the detailed component information, such as a trial version of the component with API document, can be retrieved from the repository for further evaluation.

The behavioural compatibility checking is performed thus: SCIL specifications are translated to a variety of models that can be checked by their supporting tools. These models are written by different modelling languages that are the inputs to those tools. Therefore, SCIL and its translator will allow the users to gain access to a number of tools based on formal methods. At the moment, SCIL only supports complete component plug-in, but it has the potential to support partial plug-in as long as there are a formal language and its tool that can compute partial component plug-in model [95]. The advantage of such an approach is that it becomes possible to use a couple of formal tools together to solve one problem.

3.5 The Tools and Formalism Support

The tools that can support the collaborative process of selecting required components include a general search engine by which users can narrow the search for candidate components by keywords, a SCIL translator that translates SCIL to existing modelling languages, a web platform through which users interact with the component selection system, a component repository with a specific classification scheme, and the existing model checking tools.

We reuse and integrate those previously developed model checking tools in our framework. These tools are built based on different formalisms that can be employed for checking different properties, but hidden from users by a single and uniform presentation, such as SCIL in our implementation. These supporting formalisms may rely on different mathematical theories, such as finite state machine (FSM), set theory, temporal logic, etc. Thus they provide different ways to check the compatibility of two components. For example, two FSMs can be combined to check whether there are unreachable states in the combined FSM. If no such states exist, the two modules are compatible. If two components are specified in temporal logic, the compatibility check is to see whether there exists any temporal conflict of calling methods. In gen-

eral, if a formalism can be translated from SCIL, and there is a tool implemented for checking the formalism, this formalism can be reused in our framework.

3.6 Summary

We compare our approach of searching components by behavioural specifications with the traditional approach of searching by keywords. The comparison of two processes is shown in Figure 3.4.

Although searching by keywords is easy to use, results often contain many irrelevant items. The traditional way is completely manual. Users have to examine candidate components one by one, getting rid of the obvious irrelevant ones. Then users need to download the user manual documents or the trial versions of those remaining candidates, and try one by one.

Reading SCIL specifications is faster because SCIL specifications describe high-level behaviour of the component. SCIL specification matching is semi-automatically done (as can be seen in the case study, it involves some manual work to make the requirement specification consistent with the component specifications). The components found by our approach can be ensured that they will have the required behaviour that is compatible to the user-targeting environment. This is due to the collaboration from developers who are willing to customise their components according to specific user requirements.

A disadvantage of our approach is that the developer needs to write a specification corresponding to the component. However, if the specification can be the cut-down version of the component itself, this is not an major overhead, especially given the above benefits.

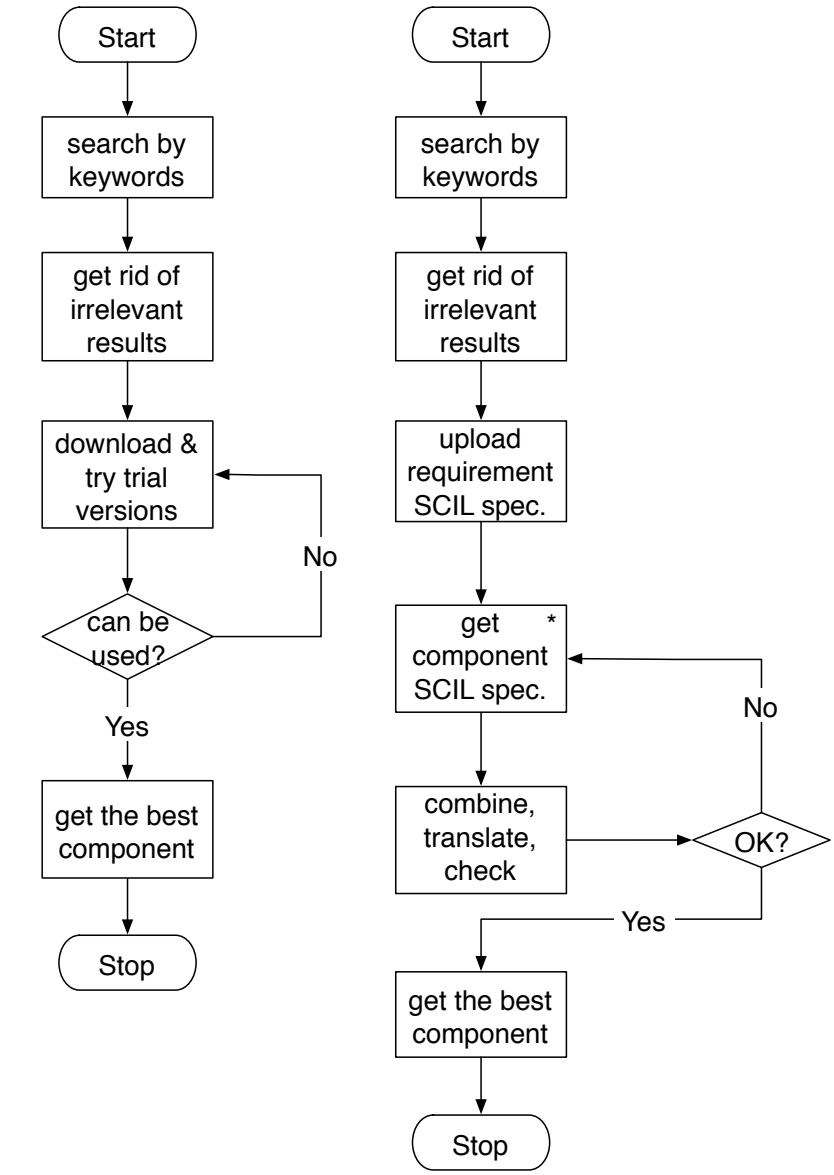

Figure 3.4: The Comparison of Two Component Searching Processes

Chapter 4

Simple Component Interface Language

4.1 Introduction

As described in the previous chapter, our framework for selecting components needs an intermediate language that can be used to specify both components and user requirements for the components. Thus this language can be used by both component users and developers to communicate when exchanging requirements and components.

Furthermore, this intermediate language should be able to describe component behaviours through interfaces. In our framework, in order to check behavioural compatibility of components with their requirements, this intermediate language needs to be translated to other previously existing formal languages at first. Thus how to build the mapping between the two languages needs to be considered when designing such an intermediate language.

In order to let those who are not mathematical professionals use this intermediate language, the language should have an easy-to-understand syntax while hide formal details of specifications written by the language.

In this thesis, Simple Component Interface Language (SCIL) is such an interme-

diate language, developed as the proof of the concept.

4.2 Design

Based on the design purpose and goal, the principles for designing SCIL are as follows:

- SCIL needs to be precise, as it is used for communication purposes.
- SCIL needs to be simple, and easy to understand, as it is used by normal users.
- SCIL needs to be able to describe component behaviours, as components in our framework are retrieved by behavioural properties.
- SCIL needs to express constraints and define properties, since it acts as the bridge to formal modelling languages for checking purposes.
- SCIL descriptions need to be composed when checking components against user requirements.

Since we are providing a framework, it is possible to use some other existing Behavioural Interface Specification Languages (BISL) [149] in the framework, such as JML [101] and Spec# [15], as long as the corresponding translators are provided. JML and Spec# are much richer languages, but verifying joint behaviour of components requires the use of sophisticated theorem provers [15, 127]. One may use Interface Definition Language (IDL) with temporal logic to describe component behaviours [90]. However, the component specifications written in IDL are concrete at the interface signature level. One cannot retrieve these components with different signature requirements, even though the components actually have the required behaviours.

We aim to write more abstract specifications about component interfaces. SCIL does not focus on interface signatures, but on the abstraction of business logic of components, i.e., *how* and *when* to invoke component services. The advantage of such

an approach is that even when the interface signatures of a component fail matching with user requirement, it is still possible to choose this component if the component's behaviour that is ruled by the invariants and constraints meets the user requirement. In order to finally integrate the component into the user's system, the component user can negotiate with the component developer on adjusting the interface signatures.

Next we are going to introduce SCIL through an example. We draw the auctioneer component from [90] (originally from [29]) with adaptation to show how to use SCIL to write specifications for components and user requirements.

In an auction system, there are three types of components: sellers, bidders and the auctioneer component standing between them. The auction system works following this way: sellers first notify the system what products are available to sell, then bidders can log into the system and bid for the products. If a user wins a bidding, the user should pay for the product. After the user logs out of the system, the transaction is terminated. Thus for the auctioneer component, the statuses of bidders and products should be remembered. For products, it can have these states: when a product is not available for auction, its state is *not_available*; after a seller makes it available, the product's state becomes *available*; later on, the state can be *engaged* if the auction is won by someone; after the winner pays for the product, the product's state further becomes *sold*. As for the state of a bidder, it becomes *logged_in* or *logged_out* after the bidder logs into or logs out of the auction system; the bidder's state becomes *win* or *not_win* if the bidder wins or does not win the auction respectively.

The auctioneer component provides interfaces for *login*, *logout*, *bid*, *purchase* and *sell* services. Service *login* and *logout* let the bidder log into or log out of the system respectively. Service *bid* allows the bidder to bid for a product. If the bidder wins the bidding, service *purchase* will be called when the bidder is paying for the product. Service *sell* is only used by the seller to make the product available for the auction.

```
SCIL ::= TYPE VARIABLE SERVICE PROTOCOL SCENARIO PROPERTY
COMPONENT ::= TYPE VARIABLE SERVICE PROTOCOL
REQUIREMENT ::= TYPE VARIABLE ENVIRONMENT_COMPONENT REQUIRED_PROPERTIES
ENVIRONMENT_COMPONENT ::= SERVICE|run PROTOCOL
REQUIRED_PROPERTIES ::= SCENARIO PROPERTY
COMPOSITION ::= ENVIRONMENT_COMPONENT|COMPONENT connects COMPONENT

TYPE ::= int | bool | enumeration | structure | user_defined | deferred
     user_defined ::= identifier is TYPE;
VARIABLE ::= identifier as TYPE
SERVICE ::= INPUT OUTPUT
            assumption(INPUT, OUTPUT) -> guarantee(OUTPUT)
PROTOCOL ::= unary_temporal_operator SERVICE |
             SERVICE binary_temporal_operator SERVICE
SCENARIO ::= {[step] expressions}+
PROPERTY ::= property_pattern
```

Figure 4.1: Abstract Description of SCIL

4.2.1 Syntax

SCIL can be viewed as a cut-down version of a normal programming language, such as Visual Basic. A typical SCIL file consists of these basic definitions (see Figure 4.1): `TYPE`, `VARIABLE`, `SERVICE`, `PROTOCOL`, `SCENARIO` and `PROPERTY`. `TYPE` defines data types, and `VARIABLE` in SCIL must have a data type. `SERVICE` defines what a component can do, and `PROTOCOL` defines the order that the component services can be invoked. `SERVICE` and `PROTOCOL` are used for specifying components. `SCENARIO` presents a sequence of execution steps. `PROPERTY` defines the properties that users require a component to have. Thus `SCENARIO` and `PROPERTY` are used for specifying user requirements for the components.

Next we introduce these syntactic constructs of SCIL in detail.

4.2.1.1 Data Types and Variables

SCIL supports primitive data types: *bool* and *int*. However, for the model checking purpose, the *int* type needs to be specified within a range, such as from 0 to 5. This is because all the model checking tools can only handle finite states.

Moreover, in order to support describing state transition systems, SCIL has enumeration type that can be used to list state values. For example, the auctioneer component uses two enumeration types to describe statuses of products and bidders (*ProductStatus* and *BidderStatus* in Program 4.1).

Program 4.1 Enumeration Type in SCIL

```
type:
  ProductStatus is {not_available, available, engaged, sold};
  BidderStatus is {logged_in, logged_out, win, not_win};
```

One may use structure type to define a class like data type when the type has more than one attribute. The structure type in SCIL is similar to a *struct* definition in C language. For example, a shopping cart can have two attributes (*SCart* in Program 4.2): the capacity and the message that the cart can receive. In the example, the maximum number of items that the cart can contain is assumed as 5. If the payment for the shopping cart is completed successfully, the *succeed* message will be received by the cart, otherwise the *fail* message will be received.

In SCIL a user can define an identifier that represents an existing data type. The user-defined datatype identifier can later be used to declare variables.

Only services can be declared as type *deferred*. The *deferred* services are not defined in the current component, but will be defined later in other components. This is necessary when describing that a component invokes the services from another

Program 4.2 Structured Type in SCIL

```
type:
  SCart is {
    Capacity is (0..5);
    RMessage is {succeed, fail};
  };
```

component.

Every variable used in the specification should be declared to the compiler. The declaration does two things: tells the compiler the variable name, and specifies what type of data the variable will hold.

4.2.1.2 Services and Rules

Service definitions tell what a component can provide. A service in SCIL corresponds to low-level methods in programming languages such as Java. That is, in the actual component, a service may be implemented by a collection of methods. Such service descriptions can be obtained from use case diagrams when designing the component.

For each service, one can define input and output variables, as well as the rules that govern the behaviour of these variables. Variables can be defined as input or output within a service by the keywords *input* or *output*.

Services have rules that define *how* a component can perform its services. These rules are similar to pre/post-conditions [112] (or assumptions/guarantees [52]). That is to say, only when the assumptions are satisfied, the guarantees can be executed. In such rules, assumptions are made by inputs and outputs, and guarantees are represented as a set of assignment statements if the assumptions are satisfied.

In the auctioneer component, take *sell* service as an example. Sellers invoke this service to notify the auction system that their products have become available

(Program 4.3). There are no assumptions on this service. For the *bid* service, the assumptions are that the bidder has already logged in, and the product is still *available*. If both are met, the *bid* service provides two possibilities: the bidding is successful (*win*), or the bidding is not successful (*not_win*). If the bidding is successful, the product status changes from *available* to *engaged*. Even though the bidder does not win the bidding, the product status is still changed to *engaged*, because another bidder has won the bidding (Program 4.3). The two rules of the *bid* service have the same assumptions, but will lead to two different results. This is called non-determinism.

Program 4.3 Services of the Auctioneer Component

```
// ps: the product's status
// bs: the bidder's status

sell {
  // product becomes available
  rule: true -> ps = available;
}

bid {
  rule:
    // the bidder wins the bidding
    bs = logged_in && ps = available -> bs = win && ps = engaged;
    // the bidder does not win the bidding
    bs = logged_in && ps = available -> bs = not_win && ps = engaged;
}
```

4.2.1.3 Protocol

Besides service rules, there are global (in terms of the component) temporal constraints on the component interface. These constraints define the order by which

services should be invoked. These interaction constraints are named protocols [90] between the component and its users.

The reason to include protocol definitions in SCIL is that one needs to specify not only *how* a component performs its services, but also *when* these services can be called. Thus the specification of component behaviour can be more complete.

The protocol in SCIL has two usages. The first allows users to specify temporal properties of services by using temporal operators [90], such as *initially*, *precedes*, *once*, etc. In SCIL, *initially* defines the service be called first, while *precedes* defines the service be called before another. And *once* requires that the service can only be called once.

The second uses protocols to find inconsistencies in service rules. For example, service definition indicates that service A relies on another service B's outputs, but the protocol specifies "A *precedes* B". This is clearly a conflict between these two definitions.

In the auctioneer example, for one product, the component developer allows the *sell* service to occur only once, because one product cannot sell twice. And *sell* must go before all the other services. This is because bidding for a product can only happen after someone wants to sell the product. Service *bid* must be invoked before *purchase*, since paying for the product is after the bidding is finished. Finally, Service *logout* must be used to end the business with the bidder. All these constraints are specified in the protocol part of the SCIL specification via the temporal operators (Program 4.4).

4.2.1.4 Scenarios

Sometimes it is unrealistic and unnecessary to check all the properties of components exhaustively. One may be more interested in checking whether some expected scenarios can be satisfied. Thus scenario-based checking is one of the ways to determine if the selected component meets user requirement.

Program 4.4 Protocol of the Auctioneer Component

protocol:

once sell;

initially sell;

bid *precedes* purchase;

eventually logout;

Scenario definitions in SCIL are similar to sequence diagrams or flow of events in use-case diagrams, viz., sequence of state transitions. In one requirement specification, users can have more than one scenario definition to check.

In the requirement specification for auctioneer component, a scenario named *best_story* is defined to show the state transition to a successful bidding. In the Program 4.5, *r_bs* and *r_ps* are the variables having enumeration types that respectively represent the statuses of the bidder and the product. The *best_story* scenario has six steps of state transition (Program 4.5):

1. the bidder has not logged into the system, and the product is still unavailable.
2. the bidder has not logged into the system, but the product becomes available.
3. the bidder now logs into the system, starts to bid for the available product.
4. the bidder wins the bidding, the product is engaged to the bidder.
5. the bidder pays for the product, so the product is sold.
6. the bidder logs out the system, the auction for this product is ended.

4.2.1.5 Properties

Another way to check whether a component is compatible to the user's system is to check various properties of the composed system by the component and the user's

Program 4.5 A Required Scenario for the Auctioneer Component

scenario:

```
best_story {
  // step 1
  r_bs = logged_out && r_ps = not_available;
  // step 2
  r_bs = logged_out && r_ps = available;
  // step 3
  r_bs = logged_in && r_ps = available;
  // step 4
  r_bs = win && r_ps = engaged;
  // step 5
  r_bs = win && r_ps = sold;
  // step 6
  r_bs = logged_out && r_ps = sold;
}
```

working environment. Component users can write properties as assertions or temporal logic expressions, in which state values are the basic units. In SCIL, users write properties using patterns defined in [56]. In the same paper [56], how to map these patterns to temporal logic expressions is also presented. For example, if users write:

```
always (P = false)
```

the expression will be translated to Linear Temporal Logic (LTL) as:

```
[](!P)
```

Then component users can use the model checking tools that support LTL to check the property. Users can also use the keyword *deadlockfreeness* to require that the system is deadlock free. This property can only be checked by some specific tools, such as [54].

Given below are examples using the *always*, *between*, *precedes* and *after* operators and also specifying a property that denotes the absence of deadlock (Program 4.6).

Program 4.6 Required Properties for the Auctioneer Component

```
property:
  p1 {
    // property 1
    always !(r_bs = logged_in && r_ps = not_available);
    // property 2
    (r_bs = win || r_bs = not_win) between (r_bs = logged_in)
    and (r_bs = logged_out);
    // property 3
    ((r_ps = engaged) precedes (r_ps = sold)) after (r_ps = available);
  }

  p2 {
    // the absence of deadlock
    deadlockfreeness;
  }
```

In Program 4.6, Property 1 checks that bidders should not log into the system if the product is not available, and Property 2 makes sure that bidders can only bid after they have logged in but before they log out of the system. It is specified in Property 3 that the product can be purchased after the bidder has won the bidding.

4.3 Writing Specifications in SCIL

With SCIL, one can specify both components and user requirements.

4.3.1 Writing Component Specifications

A software component can be regarded as a piece of software that provides a set of services. Thus a component specification should include `SERVICE` definitions as well as the `PROTOCOL` part, which adds constraints on the services the component provides. A complete specification of the auctioneer component can be found in the Appendix C.

