4-1-2006

A Framework for Checking Behavioral Compatibility for Component Selection

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

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

A Framework for Checking Behavioral Compatibility for Component Selection

Lei Wang and Padmanabhan Krishnan
School of Information Technology
Bond University
Gold Coast, Queensland 4229
Australia
Email: {lwang,pkrishna}@staff.bond.edu.au

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.

Keywords: interface specification, component selection, tool support, light weight formal methods

1. Introduction

The main advantage of Component-Based Development (CBD) is the reuse of existing components when building systems, so that time and effort can be saved. In our context, a component means any kind of reusable units, such as a class, or a library. In order to combine components, the necessary steps include retrieval of options, selection of component, adaptation of component, composition with existing system and verification and validation of the enlarged system. As the other activities depend upon the retrieval and selection of components, the process of component selection is an important aspect of CBD.

Researchers have addressed this question from different perspectives. Some view component selection as software engineering discipline, starting from requirement analysis, through whole system life cycle. Software processes for users to identify, evaluate, and finally choose components are driven by models which include the product descriptions and evaluation criteria. Other researchers focus only on detailed component storage and retrieval techniques, assuming that requirements have been elicited. In such cases the focus involves component classification and matching. Some classification techniques such as enumerated [13], faceted [9, 28, 29], and hypertext [8] can facilitate user's search of components. However, an agreed vocabulary has to be developed and component users have to be familiar with such a vocabulary [23] when searching components.

Component retrieval based on matching needs an understanding of component specification, which could be formal or informal. Informal specification is easy to understand, but not precise. The components retrieved based on informal specifications often cannot be integrated into targeting systems. This is because informal specifications contain only syntactic characteristics of component interfaces.

Application Programming Interface (API) is a typical example of syntactic approaches to component specification. Some work has been done to extend the API approach by adding invariants and pre/post condition pairs to constrain component's behavior (such as JML [22], iContract [20]). UML [4] (Unified Modeling Language) is a semi-formal modeling language, widely used in design and documentation. Integrated with OCL [26] (Object Constraint Language), UML can also describe component's behavior using other existing 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.

One can use formal descriptions to specify the behavior of component interfaces at an abstract, but sufficiently precise, complete, and unambiguous level. Formal descriptions also enable the checking of consistency and correctness of the interface model. With specifications, component matching is performed upon signature [31] and behavior [32]. Signature matching is also mainly syntactic and is not the focus of our work presented here. Behavior matching can greatly enhance the possibility of a component being integrated and working well in the target system.

While formal methods have been successful for specifying behavioral properties of components, and tools are developed to check these properties, they are still not popular with practitioners. The usual reason is that formal methods require a strong background in mathematics. It is currently unrealistic to expect normal component users and developers to have such knowledge. In order to overcome this difficulty, we think that an intermediate language is necessary. Such a language should have familiar programming language constructs, with easy to understand semantics. Interface Definition Language (IDL) is a good example. However, we developed our own IDL: Simple Component Interface Language (SCIL), for proof of the concept purpose. SCIL is derived from formal descriptions such as Interface Automata. It can also be viewed as a cut down version of a normal programming language which aims to support formal specification of component interfaces. SCIL models are able to be translated to a variety of frameworks and checked by their accompanying tools. Therefore, SCIL and its translator will allow the users to gain access to a number of tools based on formal methods.

In this paper we focus on the design and implementation of the framework that enables the utilization of tools based on formal methods. We deliberately try to hide the formal details – the formal details can be found in papers associated with the tools. We aim to convince the reader, via a main example, that the framework we propose can indeed support formal methods without necessarily exposing the details.

The remainder of this paper is organized as follows: Section 2 briefly introduces background information of our approach; an overview of SCIL is presented in Section 3; the implementation of SCIL compiler is described in Section 4; Section 5 gives an example of checking behavioral compatibility by our approach; some problems are discussed in Section 6, and the paper ends with a conclusion and potential future extensions.

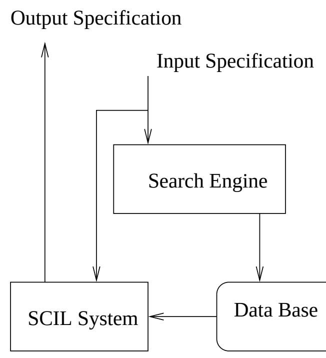

Figure 1. Component selection and Validation.