4.3.2 Writing Requirement Specifications

The user requirement specification in SCIL has two parts: user environment components and the required properties of joint behaviour by the integrated component and the user environment components. Scenarios can be regarded as special properties of the composed system.

User environment components are those components users have already had in their system, representing the user's working environment. Users may specify environment components by the keyword *environment*. Generally speaking, the description of an environment component has the same syntax as the one of a normal component except that the environment component only has one service, viz., *run* that is defined as a keyword. The *run* service defines how this environment component will interact with the integrated component.

In SCIL, the keyword *connects* is used to compose the components retrieved from the repository with the user environment components.

In the example of the auctioneer component, the user environment contains two types of components: bidders and sellers. The user defines their *run* services in order to connect to the integrated auctioneer component (Program 4.7).

Program 4.7 User Environment for the Auctioneer Component

```
environment component bidder connects auctioneer {
  run {
    ...
    // if the bidder has won the bidding and the product has been engaged to the bidder,
    // then the bidder can purchase the product
    b_bs = win && b_ps = engaged -> purchase;
  }
}

environment component seller connects auctioneer {
  run {
    true -> sell;
  }
}
```

In the *run* service of bidder components, services of the auctioneer component are called if the pre-conditions are satisfied. For example, if a bidder has won the bidding and the product has been engaged to the bidder, the bidder can purchase the product, so that the *purchase* service from the auctioneer component can be called. For seller components, they only need to notify the system that they want to sell products, so there are no pre-conditions required for the *run* service. A complete specification of the user requirement for the auctioneer component can be found in the Appendix C.

4.4 SCIL and Transition Systems

SCIL is derived from Interface Automata [52]. Formally, a typical SCIL specification describes a transition system along with requirements specified using a mix of temporal logic and the other transition systems. Formal definition of transition systems

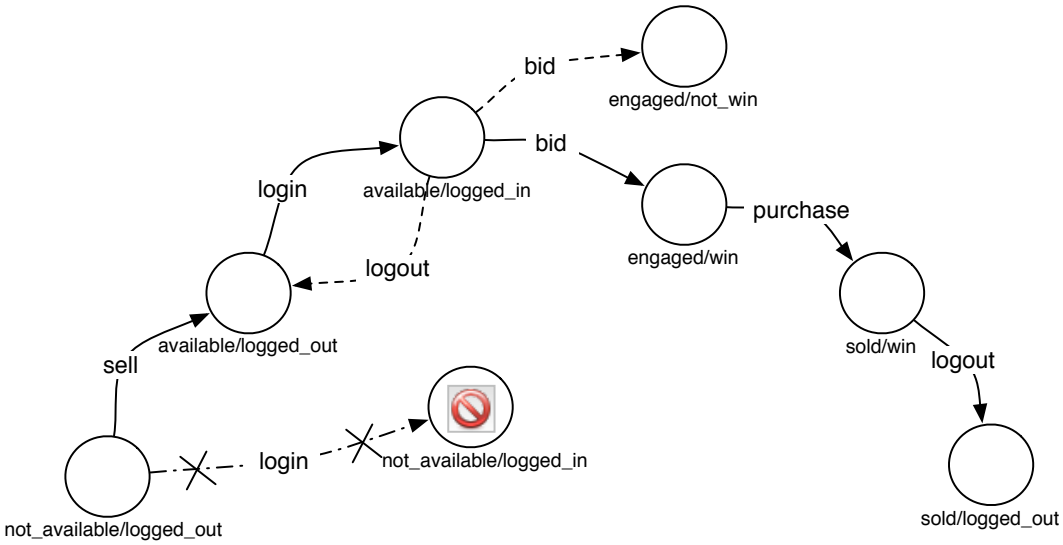

Figure 4.2: State Transitions in Auctioneer Component

can be found in [52].

In SCIL, component behaviour is the transition of interface states of the component. Interface states are represented by input and output variables. Service invocations result in the changes of those states. In the auctioneer example, `sell` service causes the product status to change from `not_available` to `available`. And `bid` service can cause two possible state transitions (see Figure 4.2).

The protocol can change the traces of the state transitions. For example, the protocol "eventually logout" forces `logged_out` to be the last state of the bidder's status.

Abstractly a scenario is just another transition system and ideally must be derivable from the original specification (e.g. the scenario `best_story` is denoted by solid lines with arrows in Figure 4.2). When checking user specified properties, model checking tools think those unreachable states are errors. In our example, the state "not_available/logged_in" (see Figure 4.2) is an unreachable state for using the auctioneer component.

4.5 Summary

Overall, using SCIL we can describe both component specifications and user requirement specifications. Component developers use SCIL to specify the services that components offer and the rules by which the components can function properly. Component users use SCIL to specify their working environment and the properties (including scenarios) they require the joint system to have.

The key syntactic constructs in SCIL (or other similar languages that can be used in our framework) should include the following:

- Named code fragments that are called services describe the high-level behavior of the component. The code fragments allow the designer to associate a transition system with the interface names.
- Constraints defined by rules and the protocol on how the various services can be combined.
- The ability to compose SCIL descriptions.
- Scenarios that describe the desired behaviour and a specification on what joint behaviour should achieve.

However, the expressiveness of SCIL is still limited since we try to keep interface model simple. For example, only parallel component composition is supported. Other composition styles may depend on the implementations of different formalism plug-ins. But no semantics of these compositions are defined in SCIL.

In conclusion, SCIL is not designed to describe interface signatures but describe a transition system along with requirements specified using a mix of temporal logic and the other transition systems, viz., user specified components and scenarios.

A combined SCIL description of components and requirements are translated to some existing modelling languages. Then these models are checked by various tools.

In the next chapter, we will present the architecture of our component selecting system and how we implement it.

Chapter 5

Implementations

5.1 Introduction

This chapter provides an architectural view and the implementation of the component selecting system that is based on checking behavioural compatibility.

The system implementation aims at providing a concrete realisation of our framework of selecting components. In the implementation, we build a web-based component search engine through which users can search required components by abstract behavioural properties instead of traditional keywords.

This chapter first presents an architectural view of the implementation, then shows how we design and implement the SCIL translator, the component repository and the web interface.

5.2 Architecture

Our prototype system implementation consists of four modules (see Figure 5.1): the web interface, the component repository database, the SCIL translator, and the model checkers (currently only jMocha [9] and Alloy Analyser [87] are supported).

The user interacts with the system through the web interface. A typical use of

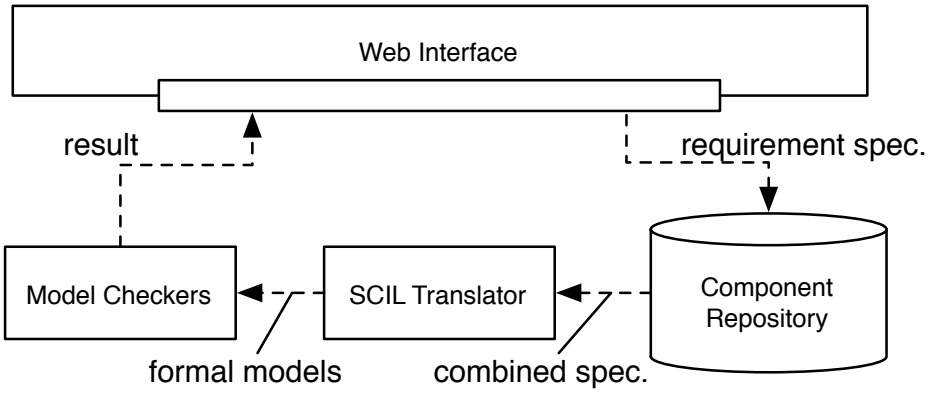

Figure 5.1: The System Modules

the system can be described as follows:

1. The user searches components by using keywords, and gets a list of candidate components that are described by the keywords.

2. The system allows the user to view the specifications of the candidate components and modify the requirement specification accordingly.

3. The user uploads the requirement specification to the component repository. The system will combine the requirement specification with each candidate component specification, and call the SCIL translator to translate the combined SCIL specification to the models in some existing formal languages.

4. The system will call the relevant model checking tools to check if the candidate component has required behaviour.

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.

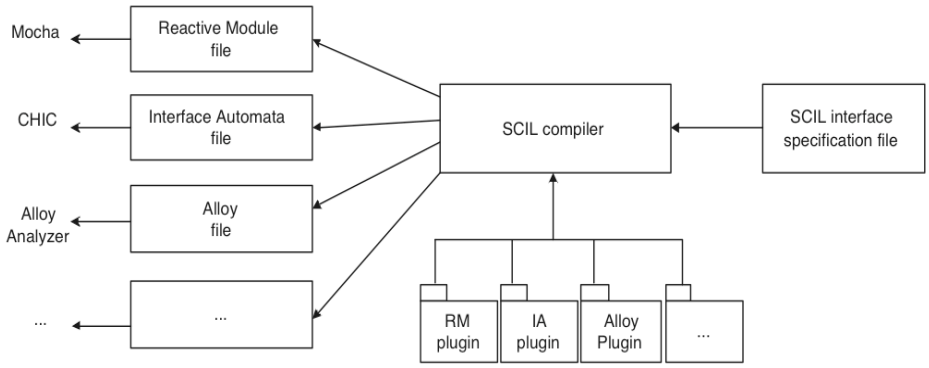

Figure 5.2: The SCIL Translator

5.3 The SCIL Translator

The SCIL Translator works in such a way: the SCIL compiler is fed with SCIL specifications. With the help from various language plug-ins, the compiler translates the SCIL specifications to those existing formal languages, such as Reactive Module (RM) [11], Alloy [87], Interface Automata (IA) [52], etc. The translated formal models will be input to their accompanying checking tools, for example, jMocha [10] for RM, Alloy Analyser [87], and CHIC [32] using IA. Figure 5.2 shows such a workflow.

Such a design has the following advantages:

- Model checking tools can be, in principle, used to check component compatibility. Although different tools provide different checking facilities, users have flexibility without having to write different specifications. A single SCIL specification is sufficient and can be reused with a variety of tools. If a new, more powerful tool becomes available, one only needs to write a code generator to use it within our framework.

- Component users and developers can exchange components and requirements descriptions in SCIL without worrying about which checking tools that they are are using.

Our prototype translator is built with plug-ins to support Reactive Module and Alloy at the moment. In order to prove the applicability of our approach, we select these two tools to demonstrate the viability of our architecture. These two tools are quite different; jMocha [10] does a temporal analysis based on state machines, while Alloy Analyser [87] checks that an assertion holds by trying to find a counterexample.

5.3.1 The Compiler

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 implementation has been done in layers (see Figure 5.3). At the bottom, the parser is generated by SableCC [64] to handle syntax of SCIL. And the class that performs grammar checking needs to follow the framework SableCC has generated. In our case, a combined specification may contain three parts: normal components, environment components (specified by an "environment" keyword) and user requirements (including scenarios and properties). Because different grammar rules apply for different parts, we use separate classes to handle grammar checking and generate separate tables for each part. The tables include symbol tables and rule tables. A symbol table is a common data structure used by a compiler or interpreter, where each symbol in the source code is associated with information such as type and scope level. A hash table implementation of a symbol table is used as standard and the symbol tables are maintained throughout all phases of translation. A rule table in our implementation stores the rules (state transitions) associated with different services the component provides.

On the top of grammar checking there exists a common translation layer. Its main task is to recognise the different parts of the specification, and assign a relevant grammar-checking module to that part. Finally, a list of symbol tables and rule tables are formed. This corresponds to the standard compilation process. As our framework

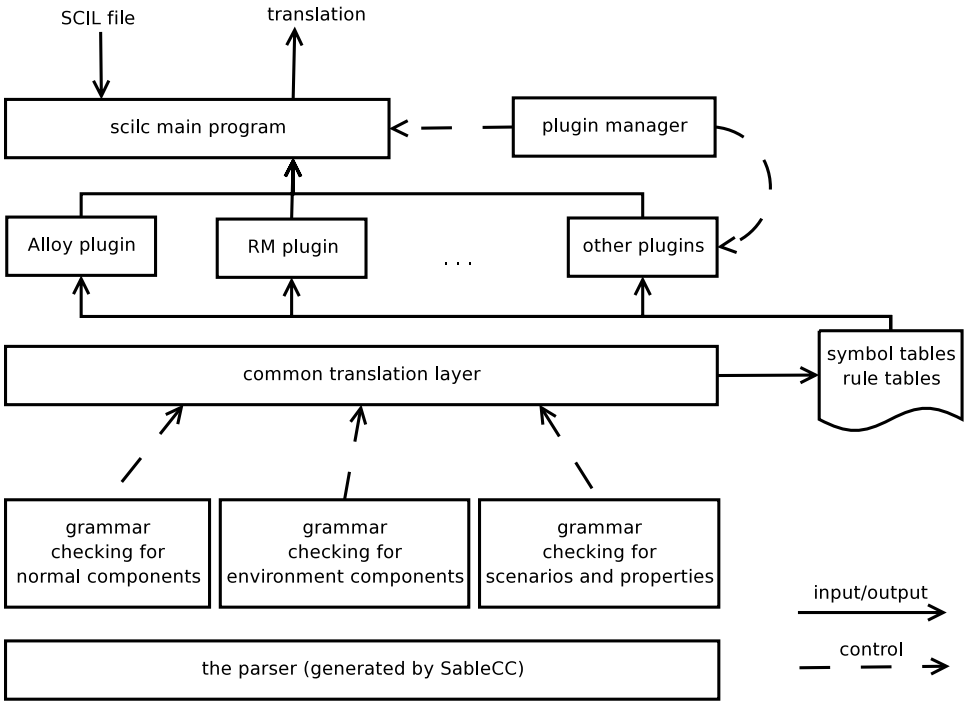

Figure 5.3: Compiler Implementation Layers

supports a number of code generators, the compiler calls the plug-in manager to load a plug-in class (for example, rm.jar for RM) that takes those intermediate tables as inputs and generates the translation for the plug-in language.

5.3.2 Developing Plug-ins

Enabling plug-ins has given users flexibility to deal with a variety of modelling languages. When developing plug-ins for the SCIL compiler, the following steps are followed:

1. Extending the common translation layer class to generate the intermediate symbol tables and rule tables. The tables are the same for all available plug-ins.

2. Defining the mapping from those tables to the language that the plug-in supports.

3. Writing a plug-in property file so that the `PluginManager` class can locate and load the plug-in at run time.

Thus it is possible to add support to various languages as long as the corresponding plug-ins are implemented. However, before picking up a language, some issues need to be considered, such as: whether the language supports describing transition systems; whether there exists a matching theory for using this language to check component compatibility, such as the specification matching from [153]. After the language is selected, the main task is to define the mapping from SCIL to that language.

We have implemented the plug-ins for RM and Alloy. Both of them can be used to describe transition systems, although Alloy needs to import the extra linear ordering module. For both tools, checking component compatibility is to check whether the composition of the two components have any illegal behaviours. RM directly supports component composition by stating `component 1 || component 2`, while Alloy does not explicitly support 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 as: `service s: a -> b` (if `a` is satisfied, then `b` is guaranteed). Translating 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 `s` is true and `a` is true, then `b` is true).

In SCIL, input ports are read-only, but output ports can be modified. RM has a similar policy on the interface ports, thus the translation from SCIL to RM is straightforward. Alloy does not have the concept of ports, assuming all the variables are writable. Thus in Alloy, there is no need to declare variables as input or output.

For the protocol definitions, the temporal operator `once` can be translated by introducing a variable controlling the number of the service invocations. The operator `initially` will make sure that the service is called before the others by adding its

rules to the initialisation code in RM or the initialisation predicate in Alloy. The operator `eventually` uses a boolean type to make the service the last one invoked. However, not all the protocol definitions can be translated to RM or Alloy, because some temporal operators are not supported by both languages, such as `causedby` and `precedes`.

One can define either scenarios and properties to be checked in the requirement specification. Scenarios can only be translated to RM because RM supports monitor automata that can execute in parallel with the component. Properties can be translated to various assertions that are supported by both RM and Alloy.

5.4 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.

5.5 Web Interface

For component developers and component users, different user interfaces are implemented. Through the web interface, the system allows developers to add components, including specifying components in SCIL. Meanwhile the system allows users to upload requirements and match components based on the requirement specification. Figure 5.4 displays what users and developers can do through the web interface.

A user starts searching for components by using keywords to reduce the number of

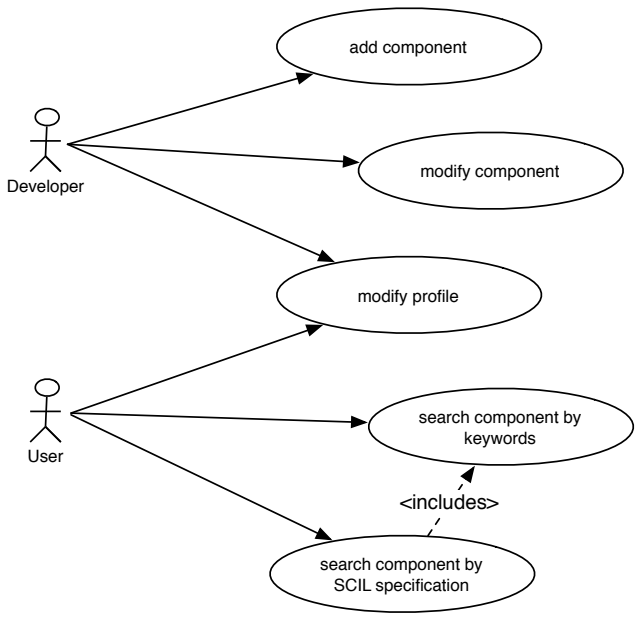

Figure 5.4: The Use Cases through the Web Interface

candidate components (see Figure 5.5 and Figure 5.6). Since all the components are described by keywords, the components that contain some of the user's required keywords will be retrieved as candidate components for further behavioural specification matching.

The user can view candidate component specifications and modify the requirement specification if needed (see Figure 5.7).

Before the requirement specification is combined with each candidate component specification, both specifications are syntactically checked. 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 user is asked to map the names between two specifications given the number of the names is the same. If no parsing errors are found, the user will be led to the page where name mapping can be done (see Figure 5.8).