2. Background

The goal of our research is to increase the possibility of finding and retrieving components that meet user's requirements. We have argued that improving the collaboration between component developers and users improves the ability to find suitable components [21]. Though the collaboration of developers and users is not really new idea, there are very few real implementations that support collaboration.

The assumption of collaboration is that components usually need to be modified to meet user's requirements. It is easier for the component developer(s) to achieve this because they have direct access to their products. Some promising technologies may be used at the developer's side to facilitate this customization, such as generative programming or software product line [16]. Otherwise the customer may have to write significant glue code to use the component.

The prerequisite of collaboration is to find satisfactory component candidates. Figure 1 represents our approach to supporting component selection using compatibility checking. The key steps in the process include:

- Retrieving a component specification from the repository. The component specification is also written in SCIL, submitted by its developer, to describe the component's behavior.
- Combining the user's requirement specification with the component specification after mapping the names in the two files.
- Invoking the SCIL compiler to translate the combined specification to the target language(s). Var-

ious plug-ins within the compiler are used to help the translation.

- Send the translated specification to the relevant model checking tools to check whether the user-specified properties can be satisfied by the component. If so, the component can be accepted. As the SCIL specifications focus only on the key aspects of behaviour, resulting state transitions systems are not too large and can be model checked.

The first step is completed by the search engine while the other steps are implemented as part of the SCIL system as shown in Figure 1. The SCIL system consists of the language (SCIL) and its translator (compiler with plug-ins), as well as those existing formal model checking tools are bound together aiming at leveraging formal methods: SCIL deals with the syntactic issues while the translator etc. deal with the formal side.

Some existing Behavioral Interface Specification Language (BISL) [30], such as JML [22] or CORBA IDL with temporal logic [19], can be used instead of SCIL by developing appropriate translators. However, JML essentially provides annotations to Java like code. JML is a much richer language and verifying joint behaviour of components expressed in Java and JML requires the use of sophisticated theorem provers [27]. Another similar approach is VCF (Vienna Component Framework) [24]. VCF is aimed at integrating components written in a variety of frameworks while our aim is just selection of components via checking potential uses. So interface descriptions are needed in our approach. Cadena [17] relies only on the SPIN model-checker while we want to support the use of a variety of tools. Combining the programming aspects with the specification aspects (in our language SCIL) allows the specifier to focus only on the interface behaviour. It also simplifies the process of building translators. In the next section we describe the features of SCIL and how it can be integrated with other tools. This will demonstrate the language features that are useful to support the collaborative effort and how the benefits of other semantics based tools can be leveraged.

3. Simple Component Interface Language

In the collaboration, SCIL (Simple Component Interface Language) is designed as a means for component users and developers to communicate. This makes it possible for both component users and developers to make agreements on whether components can meet requirements by using appropriate checking tools.

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

- Named code fragments which 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 (called protocol) on how the various services can be combined.
- The ability to compose SCIL descriptions. Currently only synchronization on common names is permitted. In general the various possibilities outlined in our earlier work [21] can be used.
- Scenarios which describe the desired behavior and a specification on what the joint behavior should achieve.

Formally, SCIL describes a transition system along with requirements specified using a mix of temporal logic and another transition system. Though SCIL looks similar to some ADLs [15] (Architecture Description Languages) and IDLs [25], it has its own features:

- SCIL describes only component interfaces, no internal actions, no connectors, and no architectural information.
- It is constraint focused. SCIL mainly tells what a component can do, and what it cannot do at the interface level. It supports the use of temporal logic to express required behavior.
- It offers some flexibility to define key data types.
- It supports the features of some existing languages so that a SCIL specification can be translated to those languages with the aid of specific compiler plug-ins.

We view components as pieces of software that provide a set of services. Each service may accept some inputs and generate some outputs based on certain rules. The registration of all the services and the constraints on their inputs and outputs together form the interface of a component. There are also global (in terms of the component) temporal constraints on the interface. These constraints define the correct order of services invoked. These are usually called interaction constraints or protocols [19] between the component and its users.