If the requirement specification has been changed according to the component

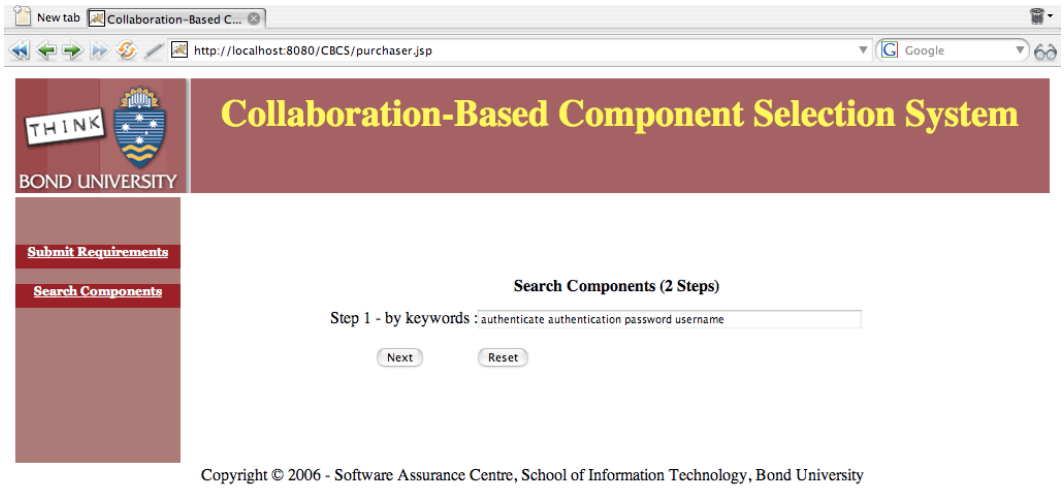

Figure 5.5: Screen-shot: Search by Keywords

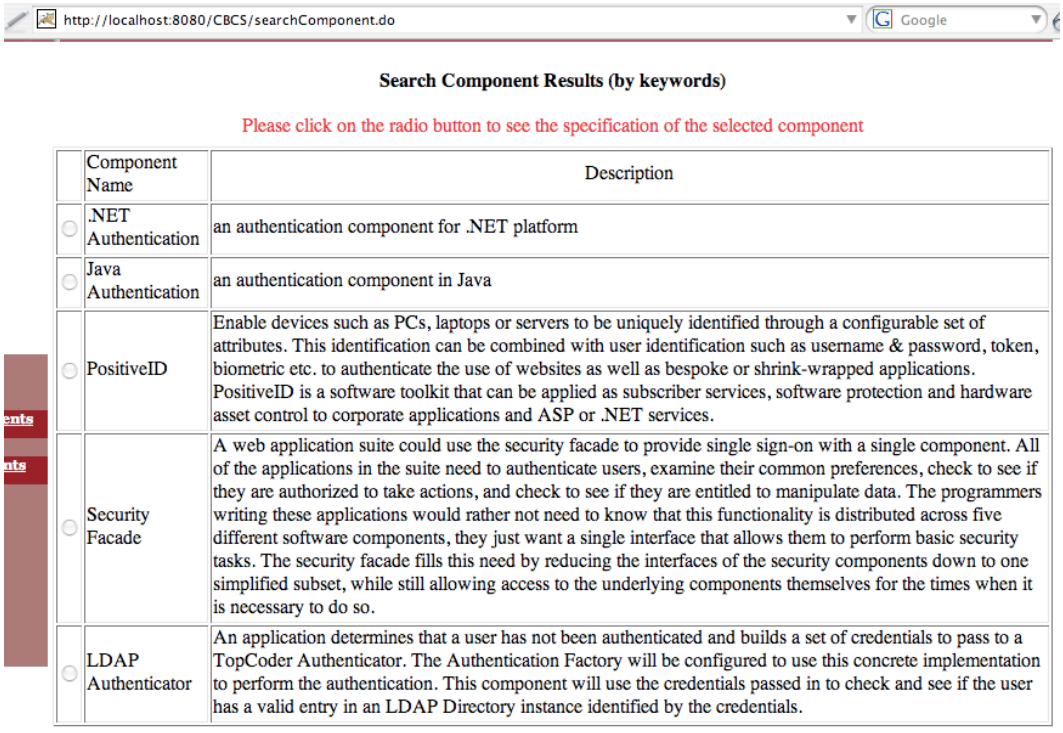

Figure 5.6: Screen-shot: Search Results by Keywords

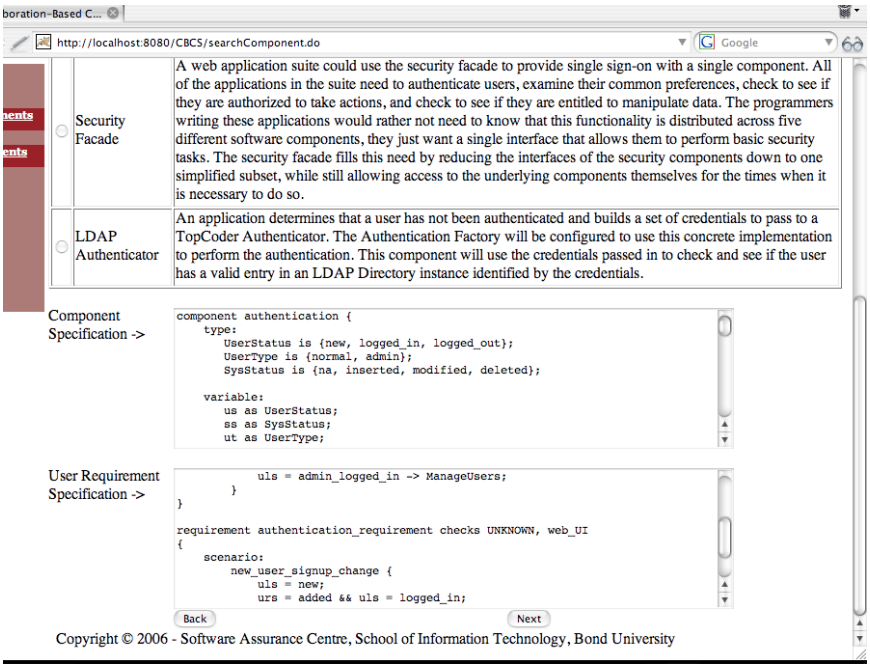

Figure 5.7: Screen-shot: View and Modify Specifications

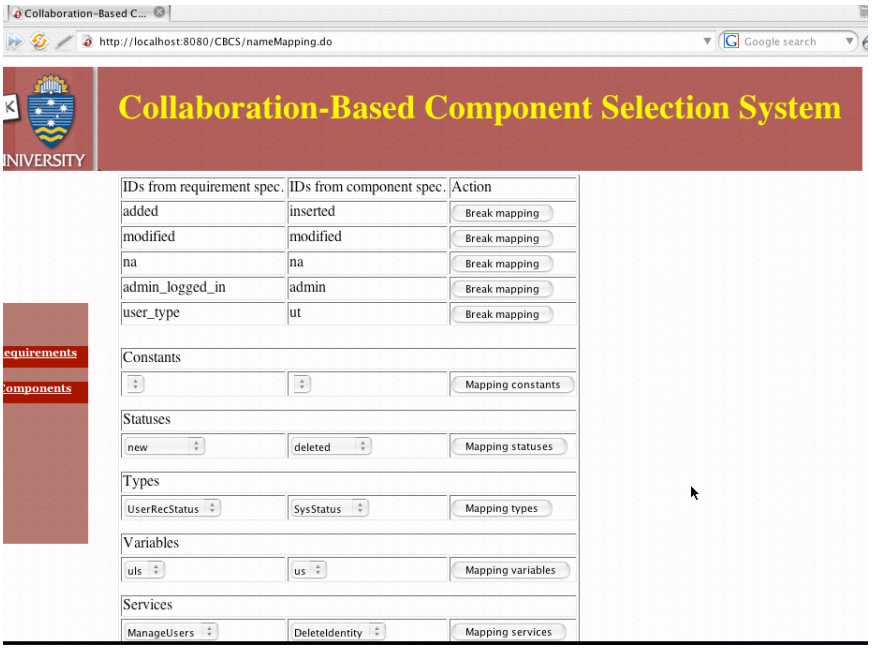

Figure 5.8: Screen-shot: Name Mapping

specification, it may not be necessary to do the name mapping. However, the name-mapping module as least can check the syntax of both specifications. Then the requirement specification will be combined with each candidate component specification.

The system decides which language (RM or Alloy) it 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. The properties that are not supported by both tools will be also ignored by the translator.

The system will run the model checkers in the background. Thus the model checking is transparent to the component user. In order to achieve that, SLang scripts 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 (see Figure 5.9). 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 if the properties can be handled by both tools.

The use of the system by component users can be depicted as Figure 5.10.

5.6 Summary

This chapter presents the architecture and detailed implementation of our prototype component selecting system based on checking behavioural compatibility. At the moment it only supports translating to RM and Alloy, and some manual work of adjusting requirement specifications is involved. The future work includes adding more tools support, such as Ticc [3], and automatic tool selection becomes an issue.

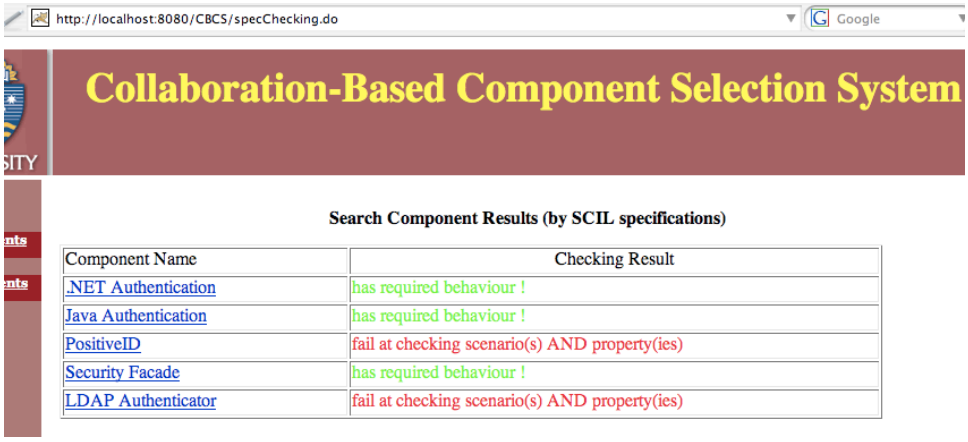

Figure 5.9: Screen-shot: Compatibility Checking Result

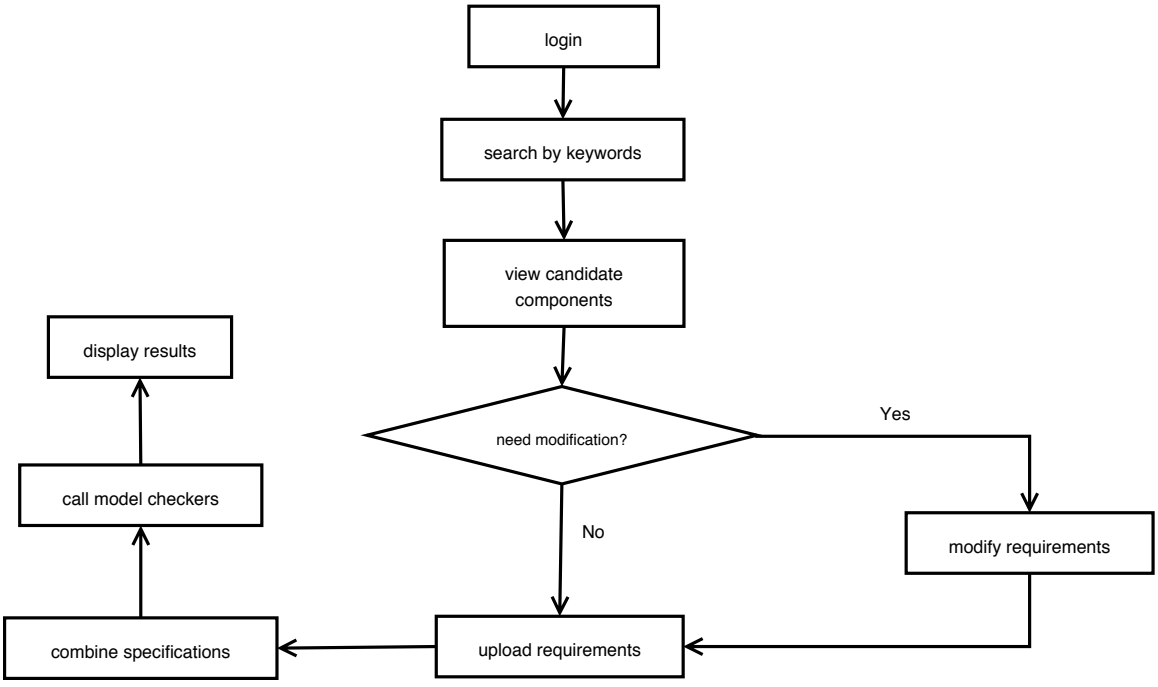

Figure 5.10: The Flowchart of Using the System

Chapter 6

Case Studies

6.1 Introduction

This chapter provides feasibility studies of our approach to selecting commercial components from the real-life component marketplace.

The first example briefly discusses how to check the behavioural compatibility of the auctioneer component with the user-specified requirement by using the tools that we developed. However, this example is not a full case study but a continuation of the example from Chapter 4 in order to illustrate the features of the tools we have used.

The other two case studies aim at identifying the advantages and disadvantages of our approach by comparing it with the traditional approach of searching by keywords. The comparison is conducted mainly based on the effort and the precision of retrieving required components.

The task in the second case study is to find a component that can spell check the texts user's input through a user interface, while the task in the third case study is to search for a group of components that support online shopping.

Our sample repository contains totally 132 components. All these components are mainly taken from the real-life component sources: ComponentSource [42] and Top-

Category	Description	Quantity
3D Modelling	Adding 3D graphics to applications	3
Addressing and Postcode	Managing people's contacts	3
Business Rules	Building a business framework	4
Calendar/Schedule	Creating calendars that support scheduling	3
Charting and Graphing	Creating charts/graphs based on user input data	7
Credit Card Authorisation	Verifying the validity of credit cards	3
Data Validation	Validating input data against criteria	3
Database Reporting	Creating reports for databases	3
Drawing	Adding 2D graphics to applications	7
e-Commerce	Handling e-business activities, such as online shopping, etc.	28
Email	Providing email functions to applications	3
Financial	Providing functions related to banks	4
Imaging	Processing images for applications	5
Maths and Stats	Providing maths or statistics functions	5
PDF	Creating PDF files in applications	4
Reporting	Generating reports for applications	4
Security and Administration	Checking security issues	9
Spelling	Providing spell checking to applications	7
Spreadsheet	Providing spreadsheet functions	7
Toolbar	Creating toolbars for applications	7
User Interface	Creating general user interface elements	7
Zip and Compression	Compressing and uncompressing files	6

Table 6.1: Component Distribution in the Sample Repository

Coder [142]. ComponentSource is the world's largest marketplace and community for reusable components, and it provides a large repository for all kinds of software components. TopCoder has its own component repository built by its members all over the world. However, both ComponentSource and TopCoder only support searching components by keywords.

The distribution of these 132 components to different categories is displayed in Table 6.1. The category names are taken from the website of ComponentSource. The brief explanations of these categories are also given in Table 6.1.

For Case Study 2 and 3, we first outline the user requirements for the desired components, and then give SCIL specifications for some existing candidate components. The details of searching in the repository, translating and checking RM and Alloy models are also presented. Finally, we will discuss the results and our experiences.

6.2 Case Study 1: Checking Behavioural Compatibility for the Auctioneer Component

Program 6.1 Type Translation

RM:

type BidderStatus is {logged_in, logged_out, win, not_win}

type ProductStatus is {not_available, available, engaged, sold}

Alloy:

```
abstract sig BidderStatus { }
one sig logged_in, logged_out, win, not_win extends BidderStatus {}
abstract sig ProductStatus { }
one sig not_available, available, engaged, sold extends ProductStatus {}
```

Program 6.2 Sell Service Translation

RM:

```
[] true & bs = logged_out & ps = not_available & sell_num_ < 1 & sell? ->
   ps' := available; sell_num_' := inc sell_num_ by 1
```

Alloy:

```
pred sell (bs, bs': BidderStatus, ps, ps': ProductStatus) {
   ps' = available && bs' = bs
}
```

Program 6.3 Check best_story Scenario in RM

```
module matching_best_story_ is auctioneer_sys || best_story
predicate pred_best_story_ is (final_best_story_ = true)
judgment J_best_story_ is matching_best_story_ |= pred_best_story_
```

In Chapter 4 we have specified the auctioneer component and the user requirement for such a component. The complete SCIL specifications can be found in Appendix C. In order to check behavioural compatibility of the component with the user requirement, two specifications are combined and then translated to both RM and Alloy.

RM directly supports enumeration type, but Alloy does not. In Alloy we introduce a signature and then extend it with the appropriate elements. The translation of the `BidderStatus` and `ProductStatus` is shown in Program 6.1.

A service is translated to an update statement in RM. In Alloy, a service is expressed as a predicate function. The different translations are shown in Program 6.2.

Checking behavioral compatibility is handled differently by the different tools. For instance, the tool jMocha checks whether the composition of auctioneer, bidder and seller components will reach some undesirable states by performing a temporal analysis. The composition is done by using renaming and parallel composition in

Program 6.4 Check a Property in RM

RM:

```
predicate pred_p10_ is   ( bs = logged_in & ps = not_available )
judgment J_p10_ is auctioneer_sys |= pred_p10_
```

Alloy:

```
assert p1 {
  all s: State | ! ( s.bs = logged_in && s.ps = not_available )
}
check p1 for 8 State
```

RM. These undesirable situations can be identified by the user requirements. For example, it is not allowed that when a bidder wins the auction, the product is still not available. This is obviously an undesirable state that should be avoided.

A monitor [11] technique introduced in RM can be used to in checking of scenarios. A monitor can only observe but not interfere with the behavior of the composition. The user describes the expected state transitions in the monitor module and the translation of the scenarios is similar to that of services.

The composite system is described as a parallel composition of the system and the scenario. If the composite system can complete the scenario the user is sure that the component is acceptable as it meets the specified requirements. In order to check this, we use a predicate that is set to true on completion of the scenario. The RM specification to check one scenario `best_story` is shown as Program 6.3.

The Alloy Analyzer allows the user to specify state transitions. So the checking of reachability of undesirable states is similar to that of RM. Both Alloy and RM support checking properties specified using predicates (see Program 6.4). The complete translations of the combined auctioneer specification to RM and Alloy can be found in Appendix C.

6.3 Case Study 2: Search For A Spell Checker Component

In this case study, we will find a spell checker component based on the user requirements. We first describe the user requirements using SCIL.

6.3.1 Requirements for the Desired Component

The basic functions of a spell checker component required by the user are:

1. The component can provide spell checking for single words, phrases and paragraphs, and text documents.
2. The component can provide spell checking as the user types.
3. The component can locate misspelled words.
4. Each spelling error should be returned with suggested modifications if suggestions are needed by the user.
5. Misspelled words can be ignored, and unknown words can be added to the dictionary.

In order to let the potential spell checker component work in the user's environment, more details about the above basic functions and the user's environment need to be specified. Program 6.5 is the complete SCIL specification of the user requirements.

In this requirement specification, `SpellCheckStatus` is defined to describe the status of spell checking text: `ready` means that the text is ready to be checked; if the word is misspelled, the checking status becomes `incorrect`; if modification suggestions are given to the misspelled word, the checking status becomes `suggested`; and if the misspelled word is ignored or added to the dictionary, the checking status will become `ignored` or `added` respectively.