A typical SCIL file consists of these definitions (see Figure 2): TYPE, SERVICE, PROTOCOL, SCENARIO and PROPERTY. As the aim of SCIL is to support lightweight formal descriptions for interfaces,

```
  SCIL ::= TYPE  VARIABLE
         SERVICE PROTOCOL
         SCENARIO PROPERTY
TYPE ::= int | bool | enumeration | structure
VARIABLE ::= identifier as TYPE
SERVICE ::= INPUT OUTPUT
              assumption(INPUT, OUTPUT)
                        -> guarantee(OUTPUT)
PROTOCOL ::= unary_temporal_op SERVICE |
             SERVICE binary_temporal_op SERVICE
SCENARIO ::= [step] expression+
PROPERTY ::= property_pattern
```

Figure 2. Abstract Description of SCIL

it does not support complex data types. Basic class definitions can be specified using structure types. While it is possible to build a translator from SCIL to a more sophisticated tool such as Bandera [7], it is harder to hide all the details of Bandera from the user.

For each service, we define input and outputs variables, and rules that govern the behavior of these variables. A service corresponds to a high level method in languages such as Java. That is, in the actual component, a service may be implemented by a collection of methods. Such service descriptions can be obtained from use case diagrams. Input and output variables (which are typed) represent the potential parameters. In this paper we focus on the compatibility of component behavior, i.e., rules and protocol compatibility. Rules for a service are about assumptions made by inputs and output guarantees based on the assumptions. In other words, a service can be performed only when the pre-condition is satisfied and the output is represented as a set of assignment statements.

Temporal operators similar to those in [19] are used to define the order of services performed, such as, which service should be called first (`initially`), which should be called before another (`precedes`), and which can only be done once (`once`), etc.

Scenario-based checking is one of the ways to determine if the selected component meets the user's requirements. Scenarios 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 are allowed to have more than one scenario to check.

Another way to check whether a component is compatible to a user's system is to check various properties of the composed system by the component and its environment. Properties can be formed by assertions or temporal logic expressions. In SCIL, we let users write properties using patterns defined in [12], in which how to map these patterns to temporal logic expressions is also presented. For example, users may write: always (P = false). It will be translated to [](!P) in linear temporal logic (LTL). Then the model checking tools that support LTL can be used to check the property. Meanwhile, users can also use keyword "deadlockfreeness" to check whether the system is deadlock free. This property may only be supported by specific tools, such as [11].

Overall, using SCIL we can describe the services that a component offers and the rules by which the component can function properly. Such descriptions of the services are suitably translated to facilitate checking by various tools. It is possible to define such translations formally (or present a formal semantics for SCIL).

4. Implementation

4.1. Architecture

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. At the bottom, the parser is generated by SableCC [14] 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 (including symbol table and rule table) for each part. On the top of grammar checking, there exist 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 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 take those intermediate tables as inputs and generate the translation for the plug-in language.

Generally speaking, the translation takes these steps: SCIL file describing components and component requirements is fed to the compiler; The plug-ins help translating the SCIL file to existing languages, such as Reactive Module (RM) [3], Alloy [18], Interface Automata (IA) [10], etc.; and the results will be checked by their accompanying checking tools, for example, Mocha [2] for RM, Alloy Analyzer [18], and CHIC [6] using IA.

Such a design has the following advantages:

- Model checking tools can be, in principle, used to check component compatibility. Though 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 utilise 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.
- It becomes possible for developers to ship "SCIL components" before the components are purchased. Component selection can be more time efficient than before when users get abstract descriptions that can be used as trial versions of components for the purposes of testing.

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

4.2. Other Implementation Issues

The protocol definition can be checked against the rules of services to find any inconsistencies. As an example consider the situation where the 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. We can trigger suitable consistency checks in the tools that we use.

It is also possible to check if the composition of two components have any illegal behaviors. To support this, one defines either a monitoring automata (a sequence of transitions) or various assertions (defined as properties in the SCIL description of the component). If a monitoring automaton is used, the component is translated to tools such as Mocha and the monitoring automaton executes in parallel with the component. If assertions are used the component is translated to tools such as Alloy which automatically check for the validity of assertions. The type of checking and hence the tool used for verification is driven by the specification(s) in the various descriptions.

Our prototype translator is built with plug-ins to support Reactive Module (RM) 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; Mocha [2] does a temporal analysis based on state machines, while Alloy Analyzer [18] checks that an assertion holds by trying to find a counterexample.

It is possible to add support to another language as long as the corresponding plug-in is implemented. However before picking up a language, some issues need to be considered, such as: whether the language supports describing transition systems; whether there exist a matching theory for using this language to check component compatibility, such as the specification matching from [32]. When developing plug-ins for SCIL, the following steps are necessary. Firstly, one has to extend the common translation layer class to handle language specific syntax such as key words and the technique to express transition systems. This is then followed by deciding the properties that can be be checked. Since different tools provide different facilities to check their formal models, the best way to use the tools need to be explored. Hence adding a new tool only requires a suitable plug-in that maps SCIL to the input language of the new tool.

In next section, we will give an example to show how SCIL can be used with its translator and some formal model checking tools to select components.

5. An Example

In this section, we draw an example from [19] (originally from [5]) with adaptation to show how SCIL can contribute to selecting right components by checking component behavioural compatibility.

5.1. Outline

A user wants to find an 'auctioneer' component for his auction web site. The user requires the auctioneer component to provide interfaces for login, logout, bid, purchase, and sell services. These services will be consumed by 'bidder' and 'seller' components the user already has.

From the user's environment (in this case, it consists of 'bidder' and 'seller' components), additional requirements have been elicited:

- The action `sell` can only happen once.
- A bidder can bid for an item and the action `bid` can only happen after a bidder `login`s;

- After bidding, a bidder can either win or not win the bid;
- If the bidder wins the auction, the bidder has the obligation to `purchase` the product, which is not available for bidding anymore;
- If the bidder does not win, the bidder cannot `purchase` the product. The product has been bought by another bidder.
- A bidder has to `login` first, and `logout` at the end.

5.2. Specification

We assume that a potential component has been identified (in our case via a simple search engine). The user needs to ensure that this potential component does indeed meet the desired requirements. With the component description, the user can download the SCIL file for this auctioneer component. We describe the key aspects of the SCIL specification below.

The auctioneer component defines two types for describing status of products and bidders.

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

The state transitions for the services and the protocol by which the component interacts with its environment need to be defined. For example, the seller sending the `sell` request to the auctioneer changes the product status to become `available`.

```
sell {
  output: ps
  rule: true -> ps = available;
}
```

When a bidder is bidding for the product, `bid` service requires the bidder's status has already become "`logged_in`", and the product is still `available`. If both are met, `bid` provides two possibilities. The product status may change from `available` to `engaged` then finally `sold` after the product has been purchased. Even though the bidder has not won the product, the product status is still changed to engaged, because it is won by other bidders. The SCIL specification of `bid` is shown below.

```
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;
}
```

The requirement that the action `sell` is allowed to occur only once is specified in the protocol part of the SCIL specification via the temporal operator `once` operator.

```
protocol:
  once sell;
```

Other restrictions on the use of the components (i.e., the protocol) require that `sell` goes before all the other services, and `bid` occurs before `purchase`. This component needs a `logout` to end the business with a bidder.

```
protocol:
  initially sell;
  bid precedes purchase;
  eventually logout;
```

The composition of components and the desired behaviour can also be specified. In general, 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. The composed behaviour (or the environment), which for our example describes the connection between a hypothetical bidder and seller to the auctioneer, is developed below.

```
  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;
  }
}

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

The joint `bidder auctioneer` system describes the bidding process (logging in, bidding, purchasing if available etc.) while the joint `seller auctioneer` system can only execute the selling of the item.

Scenario-based checking is able to determine if the selected component meets the user's requirements. Abstractly this is just another transition system and ideally must be derivable from the original specifications. The following shows the specification of scenarios of two particular behaviors that the joint system (viz., `auctioneer`, `bidder` and `seller`) must demonstrate.