Program 6.5 Requirement Specification for Spell Checker Component

```
/*
 * Component required
 */
component UNKNOWN {
   // type definitions
   type:
      SpellCheckStatus is {ready, incorrect, suggested,
         ignored, added};
      ResultDisplay is {no_error, highlighted, underlined,
         suggestion_displayed, add_new_word_displayed};

   // global variables
   variable:
      scs as SpellCheckStatus;
      rd as ResultDisplay;
      check_as_typing as bool;
      need_suggestion as bool;

   // required services from this component
   service:
      CheckSpellAsTyping {
      }
      CheckSpellInWindow {
      }
      HaveSuggestions {
      }
      CorrectBySuggestion {
      }
      AddtoCustomDictionary {
      }
      IgnoreWords {
      }
}

/*
 * User's environment component - the user interface UI
 */
environment component UI connects UNKNOWN {
   // special service for an environment component
   service:
      run {
         scs = ready && check_as_typing = true
                        -> CheckSpellAsTyping;
         scs = ready && check_as_typing = false
                        -> CheckSpellInWindow;
         scs = incorrect && need_suggestion = true
                        -> HaveSuggestions;
         scs = suggested -> CorrectBySuggestion;
         scs = incorrect -> IgnoreWords;
         scs = incorrect -> AddtoDictionary;
         scs = suggested -> IgnoreWords;
         scs = suggested -> AddtoDictionary;
         scs = ignored && check_as_typing = true
                        -> CheckSpellAsTyping;
         scs = ignored && check_as_typing = false
                        -> CheckSpellInWindow;
         scs = added && check_as_typing = true
                        -> CheckSpellAsTyping;
         scs = added && check_as_typing = false
                        -> CheckSpellInWindow;
      }
}

/*
 * Scenarios and properties
 */
requirement spell_checker checks UNKNOWN, UI {
   // scenario definitions
   scenario:
      check_as_type_with_suggestion {
         scs = ready && rd = no_error &&
               check_as_typing = true;
         scs = incorrect && rd = underlined &&
               need_suggestion = true;
         scs = suggested &&  rd = suggestion_displayed;
         scs = ready && rd = no_error;
      }
      check_as_type_ignored {
         scs = ready && rd = no_error &&
               check_as_typing = true;
         scs = incorrect && rd = underlined;
         scs = ignored;
         scs = ready && rd = no_error;
      }
      check_in_window_with_suggestion_ignored {
         scs = ready && rd = no_error &&
               check_as_typing = false;
         scs = incorrect && rd = highlighted &&
               need_suggestion = true;
         scs = suggested && rd = suggestion_displayed;
         scs = ignored;
         scs = incorrect && rd = highlighted;
      }
      check_in_window_add_new_words {
         scs = ready && rd = no_error &&
               check_as_typing = false;
         scs = incorrect && rd = highlighted;
         scs = added && rd = add_new_word_displaye;
         scs = ready && rd = no_error;
      }

   // property definitions
   property:
      p1 {
         always !(scs = incorrect && td = no_error);
         always !(scs = ready && td != no_error);
      }
      p2 {
         always (scs = incorrect) precedes (ts = ignored);
         always (scs = incorrect) precedes (ts = added);
      }
}
```

Another type `ResultDisplay` is used to describe the display changes on the user interface: if the word is misspelled, it will be `highlighted` or `underlined` with red colour depending on the checking mode. Otherwise, it shows no errors found (denoted by `no_error`). If suggestions are required for the misspelled word, a list of suggestions will be displayed (denoted by `suggestion_displayed`). When a new word is being added to the dictionary, we denote the display change as `add_new_word_displayed`.

There are two different checking modes required by the user. If the variable `check_as_typing` is true, the spelling will be checked as the user types, so the service `CheckSpellAsTyping` is called. If a word is misspelled, it will be underlined with red colour. If the variable `check_as_typing` is false, spell checking is performed only after words are completed in the window. Thus the service `CheckSpellInWindow` is required, and it will highlight the misspelled word.

If the variable `need_suggestion` is true, modification suggestions are needed when a word is misspelled. The user may call the service `CorrectBySuggestion` to correct the misspelled word by one of the suggestions, or ignore the spelling error by `IgnoreWords`. The user also has an option to add the "misspelled" word to the dictionary. Even if suggestions have been given for the misspelled word, the user still has the chance to ignore this word or add this word to the dictionary. After the misspelled word is ignored or added to the dictionary, the user may call spell checking services again.

Four spell checking scenarios are displayed in Figure 6.1. The first scenario (a) shows that when spell checking as the user types, a misspelled word is corrected by the suggestion, while in the second scenario (b) the misspelled word is simply ignored. The third scenario (c) shows checking spelling in the window, and the misspelled word being ignored even though the suggestions have been given. But if being checked again, the ignored word will still be marked as incorrect. And the fourth scenario (d) shows the "misspelled" word being added to the dictionary.

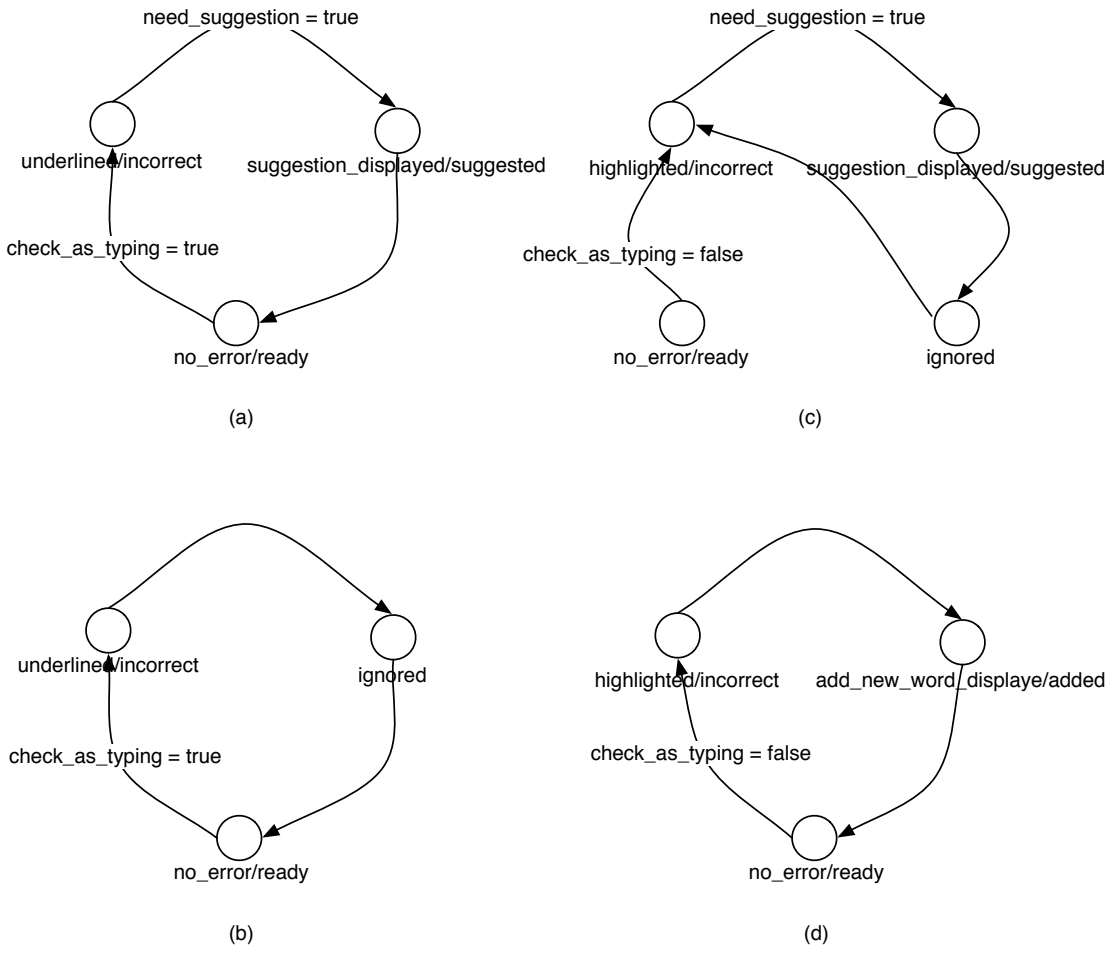

Figure 6.1: Four Spell Checking Scenarios

Two situations are prohibited by the first two property definitions: 1. a word is misspelled, but the spell checker component shows no errors found; 2. a word is correctly spelled, but it is highlighted or underlined. The third and forth properties check the order of state changes: always after spotting a word is misspelled, the user can ignore this misspelling, or add the new word to the dictionary.

6.3.2 Specifying Components

In this section, we give the SCIL specifications of some previously built components taken from our sample repository.

6.3.2.1 Telerik r.a.d.spell Component

Telerik [140] r.a.d.spell component enables users to add multilingual spell checking capabilities to their applications. From its user manual, we can develop a use case diagram as in Figure 6.2.

The services of the r.a.d.spell component can be identified from the use case diagram (see Figure 6.2). The business logic of each service can also be decided by the component's user manual. Since we are only concerned about high-level key behaviour of the component, the use cases such as choose dictionary, customise appearance and choose suggestion generating algorithm can be ignored. Based on the types we have defined, we can draw a state transition diagram of the component (see Figure 6.3).

We can specify the component in SCIL as shown in Program 6.6.

According to the component specification, when spell checking text the user can only get two results: spelled correctly (denoted by `ok`) or `misspelled`. If a word is misspelled, the incorrect word will be `underlined` with red colour when the "Check Spelling as You Type" mode is selected (denoted by `check_as_typing = true`). Otherwise, the incorrect word will be `highlighted`. If a word is misspelled, and the user needs suggestions (denoted by `need_suggestion = true`), the component will display a list of suggestions for the misspelled word (denoted by `suggestion_list`). The

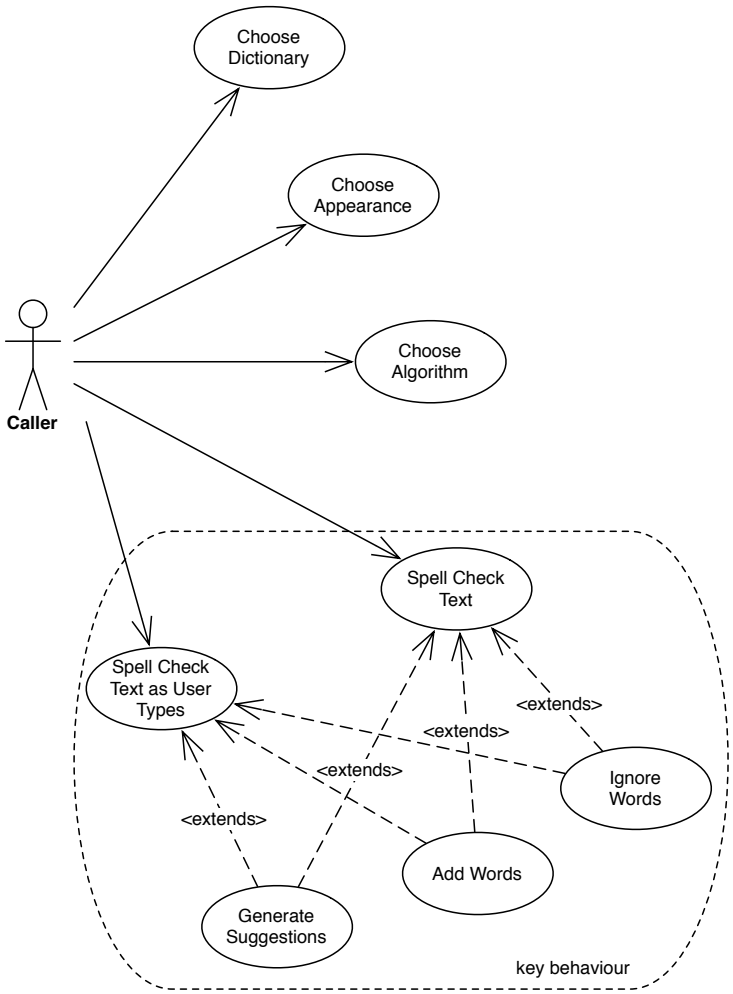

Figure 6.2: r.a.d.spell Component Use Case Diagram

Program 6.6 Specification of r.a.d.spell Component

```
component telerik {
  // type definitions
  type:
     TextStatus is {ok, misspelled, suggesting,
        ignored, adding};
     TextDisplay is {normal, highlighted, underlined,
        suggestion_list, add_word_box};

  // variables
  variable:
     ts as TextStatus;
     td as TextDisplay;
     check_as_typing as bool;
     need_suggestion as bool;

  // service definitions
  service:
     init {
        output: ts, td
        rule:
           true -> ts = ok && td = normal;
     }

     SpellCheckTextAsTyping {
        input: ts, check_as_typing
        output: ts, td
        rule:
           ts = ok && check_as_typing = true ->
              ts = misspelled && td = underlined;
           ts = ok && check_as_typing = true ->
              ts = ok && td = normal;
           ts = ignored -> ts = ok && td = normal;
           ts = adding -> ts = ok && td = normal;
     }

     SpellCheckText {
        input: ts, check_as_typing
        output: ts, td
        rule:
           ts = ok && check_as_typing = false ->
              ts = misspelled && td = highlighted;
           ts = ok && check_as_typing = false ->
              ts = ok && td = normal;
           ts = ignored -> ts = misspelled
              && td = highlighted;
           ts = adding -> ts = ok && td = normal;
     }

     GenerateSuggestions {
        input: ts, need_suggestion
        output: ts, td
        rule:
           ts = misspelled && need_suggestion = true ->
              ts = suggesting && td = suggestion_list;
     }

     ChangeBySuggestion {
        input: ts
        output: ts, td
        rule:
           ts = suggesting -> ts = ok && td = normal;
     }

     AddNewWords {
        input: ts
        output: ts, td
        rule:
           ts = misspelled -> ts = adding
              && td = add_word_box;
           ts = suggesting -> ts = adding
              && td = add_word_box;
     }

     IgnoreWords {
        input: ts
        output: ts, td
        rule:
           ts = misspelled -> ts = ignored;
           ts = suggesting -> ts = ignored;
     }

  // protocol definitions
  protocol:
     GenerateSuggestions precedes
           ChangeBySuggestion;
}
```

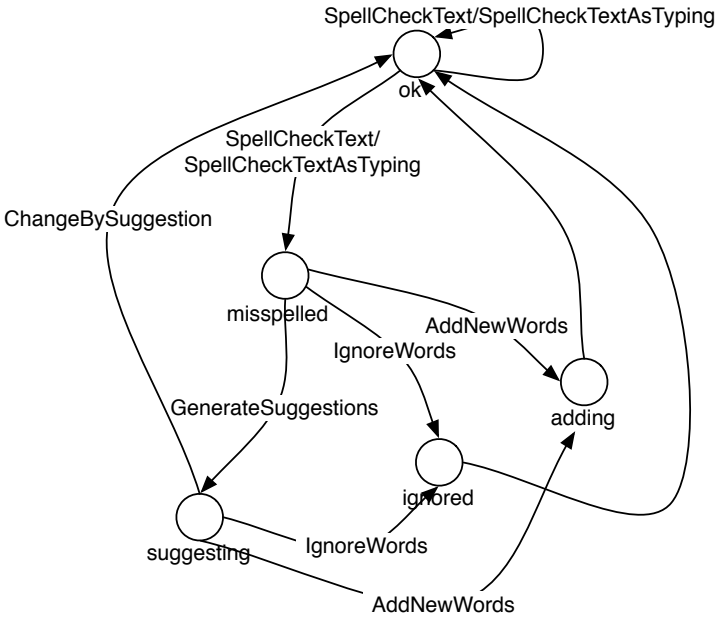

Figure 6.3: r.a.d.spell Component State Transitions

user can correct the misspelled word by using one of the suggestions. The component also allows the user to ignore the checking error even after the suggestions are given for the misspelled word. The ignored word is displayed as if no errors are found when being spell checked as the user types. However, with the "Check Spelling as You Type" option turned off, the ignored word will still be marked as incorrect (and be highlighted in this case). The component also allows users to add new words to the dictionary. After being added, these new words will be considered as correctly spelled words.

In the specification we need to make sure that `GenerateSuggestions` service should be called before `ChangeBySuggestion`. That is to say, a misspelled word can be corrected by a suggested modification only when there exist suggestions for the misspelled word.

6.3.2.2 ComponentOne C1Spell Component

Another candidate component is C1Spell from ComponentOne [102]. By the same way we can develop its specification in SCIL as shown in Program 6.7.

According to the specification, five actions are defined for the component. If a word is correctly spelled, the variable `if_correct` is true, and there is no action the component will do. If the word is misspelled, the component will send a `type_error_action` event to the user's environment. If modification suggestions for the misspelled word are needed (denoted by `auto_correct = true`), the component will send `suggesting` action. If the user wants to add the new word to the dictionary, the action `adding` is sent. Finally, the user can ignore the misspelled word by the action `ignoring`.

The spell checking behaviour of the C1Spell component is similar to the r.a.d.spell component. However, there exist some differences. We are going to talk about the differences in Section 6.3.3.

6.3.2.3 ChadoSpellText Component

ChadoSpellText component is provided by Chado Software [31]. It has a specification as Program 6.8.

This component provides only basic spell checking functions. It cannot check spelling as the user types, and it cannot generate modification suggestions for the misspelled words either.