```
  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;
}
```

The first scenario (called the `best_story`) shows the bidding succeeding while the second scenario (called the `another_story`) shows the result of the bid failing.

It is also possible to specify the key properties that the joint system must satisfy. This can be expressed in temporal logic. Given below are examples using the `always`, `between`, `precedes` and `after` combinators and also a property that uses pre-defined key words (in this case to denote the absence of deadlock).

```
  property:
 p1 {
// user won't login if 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);

}
 p2 { deadlockfreeness; }
}
```

This completes the informal description of SCIL. A tool based on the Eclipse IDE has been built to enable the development of SCIL specifications. By integrating the the system with the Eclipse IDE, component selection can become an integral part of software development environment.

5.3. Translation

We now show how SCIL can be translated for various tools to enable the checking of relevant properties. As we have experimented mainly with RM and Alloy we discuss the translation to these two tools. Translations to other tools are similar.

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

```
    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 {}
```

In order to translate the requirement that the `sell` service can occur only once, we introduce a private variable in RM to count the number of its occurrences. In Alloy a counter is not required but we can generate a transition predicate that guarantees that `sell` occurs only once.

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 below.

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

  Alloy:
pred sell (bs, bs': BidderStatus,
           ps, ps': ProductStatus)
bs = logged_out && ps = not_available =>
        ps' = available && bs' = bs
```

As scenarios are also state transitions, their translation is similar to that of services. However, as scenarios are used to check the behaviour of the composed system, they are relevant only for tools such as RM. Temporal operators such as `initially` and `eventually` can be translated to their temporal logic counterpart. But due to space limitations, we do not present the details here.

Now the user can use the auctioneer component in the desired environment (with bidder and seller), and use model checking tools to check its behavioral compatibility to the environment.

The translation to a certain language is driven by the constructs used in a particular SCIL specification. As not all tools support all features, the architecture relies on rules which specify the type of translation necessary.

Checking behavioral compatibility is handled differently by the various formal modeling tools. For instance, the tool Mocha checks whether the composition of auctioneer, bidder and seller components (it is done by using renaming and parallel composition) will reach some undesirable states by performing a temporal analysis. These undesirable situations can be identified by the user's requirements. In this 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 [1] 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.

If the composite system (which is described as a parallel composition of the system and the scenario) can complete the scenario (using a predicate that is set to true on completion of the scenario) the user is sure that the component is acceptable as it meets the specified requirements. The RM specification to check the two scenarios is as follows.

```
  module matching_best_story is
  auctioneer_system || best_story
module matching_another_story is
  auctioneer_system || another_story

predicate sc_best_story is
  (final_best_story = true)
predicate sc_another_story is
  (final_another_story = true)

judgment J_best_story is
  matching_best_story |= sc_best_story
judgment J_another_story is
  matching_another_story |= sc_another_story
```

If some unwanted situations occur, the user has a few choices. The user might use the component as it is and write the necessary glue code or point out the conflict(s) to the component's developer hoping to receive a revised version, or reject this component. In the last two cases, the technique outline here can be repeated on the revised component or the component identified by a new search.

The Alloy Analyzer allows the user to specify state transitions. So the checking of reachability of undesirable states is similar to that of RM. However Alloy does not support monitors, so that scenario can not be translated to Alloy. But both Alloy and RM support checking properties specified using predicates. In our example, the translations of properties in RM and Alloy are:

```
  RM:
predicate pred1 is ~((bs = logged_in)
& (ps = not_available))
judgment Jd1 is auctioneer_system |= pred1

Alloy:
assert as1 {
  no s: State | s.bs = logged_in &&
  s.ps = not_available
}
```

6. Discussion

We have presented a language which can be used to describe the interface behaviour of components. This language has been used to demonstrate the main contribution of our work, i.e., the framework in which one check the compatibility of components using a variety of tools. Though we have only presented our translation to RM and Alloy we can, in principle, construct a variety of similar translators to support component compatibility checking.

While we have discussed the auctioneer example in some detail, we have also experimented with a simple shopping cart component (a classic example in the literature) which limits the number of items that can be placed in the cart. The requirement that all behaviors respect the limit on the number of items is then checked using RM. The results are very similar to the auctioneer example.

There are two main purposes of our framework. The first is to enable precise communication between a potential customer and a vendor. The second is to reduce the complexity using formal methods for normal users. By using SCIL and it associated translators, both purposes are achieved. While languages like JML can be used instead of SCIL, it was easier to build the system for a simpler language as its focus is purely on interface behaviour. This was mainly because we do not separate annotations from behaviour and also do not allow general specifications such as quantifiers.

If components cannot meet user's requirements, according to our collaboration model [21], communication between users and developers are required. This communication is based on SCIL descriptions. Developers can adjust or customize the SCIL description of each component. Such SCIL descriptions are used to search for components and will be sent back and forth between users and developers till user's requirements are fulfilled.

7. Conclusion and Future Work

In this paper, we have illustrated our approach on how to leverage formal methods of checking behavioral compatibility of components by defining an intermediate language that resembles a programming language which can be translated to existing modeling languages. The appropriate components can be selected by subjecting the generated formal models to appropriate checks. We have also give an example which demonstrates the approach. The tool supports a variation of the theory presented earlier [21] that formalised the collaboration model. SCIL can be viewed as a concrete light weight language (when compared to other IDLs) that has been used to demonstrate the feasibility of the model.

We give some indication of the effort required in building the plug-ins as well as using the system to check component compatibility. Building the basic plug-ins take about two weeks of full time work. This assumes that the programmer can use SableCC, and has a good understanding of SCIL and the formal method tools. So the time indicated does not include the effort necessary to use SableCC, SCIL or any of the specific tools. The translators are only prototypes in that aspects such as error recovery are not covered. They work correctly on correct input and terminate gracefully with a suitable error message when an error in the input file is encountered. Translating the auctioneer from SCIL to the various languages takes about half a second and is approximately the same for JMocha and Alloy Analyzer. The checking time varies with JMocha taking less than a second to verify most of the properties while the Alloy Analyzer takes about eight seconds to verify the invariant.

Our future work includes adding support to other tools such as interface automata. We will also build in rules which will aid the automatic selection of the appropriate tools.

References

[1] R. Alur, L. de Alfaro, R. Grosuz, T. Henzinger, M. Kangy, R. Majumdar, F. Mang, C. Kirsch, and B. Wangy. *Mocha Manual.*

[2] R. Alur, L. de Alfaro, R. Grosuz, T. Henzinger, M. Kangy, R. Majumdar, F. Mang, C. Kirsch, and B. Wangy. jMocha: A Model Checking Tool that Exploits Design Structure. In *Proceedings of the 23rd*

IEEE/ACM International Conference on Software Engineering (ICSE 2001), pages 835–836, 2001.

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

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

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

[6] A. Chakrabarti. *CHIC: Checker for Interface Compatibility, Installation notes, User manual and API docume.* Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2003.

[7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A Language Framework for Expressing Checkable Properties of Dynamic Software. In *in Proceedings of the SPIN Software Model Checking Workshop*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[8] M. L. Creech, D. F. Freeze, and M. L. Griss. Using Hypertext in Selecting Reusable Software Components. In *Third Annual ACM Conference on Hypertext*, San Antonio, TX USA, 1991.

[9] 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.

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

[11] 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.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *The 21st International Conference on Software Engineering*, 1999.

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

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

[15] D. Garlan, R. T. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *CASCON '97*, pages 169–183, Toronto, Ontario, 1997.

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

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

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

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

[20] R. Kramer. iContract - The Java(tm) Design by Contract(tm) Tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society.

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

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

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

[24] J. Oberleitner, T. Gschwind, and M. Jazayeri. The Vienna Component Framework Enabling Composition across Component Models. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[25] OMG. CORBA 3.0, 2002.

[26] OMG. UML 2.0 OCL Specification, 2003.

[27] E. Poll, J. van den Berg, and B. Jacobs. Specification of the javacard api in jml. In D. Chan and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS)*, pages 135–154. Kluwer Academic Press, 2000.

[28] R. Prieto-Diaz. Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, 34(5):89–97, 1991.

[29] P. Vitharana, F. M. Zahedi, and H. 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.

[30] J. M. Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems*, 9:1–24, 1987.

[31] 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.

[32] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.