6.3.3 Searching Components in the Repository

The user first searches in the repository by keywords to reduce the number of candidates to 12. In this case study, the keywords are `spell spelling check checking`. These 12 components, including r.a.d.spell, C1Spell and ChadoSpellText, have been specified in SCIL. At the second step, the user is able to upload the requirement

Program 6.7 Specification of C1Spell component

```
component C1Spell {
    // type definitions
    type:
        SpellCheckAction is {no_action, type_error_action,
            suggesting, ignoring, adding};

    // variables
    variable:
        sca as SpellCheckAction;
        auto_correct as bool;
        if_correct as bool;
        check_as_typing as bool;

    // service definitions
    service:
        init {
            output: sca, if_correct
            rule:
                true -> if_correct = true && sca = no_action;
        }

        CheckTyping {
            input: check_as_typing, sca
            output: if_correct, sca
            rule:
                check_as_typing = true ->
                    if_correct = true && sca = no_action;
                check_as_typing = true ->
                    if_correct = false && sca = type_error_action;
                sca = ignoring && check_as_typing = true ->
                    if_correct = false && sca = no_action;
                sca = adding && check_as_typing = true ->
                    if_correct = true && sca = no_action;
        }

        CheckString {
            input: check_as_typing, sca
            output: if_correct, sca
            rule:
                check_as_typing = false ->
                    if_correct = true && sca = no_action;
                check_as_typing = false ->
                    if_correct = false && sca = type_error_action;
                sca = ignoring && check_as_typing = false ->
                    if_correct = false && sca = type_error_action;
                sca = adding && check_as_typing = false ->
                    if_correct = true && sca = no_action;
        }

        BuildSuggestions {
            input: if_correct, auto_correct
            output: sca
            rule:
                if_correct = false && auto_correct = true ->
                    sca = suggesting;
        }

        CorrectBySuggestion {
            input: sca
            output: sca, if_correct
            rule:
                sca = suggesting -> if_correct = true
                  && sca = no_action;
        }

        AddNewWords {
            input: if_correct
            output: sca
            rule:
                if_correct = false -> sca = adding;
                sca = suggesting -> sca = adding;
        }

        IgnoreWords {
            input: if_correct
            output: sca
            rule:
                if_correct = false -> sca = ignoring;
                sca = suggesting -> sca = ignoring;
        }
}
```

Program 6.8 Specification of ChadoSpellText Component

```
component ChadoSpellText {
 type:
    SpellCheckStatus is {ready, incorrect, ignored, adding};
    ResultDisplay is {no_error, highlighted, add_new_word_displayed};

 variable:
    scs as SpellCheckStatus;
    rd as ResultDisplay;

 service:
    init {
      output: scs, rd
      rule:
        true -> scs = ready && rd = no_error;
    }

    CheckString {
      input: scs
      output: scs, rd
      rule:
        scs = ready -> scs = incorrect && rd = highlighted;
        scs = ready -> scs = ready && rd = no_error;
        scs = ignored -> scs = incorrect && rd = highlighted;
        scs = adding -> scs = ready && rd = no_error;
    }

    AddWordsToCustom {
      input: scs
      output: scs, rd
      rule:
        scs = incorrect -> scs = adding &&
          rd = add_new_word_displayed;
    }

    IgnoreAll {
      input: scs
      output: scs, rd
      rule:
        scs = incorrect -> scs = ignored;
    }
}
```

specification in SCIL to the selection system.

The system retrieves all the naming from both component and requirement specifications, and then asks the user to map the naming between two specifications. This is necessary because users and developers use different names of types, variables and services in their specifications. However, this method can only handle simple naming differences. It requires that the number of the names mapped from two specifications should be the same, and two different names mapped should have the same type or the same meaning. Thus among the 12 candidate components, only telerik [140] r.a.d.spell component can directly use the name-mapping method with the user requirement specification, because it is modelled in a similar way as the one for the requirement specification.

The other 11 components have different styles of definitions on types, variables and services than the user requirement specification, so the name-mapping method cannot be used directly for those components. Take C1Spell component as an example. Its service names can be directly mapped with the ones specified in the user requirement specification, such as: `CheckTyping` to `CheckSpellAsTyping`, `CheckString` to `CheckSpellInWindow`, `BuildSuggestions` to `HaveSuggestions`, etc. However, the types and variables are not the case, because they have been defined differently in the two specifications. In the requirement specification, the type `SpellCheckStatus` is used to describe the states of spell checking text. And for each state, there is also a result displaying. For example, when a word is found misspelled, the word will be displayed as underlined with red colour. While in the C1Spell component specification, the spell checking states are described by the different actions (or events) the component sends. Modelling the same thing inconsistently often happens between users and developers. This is the place that the simple name-mapping method cannot be used.

Therefore we have to modify the selecting component process: for each candidate component (those filtered by the keywords), before mapping names, we have to view

the component specification. In this case study, there are 12 components to be viewed. We also have to modify the requirement specification according to the component specification, making sure they are modelled consistently. For the C1Spell component, we modify the user requirement specification in this way:

```
run {
   check_as_typing = true -> CheckTyping;
   check_as_typing = false -> CheckString;
   if_correct = false && auto_correct = true -> BuildSuggestions;
   sca = suggesting -> CorrectBySuggestion;
   if_correct = false -> IgnoreWords;
   if_correct = false -> AddNewWords;
}
```

The different names can be changed during the modification of the requirement specification, in this case, name mapping of the two specifications can be skipped. The modified specification of the user's environment actually does the same thing as before, but is presented differently. Similarly, the scenario definitions and the required properties are also modified accordingly. This is the modified spell checking scenario `check_as_type_with_suggestion`:

```
if_correct = true && sca = no_action && check_as_typing = true;
if_correct = false && sca = type_error_action && auto_correct = true;
sca = suggesting;
if_correct = true && sca = no_action;
```

Comparing with the original requirement specification, the modified version is different on displaying the misspelled words. The user requires that under different checking modes, the misspelled word is marked differently: underlined with red colour or highlighted. However, for C1Spell component, a `type_error_action` will be sent if a word is checked misspelled regardless of checking modes.

Sometimes it is not possible to modify the requirement specification according to the component specification, otherwise the user requirements would be greatly changed. For example, the RichTextBox component from [132] is a WYSIWYG (What Your See Is What You Get), rich-text content editor. It also has keywords `spell checking`, because it provides spell checking to the text edited in the text control. But obviously this is not the component we are looking for. Thus it is not possible for the user to modify the requirement specification based on an actually irrelevant component. We can just ignore this component. In this case study, there are five of 12 components that are not spell checker components, but their descriptions contain some of the keywords.

If the component specification and the requirement specification are modelled in a consistent manner, the system combines two specifications.

6.3.4 Translating and Model Checking

In this case study, two model checking tools are used: jMocha and Alloy Analyser. JMocha supports checking both scenarios and properties, while Alloy Analyser only supports properties check.

Program 6.9 is the translation of the scenario `check_as_type_with_suggestion` for the r.a.d.spell component. The check on this scenario is also automatically generated. The translations of the other components are similar.

Both Alloy and RM support checking properties specified using predicates. The translations of the first two properties in RM and Alloy are as Program 6.10.

However, neither RM nor Alloy supports temporal operators such as `precedes`. Thus the judgements that contain those temporal operators are not translated.

6.3.5 Results

There are three components that pass the checking on the user-specified scenarios and properties by both jMocha and Alloy Analyser. These components include r.a.d.spell

Program 6.9 Translation of a Scenario to RM

```
module check_as_type_with_suggestion is
 interface alert_check_as_type_with_suggestion_: (0..4);
   final_check_as_type_with_suggestion_: bool
 external need_suggestion: bool; check_as_typing: bool;
   td: TextDisplay; ts: TextStatus;
 lazy atom main controls alert_check_as_type_with_suggestion_,
   final_check_as_type_with_suggestion_
            reads alert_check_as_type_with_suggestion_,
   need_suggestion, check_as_typing, td, ts

 init
   [] true -> alert_check_as_type_with_suggestion_' := 0;
     final_check_as_type_with_suggestion_' := true
 update

   [] alert_check_as_type_with_suggestion_ = 0 & ts = ok
     & td = normal & check_as_typing = true  ->
     alert_check_as_type_with_suggestion_' := 1;
     final_check_as_type_with_suggestion_' := true
   [] alert_check_as_type_with_suggestion_ = 1 & ts = misspelled
     & td = underlined & need_suggestion = true  ->
     alert_check_as_type_with_suggestion_' := 2;
     final_check_as_type_with_suggestion_' := true
   [] alert_check_as_type_with_suggestion_ = 2 & ts = suggesting
     & td = suggestion_list  ->
     alert_check_as_type_with_suggestion_' := 3;
     final_check_as_type_with_suggestion_' := true
   [] alert_check_as_type_with_suggestion_ = 3 & ts = ok
     & td = normal  ->
     alert_check_as_type_with_suggestion_' := 4;
     final_check_as_type_with_suggestion_' := nondet


module matching_check_as_type_with_suggestion_ is
    spell_checker || check_as_type_with_suggestion

predicate pred_check_as_type_with_suggestion_ is
    (final_check_as_type_with_suggestion_ = true)

judgment J_check_as_type_with_suggestion_ is
    matching_check_as_type_with_suggestion_ |=
    pred_check_as_type_with_suggestion_
```

Program 6.10 Translation of Properties to RM and Alloy

RM:

```
predicate pred_p10_ is   ( ts = misspelled & td = normal )
judgment J_p10_ is spell_checker |= pred_p10_

predicate pred_p11_ is   ( ts = ok & td  = normal )
judgment J_p11_ is spell_checker |= pred_p11_
```

Alloy:

```
assert as_p10_ {
   no s: State | s.ts = misspelled && s.td = normal
}

assert as_p11_ {
   no s: State | s.ts = ok && s.td != normal
}
```

and C1Spell.

For the C1Spell component, although it does not specify the particular display for the misspelled words, the `type_error_action` action can be customised by the user to show different results. For example, the user can use it to beep and underline the misspelled words, or use it to highlight the misspelled word. The component leaves this flexibility to the user. Thus we accept that, with some glue code, the C1Spell component has the required behaviours.

For ChadoSpellText component, two properties are satisfied, and only one scenario `check_in_window_add_new_words` can be achieved. However, the other scenarios cannot be agreed by jMocha. This is because the component lacks some functions

that the user requires. Thus, this component does not have the required behaviour, and is rejected.

The discussion on this case study can be found at Section 6.5.

6.4 Case Study 3: Search COTS Components for a Generic e-Commerce Application

The project in this case study is from the Software Engineering subject offered by IT School, Bond University. The project is to build a generic e-commerce platform with common online shopping capabilities. The project initially focuses on each stage involved in component-based development. The "Catalysis" [55] method has been applied to the project. In the original project plan, all the components identified are built by students in order to minimise cost and gain a learning experience. In this case study, we will search and use those COTS components.

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 message centre, so that the user will be notified when the products become available. This architecture is highlighted in Figure 6.4.

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

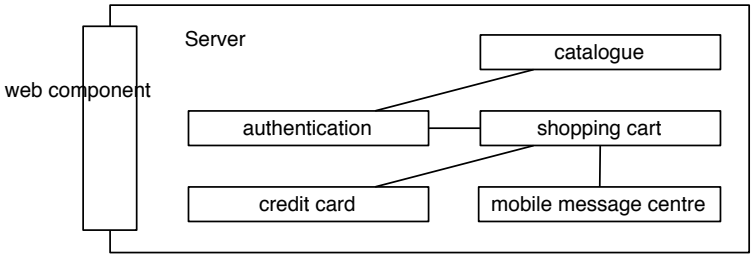

Figure 6.4: The e-Commerce Application Architecture

6.4.1 Search for the Authentication Component

We first describe the user requirements in SCIL.

6.4.1.1 The Requirement Specification

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.
- Users are able to access their own profiles.
- System administrators can manage all the users' records.

Given below (Program 6.11) is the complete SCIL specification of the requirements for the authentication component.

The type `UserLogStatus` is the user's status to the system: `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`.

The type `UserRecStatus` is the status of the user's profile: after a new user has signed up, a new profile is `added`; also the user's profile can be `modified` after the

Program 6.11 Requirement for Authentication Component

```
/*
 * Component required
 */
component UNKNOWN {
    type:
        UserLogStatus is {new, logged_in, logged_out,
              admin_logged_in};
        UserRecStatus is {na, added, modified, managed};

    variable:
        uls as UserLogStatus;
        urs as UserRecStatus;
        user_type as bool;

    service:
        Login {
        }
        SignUp {
        }
        Logout {
        }
        ChangeProfile {
        }
        ManageUsers {
        }
}

environment component web_UI connects UNKNOWN
{
    service:
        run {
            uls = logged_out -> Login;
            uls = logged_in -> Logout;
            uls = admin_logged_in -> Logout;
            uls = new -> SignUp;
            uls = logged_in -> ChangeProfile;
            uls = admin_logged_in -> ManageUsers;
        }
}

requirement authentication_requirement checks
                UNKNOWN, web_UI
{
    scenario:
        new_user_signup_change {
            uls = new;
            urs = added && uls = logged_in;
            urs = modified && uls = logged_in;
            urs = na && uls = logged_out;
        }

        admin_manage_users {
            urs = na && uls = logged_out;
            uls = admin_logged_in;
            uls = admin_logged_in && urs = managed;
            urs = na && uls = logged_out;
        }

    property:
        p1 {
            always !(uls = logged_out &&
                  (urs = added ll urs = modified));
            always !(urs = managed && uls = logged_in);
        }
}
```

user logging in; `managed` is generally 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 users' profiles.

Two scenarios are expected. The first shows such a situation: a new user successfully signs up to the system, so that the user's profile is added. Now the user is able to log in to the system, and then updates the individual profile. After it is done, the user logs out. The second scenario shows the process of the system administrator managing users' records.

The first property checks that when a new user is being added or the user's profile is being modified, the user has to be logged in to the system. The second property requires that only the administrator can manage all the users' records.

6.4.1.2 The Component Specifications

The way we specify components is similar as presented in case study 2. Program 6.12 is the specification of one candidate from the TopCoder component repository.

This component can authenticate users by checking the pair of username and password. The component also allows users to create a new account (identity), and modify the account. But only the system administrator has the privilege to delete an account.

6.4.1.3 Searching in the Repository

At the first step of searching by keywords `authentication authenticate`, we get 19 components. This number is big because many components having security features are also described by these keywords. But most of them are irrelevant to our requirements, such as RSA BSAFE Cert-C (certificate handling software library) and Rebex Secure FTP (secure file transfer component). It is not practical to view and modify a big number of specifications during the search. Thus we have to reduce the number of candidates by adding one more keyword `password`, then we get five components.

Program 6.12 Specification of an Authentication Component

```
component tc_authentication {
  type:
    UserStatus is {new, logged_in, logged_out};
    UserType is {normal, admin};
    SysStatus is {na, inserted, modified, deleted};

  variable:
    us as UserStatus;
    ss as SysStatus;
    ut as UserType;

  service:
    init {
      output: ss, us
      rule:
        true -> ss = na && us = new;
    }

    Authenticate {
      input: us
      output: us, ut
      rule:
        us = logged_out -> us = logged_in && ut = normal;
        us = logged_out -> us = logged_in && ut = admin;
    }

    InsertIdentity {
      input: us
      output: us, ss
      rule:
        us = new -> ss = inserted && us = logged_in;
    }

    EndSession {
      input: us
      output: us, ss
      rule:
        us = logged_in -> us = logged_out && ss = na;
    }

    ModifyIdentity {
      input: us
      output: ss
      rule:
        us = logged_in -> ss = modified;
    }

    DeleteIdentity {
      input: us, ut
      output: ss
      rule:
        us = logged_in && ut = admin -> ss = deleted;
    }
}
```

The searching process is similar as described in case study 2. We need to modify the requirement specification to ensure the consistent modelling of two specifications.

Since the `tc_authentication` component does not use the boolean type to decide administrator or normal user, we need to modify the user requirement specification for the component `tc_authentication`:

```
environment component web_UI connects UNKNOWN
{
    service:
        run {
            us = logged_out -> Authenticate;
            us = logged_in -> EndSession;
            us = new -> InsertIdentity;
            us = logged_in -> ModifyIdentity;
            us = logged_in && ut = admin -> DeleteIdentity;
        }
}

requirement authentication_requirement checks UNKNOWN, web_UI
{
    scenario:
        new_user_signup_change {
            us = new;
            ss = inserted && us = logged_in;
            ss = modified && us = logged_in;
            ss = na && us = logged_out;
        }
        admin_manage_users {
            ss = na && us = logged_out;
            ut = admin && us = logged_in;
            ut = admin && us = logged_in && ss = deleted;
```

```
            ss = na && us = logged_out;
        }

    property:
        p1 {
            always !(us = logged_out && (ss = inserted || ss = modified));
            always !(ss = deleted && us = logged_in && ut = normal);
        }
}
```

Please note that in the modified requirement specification, `admin_logged_in` is replaced by `us = logged_in && ut = admin` to describe the state when the system administrator has been logged in to the system.

6.4.1.4 Result

Both jMocha and Alloy Analyser advise that three components have expected behaviour, including `tc_authentication`.

6.4.2 Search for the Catalogue Component

The operations of the catalogue component include adding/removing product groups, adding/removing products, users browsing catalogue. Since browsing product catalogue does not cause state changes (not key behaviour), we will not put it into our requirement specification.

Since the authentication component has been retrieved, it should be included in the user's environment this time. The only interface between the authentication component and the catalogue component is the user's logged-in status. We import the `UserLogStatus` type from the authentication component.

The operations of the catalogue component can only be performed by the system

administrator in the logged-in status. Thus the pre-condition for all the services of catalogue component is: `uls = admin_logged_in`.

Program 6.13 is the complete requirement specification for the catalogue component.

The requirement only checks the fact that when the administrator adds/removes groups or products, their numbers cannot exceed the maximum numbers allowed, and also the numbers cannot be less than zero.

There are not many COTS catalogue components available for use. When searching our repository by keywords `catalog catalogue`, we only get two candidates. 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 candidate component, .NETCatalog, can pass the model checking by both Alloy Analyser and jMocha. However, it offers more functions that we do not need, such as creating rules, calculating discount price, etc.

6.4.3 Search for the Shopping-Cart Component

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

6.4.3.1 The Requirement Specification

The complete requirement description in SCIL for the shopping-cart component is given in Program 6.14.

The type `Quantity` represents the number of items that have already been put in the shopping cart, and `capacity` is the maximum number allowed. We give `capacity` a value of 5 as an example in this case study. `ProductStatus` denotes the status of a product, it could be in stock (`available`), or out of stock (`not_available`).

Program 6.13 Requirement Specification for Catalogue Component

```
component UNKNOWN {
   constant:
      gn_capacity = 5;
      pn_capacity = 5;

   type:
      UserLogStatus is {new, logged_in, logged_out, admin_logged_in};
      GroupNumber is (0..gn_capacity);
      ProductNumber is (0..pn_capacity);

   variable:
      uls as UserLogStatus;
      gn as GroupNumber;
      pn as ProductNumber;

   service:
      CreateGroup {
      }
      DeleteGroup {
      }
      AddProduct {
      }
      RemoveProduct {
      }
}

environment component web_UI connects UNKNOWN {
   service:
      run {
         uls = admin_logged_in -> CreateGroup;
         uls = admin_logged_in -> DeleteGroup;
         uls = admin_logged_in -> AddProduct;
         uls = admin_logged_in -> RemoveProduct;
      }
}

requirement catelog_req checks UNKNOWN, web_UI {
   property:
      p1 {
         always !(gn < 0 || gn > gn_capacity);
         always !(pn < 0 || pn > pn_capacity);
      }
      p2 {
         always uls = admin_logged_in;
      }
}
```

Program 6.14 Requirement Specification for Shopping-Cart Component

```
component UNKNOWN connects user_env {
   constant:
      capacity = 5;

   type:
      Quantity is (0..capacity);
      UserLogStatus is {new, logged_in,
            admin_logged_in, logged_out};
      ProductStatus is {not_available, available};
      ReturnMsg is {na, succeed, fail, registered};

   variable:
      q as Quantity;
      uls as UserLogStatus;
      ps as ProductStatus;
      rm as ReturnMsg;
      CC_Verify as deferred;
      MC_Register as deferred;

   service:
      AddItem {
      }
      RemoveItem {
      }
      Clear {
      }
      Checkout {
      }
}

component user_env connects UNKNOWN {
   service:
      init {
         output: rm, uls
         rule:
            true -> rm = na && uls = logged_in;
      }

      CC_Verify {
         input: rm
         output: rm
         rule:
            rm = na -> rm = succeed;
            rm = na -> rm = fail;
      }

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

```
requirement webshop checks UNKNOWN, user_env {
   scenario:
      successful_story {
         q = 0;
         q = 1;
         q = 2;
         q = 3;
         q = 0;
         q = 1 && ps = available;
         rm = succeed && q = 0;
      }

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

   property:
      p1 {
         // invariants on the number of
         // items in the shopping cart
         always !(q < 0 || q > capacity);
         // the user has to be logged in
         always uls = logged_in;
      }
      p2 {
         always !(rm = registered && q = 0);
      }
}
```

`ReturnMsg` is the list of possible messages sent by the credit card verifier and the mobile message centre components: `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.

This time the user's environment includes the authentication, the catalogue, the credit card verifier and the mobile message centre. Still we import the `UserLogStatus` type as the interface of the authentication component. Since browsing catalogue does not need to be specified and catalogue operations are only allowed by the system administrator, there are no interfaces that need to be specified 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 in the user's environment: `CC_Verify` is the service provided by the credit card verifier component, checking the validity of credit cards, and sending a 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.

The first scenario (see Figure 6.5 – a) shows: the user first puts an item into 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.

The second scenario (see Figure 6.5 – b) 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.

Two properties of the joint system by the user's environment and the potential shopping-cart component are required. The first property is to ensure the number of the items put in the shopping cart can never be less than zero, or exceed the

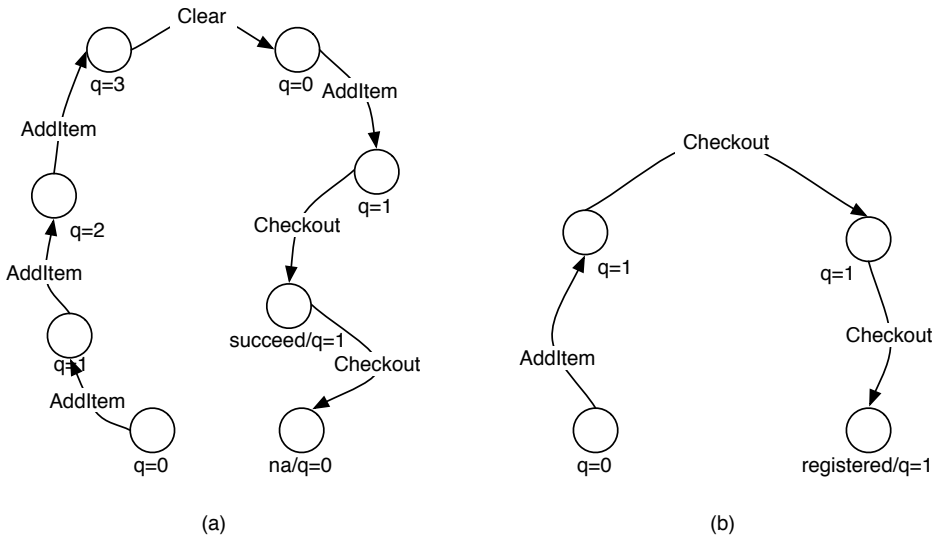

Figure 6.5: Two Scenarios for the Shopping Cart Component

maximum number allowed, which is 5 in our case. The second property is to check the fact that the shopping cart will not be cleared if the product is not available, and has been registered with the mobile message centre. The number of items in the cart remains the same, but cannot be zero.

6.4.3.2 The Component Specifications

Program 6.15 below is the specification of the component JavaCart taken from TopCoder.

It is obvious that this component does not have the interface to the mobile message centre component.

6.4.3.3 Searching in the Repository and Result

The first search by keywords `shopping cart ecommerce` has found five candidates, but none of them can be totally passed by both jMocha and Alloy Analyser. This is because no COTS shopping-cart components have an interface to our mobile message centre. For example, the JavaCart component can achieve `successful_story`

Program 6.15 Specification of JavaCart Component

```
component JavaCart connects credit_card_component {
  constant:
    capacity = 5;

  type:
    Quantity is (0..capacity);
    UserLogStatus is {new, logged_in, admin_logged_in, logged_out};
    ReturnMsg is {na, succeed, fail};

  variable:
    q as Quantity;
    uls as UserLogStatus;
    rm as ReturnMsg;
    CC_Verify as deferred;

  service:
    init {
      output: q
      rule:
        true -> q = 0;
    }

    AddItem {
      input: uls, q
      output: q
      rule:
        uls = logged_in && q < 5 -> q = q + 1;
    }

    RemoveItem {
      input: uls, q
      output: q
      rule:
        uls = logged_in && q > 0 -> q = q - 1;
    }

    Clear {
      input: uls
      output: q
      rule:
        uls = logged_in -> q = 0;
    }

    Checkout {
      input: uls, q, rm
      output: q
      rule:
        uls = logged_in && q > 0 -> CC_Verify;
        rm = succeed -> q = 0;
    }

  protocol:
    once Checkout;
    eventually Checkout;
}
```

scenario, but fail at checking `need_notification` scenario. Both jMocha and Alloy Analyser pass the first property check, but give a counterexample when checking the second property.

The shopping-cart component .netCART [119] supports an unlimited number of items. This would remove the constraint that the number of items should be within the range. Another candidate CartWIZ [30] is actually a combination of the catalogue and shopping-cart components. It provides much richer functions than we required. In order to use such a component in our application, glue code is needed [95].

6.5 Discussion

Compared 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 the 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 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 normally takes less than 20 seconds. However, it takes a 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, given the component specification is fully commented (we discuss this later in this section). 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.

Although searching by keywords is easy to use, results often contain many irrelevant items. In the first case study, we get five of 12 components (nearly 50%) that are irrelevant to spell checking. Thus, results need to be refined. The traditional way is completely manual. Users have to examine candidate components one by one, getting rid of the obvious irrelevant ones. Then users need to download the user manual files or the trial versions of those left candidates, and try one by one. For each component, we estimate based on some tests, in order to understand what it provides and how it works, it takes one person nearly one day (eight hours) approximately. It is even more time-consuming when selecting several components at the same time while these components are also connected to each other, because users need to not only try each single component, but also the assemblies of the connecting components. If each component has a few candidates, in order to pick the best group the possibilities of combining those components can consume a large amount of time and effort if doing manually.

SCIL specification matching is done semi-automatically, because it involves some manual work to make the requirement specification consistent with the component specification. But the components that combined with the user requirements, if passed by the model checking tools, can be ensured to have required behaviour that is compatible to the user's environment.

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

1. Map names between specifications developed by 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. In this case study, it causes two acceptable components to be rejected.

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 a 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 it in our framework.

One cannot easily tell the difference between SCIL specifications, thus comments for SCIL code fragments are needed. Here are example comments on the type `SpellCheckAction`:

```
//@ SpellCheckAction is a set of actions the component
//       sends when spell checking text:
// @@ no_action = no actions sent
// @@ type_error_action = the action sent when a word is misspelled
// @@ suggesting = the action sent when suggestions are given
// @@ ignoring = the action sent when the misspelled word is ignored
// @@ adding = the action sent when the new word is added to the dictionary
SpellCheckAction is {no_action, type_error_action,
     suggesting, ignoring, adding};
```

We often get the components that have more or less functions (interfaces) than we required. Assuming a component has the exact behaviour as described in the user

requirement specification, if this component now has been added more functions, when adjusting the requirement specification, normally we do not need to introduce new types (or new states), and it is likely that the model checking tools (jMocha and Alloy Analyser) still can pass the scenario and property check. This component is acceptable. In order to use such a component, glue code is required when integrating the component into our application [95].

But if now some functions have been removed from this component (in this case, the requirement has more states than the component has), when adjusting the requirement specification, normally we have to remove some types (or states). We still can map the types and variables, however, it is unlikely that the model checking tools (jMocha and Alloy Analyser) still can pass the scenario and property check. Thus this component is not acceptable according to the user requirements.

6.6 Summary

In this chapter we have presented one small case study of using our tools to check component behavioural compatibility and two full case studies of selecting commercial components from our sample repository. The first case study is the continuous work on the example from Chapter 4. The second case study is selecting a single component, while the third is searching for a group of connecting components. By experiments and comparison, we think our approach can decrease the system development time but increase the precision of component selection.

Chapter 7

Conclusion and Future Work

In this thesis, we focus on how to select components that have user-required behaviour, since component selection is a key to a component-based development (CBD). We have proposed a framework in which a collaborative process is conducted by component users and component developers to select the required components. In order to support the process, tools are provided for both users and developers. For proof of the concept purpose, we have designed Simple Component Interface Language (SCIL) as the communication and specification language to capture component behaviours. Therefore a component can be selected if it is checked behaviourally compatible with user requirement. Based on SCIL, we have developed a prototype component selection system and used it in three case studies: checking the compatibility of an auctioneer component, finding a spell checker component and searching for the components for a generic e-commerce application.

The structure of the whole thesis can be concluded as follows: In Chapter 1 we provided an introduction to the problem and an overview of the solution in our approach. In Chapter 2 we described the related work and showed that previous work is not sufficient to address our problems. In Chapter 3 we elaborated on the detailed solution framework by presenting a collaborative process to select components and the potential tools support. In Chapter 4 we introduced the syntax of SCIL lan-

guage and how to use SCIL to write component specifications and user requirement specifications. In Chapter 5 we showed the architecture of our prototype component selection system, and the way we designed and implemented it. In Chapter 6 we used three case studies to illustrate the applicability and accuracy of our approach in different applications. However, some difficulties have also been identified.

7.1 Main Contributions

The goal of our research is to maximise the possibility of finding components that have the required behaviours, so that the component adaptation cost can be minimised. The results of the case studies indicate that our approach can indeed find components that have the required behaviours. Compared to the traditional way of searching by free text, our approach can ensure the users to get more relevant results. Furthermore, with a collaborative selection process, it becomes possible for the users to receive components that exactly meet both syntactical and semantical requirements.

The main contributions of this thesis can be summarised as follows:

- We have divided the process of selecting components into two activities: first, match components by the required behaviour; second, select components by the required interface syntax. The first activity is the prerequisite of the second one. And both activities may need component developers' involvement.

- We have developed a collaborative process to select components that is different from other approaches, such as Select Perspective [6]. Our process for collaboration sets the base on a common specification language that is able to capture component behaviour. Concrete tools are provided to support such a collaborative process.

- We have designed SCIL as the bridge to other formal modelling languages, so it becomes possible to formally analyse the compatibility of components with

user requirements. While SCIL has simple, easy to understand syntax, normal users can use it without knowing the formal details.

- We have reused the model checking tools developed by other research groups in our framework. Therefore, through SCIL and its translator, one can check various behavioural properties supported by various previously existing tools.

- We have used three case studies to illustrate the applicability and accuracy of our approach. The case studies are related to different applications. We have specified user requirements and candidate components, given the translation of the combined specification, and finally shown the model checking results.

7.2 Future Work

Our future work can be carried out in the following four directions:

- Since our approach has not achieved complete automation due to the modelling inconsistency from different people, some manual work to adjust user requirements is needed when using our prototype system. We will try to solve this problem by defining a formal transformation from one model (a transition system) to another. Based on this definition, all the models can be unified.

- As the number of the model checking tools used in our framework increases, it is necessary to implement a trigger that is able to automatically select the proper tools for checking component behavioural compatibility, since different tools have different features and support checking different properties.

- Another focus will be put on the component developer's side, i.e., how to facilitate customising components. Component customisation requires investigating the technologies that are based on modelling software system families and aim at providing highly customised and optimised intermediates. An approach to this

goal is to extend SCIL to be able to specify component families. Such a specification can be parameterised, and it can be instantiated to a concrete component specification based on the particular user requirement. Different from domain specific languages, SCIL is for general purpose. Because difficulties would occur when deciding parameters for a product family, component developers should have a way to collect all the related requirements and generate the common part.

system implementation can be improved by adding the function of analysing the outputs of model checking tools, thus the services that cause the failure can be identified. When developers customising their components, SCIL specifications can provide a guide.

- Also there is a need to integrate SCIL specification and its verification into the component development process, so that any changes to component source code will cause the changes to its specification automatically, and vice versa. This needs to build a mapping from SCIL to some programming languages, such as Java.

Appendix A

Related Publications

The publications that are related to this research project are described as below:

1. Lei Wang and Padmanabhan Krishnan, An Approach to Provisioning E-Commerce Applications with Commercial Components, In *IEEE International Conference on e-Business Engineering*, pp. 323-330 (IEEE, 2006).

   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.

2. Lei Wang and Padmanabhan Krishnan, A Framework for Checking Behavioral Compatibility for Component Selection, In *Australian Software Engineering Conference, ASWEC 2006*, pp. 4960 (IEEE, 2006).

   Abstract: Component selection and composition are the main issues in Component-Based Development (CBD). Existing approaches 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.

   In this paper, we propose a framework to support a semantic description and selection of components. Towards this we first introduce a Simple Component Interface Language (SCIL). SCIL files can be translated to a variety of models including those that have a formal basis. We report our experience with two specific tools, viz., Reactive Modules and Alloy with a view to using tools based on formal methods but without exposing the details of the tools.

3. Padmanabhan Krishnan and Lei Wang, Supporting Partial Component Matching, In *Distributed Computing and Internet Technology: First International Conference, ICDCIT 2004*, volume 3347 of LNCS, pp. 294303 (Springer Verlag, 2004).

   Abstract: In this paper we define a formal framework for describing components and gaps or holes (where components can be plugged in). This is based on the theory of interface automata. The main focus is to define a component partially satisfying the requirements of a hole. A partial plug-in of a hole will result in

other holes. The definition of a partial plug-in does not result in a unique set of holes, i.e., the resulting holes can have different properties. We define an software engineering process which uses the formal framework to complete the component selection and insertion process. The process is defined in terms of the possible interactions between a component vendor and a customer seeking a component.

4. Lei Wang and Daniela Mehandjiska-Stavreva, An Initial Framework for Collaboration Based Component Selection, In *Software Engineering Research and Practice, SERP 04*, pp. 799806 (CSREA, 2004).

   Abstract: Selecting high quality components is the key to component-based system development. There is no lack of literature about evaluating and selecting components. However, most of the proposed methods only focus on the component user side, neglecting what component vendors should contribute to the selection process. This paper provides a more comprehensive component selection process based on a collaboration between component users and vendors. A new semi-formal approach to describing software components has been devised to support the collaboration. The proposed and implemented Software Component Description (SCD) tree provides a vocabulary for specifying components by users and vendors in a collaboration repository. SCD can be used as a meta-level description of components at most component marketplaces.

Appendix B

Simple Component Interface Language Grammar in SableCC

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Simple Component Interface Language (SCIL) Grammar                  *
 * for describing software component interfaces                        *
 *             Version 3.0                                             *
 *                                                                     *
 * Author: Lei Wang (Kevin)                                            *
 * Faculty of Information Technology, Bond University                  *
 * Gold Coast, QLD 4229, Australia                                     *
 *                                                                     *
 *                                                    13/09/2005       *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

Package edu.bond.it.scil;

Helpers
  ascii_small      = ['a'..'z'];
  ascii_caps       = ['A'..'Z'];
  letter           = ascii_small | ascii_caps | '_';
```

```
  zero             = '0';
  digit            = ['0'..'9'];
  nonzero_digit    = ['1'..'9'];
  any_character    = [0..0xffff];
  tab              = 9;
  lf               = 10;
  cr               = 13;
  space            = 32;
  line_terminator  = lf | cr | cr lf;
  input_character  = [any_character - ['"' + [cr + lf]]];
  not_star         = [input_character - '*'] | line_terminator;
  not_star_not_slash = [input_character - ['*' + '/']] | line_terminator;

Tokens
  /* Keywords */
  component        = 'component';
  requirement      = 'requirement';
  environment      = 'environment';
  connects         = 'connects';
  checks           = 'checks';
  constant         = 'constant';
  type             = 'type';
  variable         = 'variable';
  service          = 'service';
  init             = 'init';
  run              = 'run';
  protocol         = 'protocol';
  scenario         = 'scenario';
  property         = 'property';
  input            = 'input';
  output           = 'output';
  rule             = 'rule';
  bool             = 'bool';
  int              = 'int';
```

```
true             = 'true';
false            = 'false';
is               = 'is';
as               = 'as';
deferred         = 'deferred';

/* Temporal Operators */
deadlockfreeness = 'deadlockfreeness';
always           = 'always';
initially        = 'initially';
eventually       = 'eventually';
precedes         = 'precedes';
causedby         = 'causedby';     /* immediate effect */
alternate        = 'alternate';
once             = 'once';
before           = 'before';
after            = 'after';
until            = 'until';
between          = 'between';
be_and           = 'and';

/* Additional tokens */
l_parenthesis    = '(';
r_parenthesis    = ')';
l_brace          = '';
r_brace          = '';
l_bracket        = '[';
r_bracket        = ']';
semicolon        = ';';
comma            = ',';
colon            = ':';
dot              = '.';
range_dots       = '..';
less_than        = '<';
```

```
  less_than_equal  = '<=';
  greater_than     = '>';
  greater_than_equal = '>=';
  equal            = '=';
  not_equal        = '!=';
  not              = '!';
  and              = '&&';
  or               = '||';
  plus             = '+';
  minus            = '-';
  mult             = '*';
  div              = '/';
  mod              = '%';
  becomes          = ':=';
  leadsto          = '->';

  /* Combinations */
  identifier       = letter (letter | digit)*;
  number_literal   = nonzero_digit digit* | zero;
  blank            = (space | line_terminator | tab)+;
  end_of_line_comment = '//' input_character* line_terminator?;
  traditional_comment = '/*' not_star+ '*'+ (not_star_not_slash not_star* '*'+)* '/';
  documentation_comment = '/**' '*'* (not_star_not_slash not_star* '*'+)* '/';

Ignored Tokens
  blank,
  end_of_line_comment,
  traditional_comment,
  documentation_comment;

Productions
  specification    =
                     component_spec+
                     requirement_spec*;
```

```
component_spec   =
                   environment? component identifier connecting_comp?
                   l_brace
                        constants_dcl?
                        types_dcl?
                        variables_dcl?
                        services_dcl
                        protocols_dcl?
                   r_brace;
connecting_comp  =
                   connects identifier;
requirement_spec =
                   requirement identifier checking_comp
                   l_brace
                        constants_dcl?
                        types_dcl?
                        variables_dcl?
                        scenarios_dcl?
                        properties_dcl
                   r_brace;
checking_comp    =
                   checks component_list;
component_list   =
                   single   identifier |
                   multiple component_list comma identifier;

/* Constants */
constants_dcl    =
                   constant colon constant_dcl+;
constant_dcl     =
                   identifier equal number_literal semicolon;

/* Types */
types_dcl        =
```

```
                  type colon type_dcl+;
type_dcl         =
                  original identifier is type_def semicolon |
                  defined  [id]:identifier is [dtype]:identifier semicolon;
type_def         =
                  basic_type    basic_type |
                  compound_type compound_type;
basic_type       =
                  bool bool |
                  int  int;
compound_type    =
                  enum_type   enum_type |
                  range_type  range_type |
                  struct_type struct_type;
array_type       =
                  l_bracket number_literal r_bracket;
enum_type        =
                  l_brace nonempty_enum_val_list r_brace;
nonempty_enum_val_list =
                  multiple nonempty_enum_val_list comma identifier |
                  single   identifier;
range_type       =
                  l_parenthesis [start]:range_start_end range_dots
                                [end]:range_start_end r_parenthesis;
range_start_end  =
                  numbers  number_literal |
                  identifiers identifier;
struct_type      =
                  l_brace type_dcl+ r_brace;

/* Variables */
variables_dcl    =
                  variable colon variable_dcl+;
variable_dcl     =
```

```
                  basic_types identifier as basic_type array_type? semicolon |
                  defined     [var_id]:identifier as
                                [type_id]:identifier array_type? semicolon |
                  deferred    identifier as deferred semicolon;

/* Service */
services_dcl     =
                  service colon service_dcl;
service_dcl      =
                  normal init_service? normal_service+ |
                  env    run_service;
init_service     =
                  init l_brace inputs_dcl? outputs_dcl rules_dcl r_brace;
normal_service   =
                  identifier l_brace inputs_dcl? outputs_dcl? rules_dcl? r_brace;
run_service      =
                  run l_brace run_rule_dcl+ r_brace;
inputs_dcl       =
                  input colon para_dcl_list;
outputs_dcl      =
                  output colon para_dcl_list;
para_dcl         =
                  identifier  identifier |
                  basic_types identifier as basic_type  |
                  defined     [para_id]:identifier as [type_id]:identifier;
para_dcl_list    =
                  single   para_dcl |
                  multiple para_dcl_list comma para_dcl;
rules_dcl        =
                  rule colon rule_dcl+;
rule_dcl         =
                  [assumption]:or_expression leadsto
                  [guarantee]:or_expression semicolon;
run_rule_dcl     =
```

```
                  or_expression leadsto service_name_list semicolon;

/* Protocols */
protocols_dcl    =
                  protocol colon protocol_dcl+;
protocol_dcl     =
                  temporal_expression semicolon;

 /* Scenarios */
scenarios_dcl    =
                  scenario colon scenario_dcl+;
scenario_dcl     =
                  identifier l_brace step_dcl+ r_brace;
step_dcl         =
                  or_expression semicolon;

/* Properties */
properties_dcl   =
                  property colon property_dcl+;
property_dcl     =
                  identifier l_brace predicate_dcl+ r_brace;
predicate_dcl    =
                  property_pattern property_pattern semicolon |
                  deadlockfreeness deadlockfreeness semicolon;

/* Arithmetic Expressions */
unary_expression =
   plus       plus unary_expression |
   minus      minus unary_expression |
   not        not unary_expression |
   literal    literal |
   parenthese l_parenthesis or_expression r_parenthesis |
   name       name;
multiplicative_expression =
```

```
   unary_expression unary_expression |
   multi            multiplicative_expression mult unary_expression |
   div              multiplicative_expression div unary_expression |
   mod              multiplicative_expression mod unary_expression;
additive_expression       =
   multiplicative_expression multiplicative_expression |
   plus             additive_expression plus multiplicative_expression |
   minus            additive_expression minus multiplicative_expression;
relational_expression     =
   additive_expression additive_expression |
   less_than           relational_expression less_than additive_expression |
   greater_than        relational_expression greater_than additive_expression |
   less_than_equal     relational_expression less_than_equal additive_expression |
   greater_than_equal  relational_expression greater_than_equal additive_expression;
equality_expression       =
   relational_expression relational_expression |
   equal                  equality_expression equal relational_expression |
   not_equal              equality_expression not_equal relational_expression;
and_expression            =
   equality_expression equality_expression |
   and_expression      and_expression and equality_expression;
or_expression             =
   and_expression and_expression |
   or_expression  or_expression or and_expression;
expression                =
   or_expression or_expression |
   becomes    name becomes or_expression;

/* Temporal Expressions */
unary_temporal_operator   =
   always     always |
   once       once |
   initially  initially |
   eventually eventually;
```

```
binary_temporal_operator  =
   causedby  causedby |
   precedes  precedes |
   alternate alternate |
   until     until;
temporal_expression       =
   unary_temporal_exp  unary_temporal_operator service_name_list |
   binary_temporal_exp [left]:service_name_list
                binary_temporal_operator [right]:service_name_list;
service_name_list         =
   identifier        identifier |
   service_name_list service_name_list comma identifier;
property_pattern          =
   always      always p_statement |
   before      [left]:p_statement before [right]:p_statement |
   after       [left]:p_statement after [right]:p_statement |
   between_and [left]:p_statement between [right1]:p_statement be_and
                                          [right2]:p_statement |
   after_until [left]:p_statement after [right1]:p_statement until
                                          [right2]:p_statement;
p_statement               =
   unary    expression |
   precedes [left]:expression_list precedes [right]:expression_list |
   causedby [left]:expression_list causedby [right]:expression_list;
expression_list           =
   single   expression |
   multiple expression_list comma expression;

/* Literals */
literal          =
                  number_literal number_literal |
                  bool_literal   bool_literal;
bool_literal     =
                  true  true |
```

```
                  false false;
name              =
                  simple_name    simple_name |
                  qualified_name qualified_name;
simple_name       =
                  identifier array_type?;
qualified_name    =
                  name dot identifier;
```

Appendix C

Auctioneer Component Specification and Its User Requirement

C.1 The SCIL Specification of the Auctioneer Component

```
component auctioneer
{
   // type definitions
   type:
      BidderStatus is {logged_in, logged_out, win, not_win};
      ProductStatus is {not_available, available, engaged, sold};

   // global variables
   variable:
      bs as BidderStatus;
      ps as ProductStatus;
```

```
// service definitions
service:
   init {
     output: bs, ps
     rule:
     true -> bs = logged_out && ps = not_available;
   }
   sell {
     output: ps
     rule:
     true -> ps = available;
   }
   login {
     input: bs, ps
     output: bs
     rule:
     ps != not_available && bs = logged_out -> bs = logged_in;
   }
   logout {
     input: bs
     output: bs
     rule:
     bs != logged_out -> bs = logged_out;
   }
   purchase {
     input: bs, ps
     output: bs, ps
     rule:
     bs = win && ps = engaged -> ps = sold;
   }
   bid {
     input: bs, ps
     output: bs, ps
     rule:
```

```
        bs = logged_in && ps = available -> bs = win && ps = engaged;
        bs = logged_in && ps = available -> bs = not_win && ps = engaged;
      }

   // protocol definitions
   protocol:
      initially sell;
      once sell;
      bid precedes purchase;
      eventually logout;
}
```

C.2 The SCIL Specification of the User Requirement for the Auctioneer Component

```
/*
 * Environment component bidder
 */
environment component bidder connects auctioneer
{
   variable:
      b_bs as BidderStatus;
      b_ps as ProductStatus;

   // special service for an environment component
   service:
      run {
        b_bs = logged_out && b_ps != not_available -> login;
        b_bs = logged_in && b_ps = available -> bid || logout;
        b_bs = win && b_ps = engaged -> purchase;
```

```
        b_bs = win && b_ps = sold -> logout;
        b_bs = not_win && b_ps = engaged -> logout;
        b_bs = logged_in && (b_ps = engaged || b_ps = sold) -> logout;
      }
}

/*
 * Environment component seller
 */
environment component seller connects auctioneer
{
   service:
      run {
        true -> sell;
      }
}

requirement auctioneer checks auctioneer, bidder, seller
{
   variable:
      r_bs as BidderStatus;
      r_ps as ProductStatus;

   // scenario definitions
   scenario:
      best_story {
        r_bs = logged_out && r_ps = not_available;
        r_bs = logged_out && r_ps = available;
        r_bs = logged_in && r_ps = available;
        r_bs = win && r_ps = engaged;
        r_bs = win && r_ps = sold;
        r_bs = logged_out && r_ps = sold;
      }
      another_story {
```

```
        r_bs = logged_out && r_ps = not_available;
        r_bs = logged_out && r_ps = available;
        r_bs = logged_in && r_ps = available;
        r_bs = not_win && r_ps = engaged;
        r_bs = logged_out && r_ps = engaged;
      }

   // property definitions
   property:
      p1 {
        // user won't login if the product is not available
        always !(r_bs = logged_in && r_ps = not_available);
        // user can bid only after they have logged in and before finally log out
        (r_bs = win || r_bs = not_win) between (r_bs = logged_in)
                                        and (r_bs = logged_out);
        // product can be purchased after it has been won by someone
        (r_ps = engaged) precedes (r_ps = sold) after (r_ps = available);
        // user eventually logout
        (r_bs = logged_out) after (r_ps = sold || r_ps = engaged);
      }
      p2 {
        deadlockfreeness;
      }
}
```

C.3 The RM Translation of the Combined Specification

```
type BidderStatus is logged_in, logged_out, win, not_win
type ProductStatus is not_available, available, engaged, sold
```

```
module auctioneer is
interface ps: ProductStatus; bs: BidderStatus
external sell, bid, logout, purchase, login: event
private sell_num_: (0..1)

lazy atom main controls ps, bs, sell_num_
reads ps, bs, sell, bid, logout, purchase, login, sell_num_
awaits sell, bid, logout, purchase, login

init
[] true -> bs' := logged_out; ps' := not_available; sell_num_' := 0
update
[] bs = logged_in & ps = available & bid? -> bs' := win; ps' := engaged
[] bs = win & ps = engaged & purchase? -> ps' := sold
[] bs ~= logged_out & logout? -> bs' := logged_out
[] bs = logged_in & ps = available & bid? -> bs' := not_win; ps' := engaged
[] ps ~= not_available & bs = logged_out & login? -> bs' := logged_in
[] true & sell? & sell_num_ < 1 & bs = logged_out & ps = not_available ->
   ps' := available; sell_num_' := inc sell_num_ by 1


module bidder is
interface purchase: event; logout: event; bid: event; login: event
external ps: ProductStatus; bs: BidderStatus;
lazy atom main controls purchase, logout, bid, login
reads purchase, logout, bid, login, ps, bs

initupdate
[] bs = logged_in & ps = available -> bid!
[] bs = logged_in & ps = available -> logout!
[] bs = win & ps = sold -> logout!
[] bs = logged_in & ( ps = engaged | ps = sold ) -> logout!
[] bs = logged_out & ps ~= not_available -> login!
```

```
[] bs = win & ps = engaged -> purchase!
[] bs = not_win & ps = engaged -> logout!


module seller is
interface sell: event
lazy atom main controls sell
reads sell

initupdate
[] true -> sell!


module auctioneer_sys is auctioneer || bidder || seller

module another_story is
interface alert_another_story_: (0..5); final_another_story_: bool
external ps: ProductStatus; bs: BidderStatus;
lazy atom main controls alert_another_story_, final_another_story_
reads alert_another_story_, ps, bs
init
[] true -> alert_another_story_' := 0; final_another_story_' := true
update
[] alert_another_story_ = 0 & bs = logged_out & ps = not_available  ->
   alert_another_story_' := 1; final_another_story_' := true
[] alert_another_story_ = 1 & bs = logged_out & ps = available  ->
   alert_another_story_' := 2; final_another_story_' := true
[] alert_another_story_ = 2 & bs = logged_in & ps = available  ->
   alert_another_story_' := 3; final_another_story_' := true
[] alert_another_story_ = 3 & bs = not_win & ps = engaged  ->
   alert_another_story_' := 4; final_another_story_' := true
[] alert_another_story_ = 4 & bs = logged_out & ps = engaged  ->
   alert_another_story_' := 5; final_another_story_' := nondet
```

```
module matching_another_story_ is auctioneer_sys || another_story
predicate pred_another_story_ is (final_another_story_ = true)
judgment J_another_story_ is matching_another_story_ |= pred_another_story_


module best_story is
interface alert_best_story_: (0..6); final_best_story_: bool
external ps: ProductStatus; bs: BidderStatus;
lazy atom main controls alert_best_story_, final_best_story_
reads alert_best_story_, ps, bs
init
[] true -> alert_best_story_' := 0; final_best_story_' := true
update
[] alert_best_story_ = 0 & bs = logged_out & ps = not_available  ->
   alert_best_story_' := 1; final_best_story_' := true
[] alert_best_story_ = 1 & bs = logged_out & ps = available  ->
   alert_best_story_' := 2; final_best_story_' := true
[] alert_best_story_ = 2 & bs = logged_in & ps = available  ->
   alert_best_story_' := 3; final_best_story_' := true
[] alert_best_story_ = 3 & bs = win & ps = engaged  ->
   alert_best_story_' := 4; final_best_story_' := true
[] alert_best_story_ = 4 & bs = win & ps = sold  ->
   alert_best_story_' := 5; final_best_story_' := true
[] alert_best_story_ = 5 & bs = logged_out & ps = sold  ->
   alert_best_story_' := 6; final_best_story_' := nondet

module matching_best_story_ is auctioneer_sys || best_story
predicate pred_best_story_ is (final_best_story_ = true)
judgment J_best_story_ is matching_best_story_ |= pred_best_story_


predicate pred_p10_ is  ~ ( bs = logged_in & ps = not_available )
judgment J_p10_ is auctioneer_sys |= pred_p10_
```

C.4 The Alloy Translation of the Combined Specification

```
module auctioneer
open util/ordering[State] as ord

abstract sig ProductStatus
one sig not_available, available, engaged, sold extends ProductStatus
abstract sig BidderStatus
one sig logged_in, logged_out, win, not_win extends BidderStatus

sig State
ps: ProductStatus,
bs: BidderStatus


pred logout (ps, ps': ProductStatus, bs, bs': BidderStatus)
bs != logged_out => bs' = logged_out && ps' = ps


pred bid (ps, ps': ProductStatus, bs, bs': BidderStatus)
bs = logged_in && ps = available => bs' = not_win && ps' = engaged


pred purchase (ps, ps': ProductStatus, bs, bs': BidderStatus)
bs = win && ps = engaged => ps' = sold && bs' = bs
```

```
pred bid0 (ps, ps': ProductStatus, bs, bs': BidderStatus)
bs = logged_in && ps = available => bs' = win && ps' = engaged


pred login (ps, ps': ProductStatus, bs, bs': BidderStatus)
ps != not_available && bs = logged_out => bs' = logged_in && ps' = ps


pred sell (ps, ps': ProductStatus, bs, bs': BidderStatus)
ps' = available && bs' = bs


pred Initialisation (s: State)
s.bs = logged_out && s.ps = not_available


pred Transition (s, s': State)
s.bs != logged_out => logout(s.ps, s'.ps, s.bs, s'.bs)
s.bs = logged_in && s.ps = available => bid(s.ps, s'.ps, s.bs, s'.bs)
s.bs = win && s.ps = engaged => purchase(s.ps, s'.ps, s.bs, s'.bs)
s.bs = logged_in && s.ps = available => bid0(s.ps, s'.ps, s.bs, s'.bs)
s.ps != not_available && s.bs = logged_out => login(s.ps, s'.ps, s.bs, s'.bs)
sell(s.ps, s'.ps, s.bs, s'.bs)


fact Execution
Initialisation (ord/first())
all s: State - ord/last() | let s' = ord/next(s) | Transition (s, s')


assert p1
all s: State | ! ( s.bs = logged_in && s.ps = not_available )
```

```
check p1 for 8 State
```

Bibliography

[1] J. Aagedal and E. F. Ecklund. Modelling QoS: Towards a UML Profile. In *the 5th International Conference on Unified Modeling Language (UML 2002)*, volume 2460/2002, pages 275–289. Springer Berlin / Heidelberg, 2002.

[2] C. Abst, B. Boehm, and E.B. Clark. Empirical Observations on COTS Software Integration Effort Based on the Initial COCOTS Calibration Database. In *the 22nd International Conference on on Software Engineering (ICSE 2000) COTS Workshop*, Limerick, Ireland, 2000.

[3] B. Thomas Adler, L. de Alfaro, L. Dias Da Silva, M. Faella, A. Legay, V. Raman, and P. Roy. Ticc: A Tool for interface Compatibility and Composition. Technical report, School of Engineering, University of California, Santa Cruz, 2006.

[4] 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.

[5] C. Albert and L. Brownsword. Evolutionary Process for Integrating COTS-Based Systems (epic): An Overview. Technical Report CMU/SEI-2002-TR-009, Software Engineering Institute, Carnegie Mellon University, 2002.

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

[7] R. Allen, R. Douence, and D. Garlan. Specifying Dynamism in Software Architectures. In *Foundations of Component-Based Systems Workshop*, Zurich, Switzerland, 1997.

[8] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, 1997.

[9] R. Alur, L. de Alfaro, R. Grosuz, T.A. Henzinger, M. Kangy, R. Majumdar, F. Mang, C.M. Kirsch, and B.Y. Wangy. *Mocha Manual*. Computer Science Department, Stony Brook University, 2000.

[10] R. Alur, L. de Alfaro, R. Grosuz, T.A. Henzinger, M. Kangy, R. Majumdar, F. Mang, C.M. Kirsch, and B.Y. Wangy. jMocha: A Model Checking Tool that Exploits Design Structure. In *the 23rd IEEE/ACM International Conference on Software Engineering (ICSE 2001)*, pages 835–836, Toronto, Canada, 2001.

[11] R. Alur and T. A. Henzinger. Reactive Modules. *Formal Methods in System Design*, 15:7–48, 1999.

[12] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume II: Technical Concepts of Components-Based Software Engineering, 2nd Edition. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.

[13] F. Bachmann and G. Chastek L. Bass. The Architecture Based Design Method. Technical report, Software Engineering Institute, Carnegie Mellon University, 2000.

[14] M. Barnett, R. DeLine, M. Fahndrich, KRM Leino, and W. Schulte. Verification of Object-oriented Programs with Invariants. *Journal of Object Technology*, 3:27–56, 2004.

[15] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *LNCS*, Marseille, France, 2004. Springer.

[16] N. Barthwal and M. Woodside. Efficient Evaluation of Alternatives for Assembly of Services. In *the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 275.1, Washington DC, USA, 2005. IEEE.

[17] R. Bastide, O. Sy, and P. Palanque. Formal Specification and Prototyping of CORBA Systems. In *the 13th European Conference on Object-Oriented Programming*, pages 474–494, London, UK, 1999. Springer-Verlag.

[18] R. Bergmann, S. Schmitt, and A. Stahl. Intelligent Customer Support for Product Selection with Case-Based Reasoning. *E-Commerce and Intelligent Methods*, 105:322–341, 2002.

[19] A. Beugnard, J. Jezequel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[20] B. W. Boehm, D. Port, Y. Yang, and J. Bhuta. Not All CBS Are Created Equally: COTS-Intensive Project Types. In *the 2nd International Conference on COTS-Based Software Systems (ICCBSS '03)*, pages 36–50, London, UK, 2003. Springer-Verlag.

[21] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[22] S. Boonsiri, R. C. Seacord, and R. Bunting. Automated Component Ensemble Evaluation. *International Journal of Information Technology*, 8(1), 2002.

[23] N. Boyette. Reusable asset specification repository for workgroups : Overview. *IBM alphaWorks*, 2005.

[24] T. Brijs, B. Goethals, G. Swinnen, K. Vanhoof, and G. Wets. A Data Mining Framework for Optimal Product Selection in Retail Supermarket Data: the Generalized profset Model. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 300–304. ACM Press, New York, NY, USA, 2000.

[25] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets. Using Association Rules for Product Assortment Decisions: A Case Study. In *the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data mining (KDD '99)*, pages 254–260, New York, NY, USA, 1999. ACM Press.

[26] F. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[27] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[28] 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.

[29] C. Canal, E. Pimentel, J. M. Troya, and A. Vallecillo. Extending CORBA Interfaces with Protocols. *The Computer Journal,*, 44(5):448–462, 2001.

[30] CartWIZ. http://www.cartwiz.com/. 2005-2006.

[31] ChadoSoftware. http://www.chado-software.com. 2006.

[32] A. Chakrabarti. *CHIC: Checker for Interface Compatibility*. Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2003.

[33] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, Boston, MA, 2001.

[34] S. Chen, I. Gorton, A. Liu, and Y. Liu. Performance Prediction of COTS Component-based Enterprise Applications. In *the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly*, Orlando, Florida USA, 2002.

[35] Y. Chen and H. C. Cheng. A Semantic Foundation for Specification Matching. In G. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 91–109. Cambridge University Press, 2000.

[36] I. Cho. A Framework for the Specification and Testing of the Interoperation Aspect of Components. In *ECOOP Workshop on Object Interoperability*, pages 53–64, Sophia Antipolis, France, 2000.

[37] L. Chung and K. Cooper. A Knowledge-based COTS-aware Requirements Engineering Approach. In *the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 175–182, Ischia, Italy, 2002.

[38] L. Chung, W. Ma, and K. Cooper. Requirements Elicitation through Model-Driven Evaluation of Software Components. In *the 5th International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS 2006)*, page 10, Orlando, Florida USA, 2006.

[39] P. Ciancarini and S. Cimato. Specifying Component-based Software Architectures. In *Foundations of Component-Based Systems Workshop*, pages 60–70, Zurich, Switzerland, 1997.

[40] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA.

[41] P. C. Clements. A Survey of Architecture Description Languages. In *the 8th International Workshop on Software Specification and Design*, pages 16–25, Schloss Velen, Germany, 1996.

[42] ComponentSource. http://www.componentsource.com. 1996-2006.

[43] Microsoft Corporation. The Component Object Model Specification, 1995.

[44] Michael L. Creech, Dennis F. Freeze, and Martin L. Griss. Using Hypertext in Selecting Reusable Software Components. In *the 3rd Annual ACM Conference on Hypertext*, San Antonio, TX USA, 1991.

[45] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based Development Process and Component Lifecycle. In *International Conference on Software Engineering Advances (ICSEA'06)*, Tahiti, French Polynesia, October 2006. IEEE.

[46] I. Crnkovic and M. Larssom. Challenges of Component-Based Development. *Journal of Systems and Software*, 61(3):201–212, 2002.

[47] I. Crnkovic and M. Larsson, editors. *Building Reliable Component-based Software Systems*. Artech House Publishers, 2002.

[48] I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau. Component Certification and System Predication. In *the 4th ICSE Workshop on Component-Based Software Engineering*, page 97, Toronto, Canada, 2001.

[49] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[50] E. Damiani and M. G. Fugini. Automatic Thesaurus Construction Supporting Fuzzy Retrieval of Reusable Components. In *the ACM Symposium on Applied Computing*, pages 542–547, Tennessee, US, 1995.

[51] E. Damiani, M. G. Fugini, and C. Bellettini. A Hierarchy-Aware Approach to Faceted Classification of Objected-Oriented Components. *ACM Transactions on Software Engineering and Methodology*, 8(3):215–262, 1999.

[52] L. de Alfaro and T. A. Henzinger. Interface Automata. In *the 9th Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[53] L. de Alfaro and T. A. Henzinger. Interface Theories for Component-Based Design. *Lecture Notes in Computer Science*, 2211:148–165, 2001.

[54] C. Demartini, R. Sisto, and R. Iosif. A Concurrency Analysis Tool for Java Programs. Technical report, System and Computer Engineering Department, Polytechnic of Turin, 1997.

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

[56] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. In *the 2nd workshop on Formal Methods in Software Practice (FMSP '98)*, pages 7–15, New York, NY, USA, 1998. ACM Press.

[57] P. Eeles, K. A. Houston, and W. Kozaczynski. *Building J2EE Applications with the Rational Unified Process*. Addison Wesley Professional, 2002.

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

[59] B. Fischer. Specification-Based Browsing of Software Component Libraries. *Journal of Automated Software Engineering*, 7:179–200, 2000.

[60] International Organization for Standardization. Information Technology: Open Distributed Processing - Reference Model - Quality of Service, ISO Document ISO/IEC JTC1/SC7 N1996, 1998.

[61] W. B. Frakes and T. P. Pole. Proteus: A Software Reuse Library System that Supports Multiple Representation Methods. *ACM SIGIR Forum*, 24:43–55, 1990.

[62] William B Frakes and Thomas P Pole. An Empirical Study of Representation Methods for Reusable Software Components. *IEEE Transactions on Software Engineering*, 20:617–631, 1994.

[63] S. Frolund and J. Koistinen. Quality of Service Specification in Distributed Object Systems. *Distributed Systems Engineering Journal*, 5(4):179–202, 1998.

[64] Etienne Gagnon. *SableCC, an Object-Oriented Compiler Framework*. Master thesis, McGill University, Montreal, 1998.

[65] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or, Why it's hard to build systems out of existing parts. *IEEE Software*, 12(6):17–26, 1995.

[66] CJM Geisterfer and S. Ghosh. Software Component Specification: A Study in Perspective of Component Selection and Reuse. In *International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*, volume 0, pages 100–108, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[67] B. Genest and A. Muscholl. Message Sequence Charts: A Survey. In *the 5th International Conference on Application of Concurrency to System Design (ACSD 2005)*, pages 2–4, St. Malo, France, 2005.

[68] M. R. Girardi and B. Ibrahim. A Software Reuse System Based on Natural Language Specifications. In *the Fifth International Conference on Computing and Information (ICCI '93)*, pages 507–511, Washington, DC, USA, 1993. IEEE Computer Society.

[69] M.R. Girardi and B. Ibrahim. A Similarity Measure for Retrieving Software Artifacts. In *the 6th International Conference on Software Engineering and Knowledge Engineering (SEKE'94)*, pages 478–485, Jurmala, Latvia, 1994.

[70] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Journal on Software and System Modelling*, 4:386–398, 2005.

[71] S.J. Goldsack, K. Lano, and E. Der. Invariants as Design Templates in Object-based Systems. In *Foundations of Component-Based Systems Workshop*, Zurich, Switzerland, 1997.

[72] P. Gomes, F. Pereira, P. Paiva, N. Seco, P. Carreiro, F. Ferreira, and C. Bento. Using WorldNet for Case-Based Retrieval of UML Methods. *AI Communications*, 17:1323, 2004.

[73] P. Graubmann, E. Rudolph, and J. Grabowski. Component Interface Description Using HyperMSCs and Connectors. In *Symposium on Human Centric Computing Languages and Environments (HCC'01)*, volume 0, page 96, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[74] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories*. Wiley, 2004.

[75] Object Management Group. CORBA Components, Version 3.0, 2002.

[76] Object Management Group. CORBA Specification 3.0, 2002.

[77] Object Management Group. UML 2.0 OCL Specification, 2003.

[78] Object Management Group. Reusable Asset Specification, Version 2.2, 2005.

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

[80] J. Han and K. K. Ker. Ensuring Compatible Interactions within Component-based Software Systems. In *the 10th Asia-Pacific Software Engineering Conference*, volume 00, page 436, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[81] D. Hemer. Specification Matching of State-Based Modular Components. In *the 10th Asia-Pacific Software Engineering Conference Software Engineering Conference (APSEC '03)*, page 446, Washington, DC, USA, 2003. IEEE Computer Society.

[82] D. Hemer and P. Lindsay. Specification-Based Retrieval Strategies for Module Reuse. In D. Grant and L. Stirling, editors, *Australian Software Engineering Conference (ASWEC 2001)*, pages 235–243, Canberra, Australia, 2001. IEEE Computer Society.

[83] S. Hissam, R. Seacord, and K. Wallnau. *Building Systems from Commercial Components*. Addison Wesley Professional, 2001.

[84] S. A. Hissam, G. A. Moreno, J. Stafford, and K. C. Wallnau. Packaging predictable assembly with prediction-enabled component technology. Technical Report CMU/SEI-2001-TR-024, Software Engineering Institute, Carnegie Mellon University, 2001.

[85] L. Iribarne, J. M. Troya, and A. Vallecillo. A Trading Service for COTS Components. *The Computer Journal*, 47:342–357, 2003.

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

[87] D. Jackson. *Alloy 3.0 Reference Manual*, 2004.

[88] H. Jain, N. Chalimeda, N. Ivaturi, and B. Reddy. Business Component Identification - A Formal Approach. In *the 5th IEEE International Conference on Enterprise Distributed Object Computing*, pages 183–187, Seattle, WA USA, 2001. IEEE Computer Society.

[89] Y. Jin. *Compositional Verification of Component-Based Heterogeneous Systems*. PhD thesis, School of Computer Science, the University of Adelaide, 2004.

[90] Y. Jin and J. Han. Specifying Interaction Constraints of Software Components for Better Understandability and Interoperability. In *the 4th International Conference on COTS-Based Software Systems (ICCBSS 2005)*, pages 54–64, Bilbao, Spain, 2005.

[91] Y. Jin, C. Lakos, and R. Esser. Component-based Design and Analysis: A Case Study. In *the International Conference on Software Engineering and Formal Methods*, Brisbane, Australia, 2003. IEEE Computer Society Press.

[92] D. R. Johnson and H. Kilov. An Approach to an RM-ODP Toolkit in Z. In *Foundations of Component-Based Systems Workshop*, Zurich, Switzerland, 1997.

[93] Keyoti. http://www.keyoti.com/. 2002-2006.

[94] J. Kontio, G. Galdiera, and V. R. Basili. Defining Factors, Goals and Criteria for Reusable Component Evaluation. In *the 1996 conference of the Centre for Advanced Studies on Collaborative research (CASCON '96)*, Toronto, Canada, 1996.

[95] P. Krishnan and L. Wang. Supporting Partial Component Matching. In *the 1st International Conference on Distributed Computing and Internet Technology (ICDCIT 2004)*, pages 294–303, Bhubaneswar, India, 2004.

[96] P. Kruchten. Architectural Blueprints - The "4+1" View Model of Software Architecture. *IEEE Software*, 12:42–50, 1995.

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

[98] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2003.

[99] K. K. Lau and M. Ornaghi. *Logic for Component-based Software Development*, pages 347–373. Computational Logic: Logic Programming and Beyond, Lecture Notes in Artificial Intelligence 2407. Springer-Verlag, 2002.

[100] G. Leavens and Y. Cheon. Design by Contract with JML, 2003.

[101] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical report, Department of Computer Science, Iowa State University, 1999.

[102] ComponentOne LLC. http://www.componentone.com/. 1987-2006.

[103] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and Measuring Quality of Service in Distributed Object Systems. In *Object-Oriented Real-Time Distributed Computing Proceedings*, pages 43–52, Kyoto, Japan, 1998.

[104] M. Lumpe, J. G. Schneider, O. Nierstrasz, and F. Achermann. Towards A Formal Composition Language. In *Foundations of Component-Based Systems Workshop*, Zurich, Switzerland, 1997.

[105] N. A. Maiden and C. Ncube. Acquiring COTS Software Selection Requirements. *IEEE Software*, 15(2):46–56, 1998.

[106] E. Mancebo and A. Andrews. A Strategy for Selecting Multiple Components. In *the 2005 ACM Symposium on Applied Computing*, pages 1505–1510, New York, USA, 2005. ACM Press.

[107] G. Martin, R. Seepold, T. Zhang, L. Benini, and G. De Micheli. Component Selection and Matching for IP-based Design. In *the conference on Design, Automation and Test in Europe (DATE '01)*, pages 40–46, Piscataway, NJ, USA, 2001. IEEE.

[108] M. Matsushita. Ranking Significance of Software Components Based on Use Relations. *IEEE Transactions on Software Engineering*, 31(3):213–225, 2005. Member-Katsuro Inoue and Member-Reishi Yokomori and Member-Tetsuo Yamamoto and Member-Shinji Kusumoto.

[109] V. Maxville, J. Armarego, and C. P. Lam. Intelligent Component Selection. In *the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, pages 244–249, Washington, DC, USA, 2004. IEEE.

[110] M.D. McIlroys. Mass-produced software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155. Scientific Affairs Division, NATO, 1968.

[111] R. Meling, E. J. Montgomery, P. S. Ponnusamy, E. B. Wong, and D. Mehandjiska. Storing and Retrieving Software Components: A Component Description Manager. In *Australian Software Engineering Conference*, Gold Coast, Queensland, Australia, 2000.

[112] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall International, London, UK, 2nd edition, 1997.

[113] Sun Microsystems. JavaBeans API Specification, Version 1.01, 1997.

[114] Sun Microsystems. Enterprise JavaBeans Specification, Version 2.1, 2002.

[115] Sun Microsystems. J2EE Platform Specification 1.4, 2003.

[116] Af. Mili, F. Mili, and Al. Mili. Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, 1995.

[117] R. Mili, A. Mili, and R. T. Mittermeir. Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions on Software Engineering*, 23:445–460, 1997.

[118] M. Morisio and A. Tsoukias. Iusware: A Methodology for the Evaluation and Selection of Software Products. *IEE Proceedings Software Engineering*, 144:162–174, 1997.

[119] .netECOMMERCE. http://www.dotnetecommerce.com/. 2001-2006.

[120] O. Nierstrasz. Regular Types for Active Objects. In *the 8th annual conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pages 1–15, New York, NY, USA, 1993. ACM Press.

[121] 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)*, Portland, Oregon USA, 2003. IEEE Press.

[122] P. Oberndorf, L. Brownsword, E. Morris, and C. Sledge. Workshop on COTS-Based Systems. Technical report, Software Engineering Institute, Carnegie Mellon University, Nov. 1997.

[123] M. Ochs, D. Pfahl, and G. Chrobok-Diening. A COTS Acquisition Process: Definition and Application Experience. In *the 11th ESCOM Conference*, pages 335–343, Shaker, Maastricht, 2000.

[124] M. Paludo, R. Burnett, and S. Reinehr. Applying Pattern Techniques to Leverage Component-based Development. In *Advances in Computer Science and Technology*, 2006.

[125] J. Penix and P. Alexander. Rebound: A framework for automated component adaptation. In *Component-Based Software Engineering: Case Studies*, chapter 10. World Scientific, 2004.

[126] F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[127] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In D. Chan and A. Watson, editors, *the 4th Smart Card Research and Advanced Application Conference (CARDIS)*, pages 135–154. Kluwer Academic Press, 2000.

[128] P. Popic, D. Desovski, W. Abdelmoez, and B. Cukic. Error Propagation in the Reliability Analysis of Component Based Systems. In *the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 53–62, Chicago, Illinois, USA, 2005. IEEE Computer Society.

[129] R. Prieto-Diaz. Domain Analysis for Reusability. In *IEEE Computer Software and Applications Conference*, Los Alamitos, CA USA, 1987.

[130] Ruben Prieto-Diaz. Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, 34(5):89–97, 1991.

[131] R. Reussner. Enhanced Component Interfaces to Support Dynamic Adaption and Extension. In *the 34th Annual Hawaii International Conference on System Sciences (HICSS '01)*, volume 9, page 9043, Washington, DC, USA, 2001. IEEE Computer Society.

[132] RicherComponents. http://www.rickercomponents.com. 2004-2005.

[133] T. L. Saaty. *Decision Making for Leaders: The Analytic Hierarchy Process for Decisions in a Complex World*. RWS Publications, 1999.

[134] D. C. Schmidt, R. E. Johnson, and M. Fayad. Software Patterns. *Communications of the ACM*, 39:37–39, 1996.

[135] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an emerging discipline*. Prentice Hall, 1996.

[136] J. Stafford and J. D. McGregor. Issues in Predicting the Reliability of Composed Components. In *the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly*, Orlando, Florida, USA, 2002.

[137] V. Sugumaran and V. C. Storey. A Semantic-Based Approach to Component Retrieval. *SIGMIS Database*, 34(3):8–24, 2003.

[138] Z. Sun. *Case-Based Reasoning in Electronic Commerce*. PhD thesis, School of Information Technology, Bond University, 2003.

[139] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[140] telerik. http://www.telerik.com/. 2002-2006.

[141] A. L. Tello and A. Gómez-Pérez. BAREMO: how to choose the appropriate software component using the analytic hierarchy process. In *the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 781–788, Ischia, Italy, 2002.

[142] TopCoder. http://software.topcoder.com/. 2000-2003.

[143] S. Varadarajan, A. Kumar, D. Gupta, and P. Jalote. ComponentXchange: An E-exchange for Software Components. In *IADIS International Conference WWW/Internet 2002 Proceeding*, Lisbon, Portugal, 2002.

[144] Padmal Vitharana, Fatemeh "Mariam" Zahedi, and Hemant Jain. Knowledge-Based Repository Scheme for Storing and Retrieving Business Components: A Theoretical Design and an Empirical Analysis. *IEEE Transactions on Software Engineering*, 29(7):649–664, 2003.

[145] J. Voas. Software Certification Laboratories: To be or not to be liable? *Crosstalk*, 11:21–23, 1998.

[146] Z. Wang, X. Xu, and D. Zhan. A Survey of Business Component Identification Methods and Related Techniques. *International Journal of Information Technology*, 2(4):229–238, 2005.

[147] T. Wanyama and B. H. Far. Agent-Based Commercial Off-The-Shelf Software Components Evaluation Method. In *the 1st International Conference on Agent Based Technologies and Systems*, pages 133–141, University of Calgary, Canada, 2003.

[148] H. Washizaki and Y. Fukazawa. A Retrieval Technique for Software Components Using Directed Replaceability Similarity. In *the 8th International Conference on Object-Oriented Information Systems (OOIS'02)*, volume LNCS 2425, pages 298–310, London, UK, 2002. Springer-Verlag.

[149] Jeannette M. Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems*, 9:1–24, 1987.

[150] D. M. Yellin and R. E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

[151] D.l M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-automatic Construction of Software Adaptors. In *the 9th Annual Conference on Object-oriented Programming Systems, Language, and Applications (OOPSLA '94)*, pages 176–190, New York, NY, USA, 1994. ACM Press.

[152] A. M. Zaremski and J. M. Wing. Signature Matching: a Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology*, 4:146–170, 1995.

[153] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.