Composing Components in Design Phase using Exogenous Connectors

Kung-Kiu Lau, Ling Ling and Zheng Wang
School of Computer Science, The University of Manchester
Manchester M13 9PL, United Kingdom
{kung-kiu, lling, zw}@cs.man.ac.uk

Abstract

Composition can take place during different stages of component life cycle. We identify two main stages : design phase - components are composed into composite components for reuse; deployment phase - components are compiled and the resulting binaries are assembled into executable systems. Ideally, the design phase composition should maximise component reuse. However, this ideal is not realised in current component-based development because they can not reuse composite components in design phase. In this paper, we propose a novel approach for composing components in design phase using exogenous connectors. In contrast to existing composition approaches, our approach allows composite components built in design phase to be further reusable in both design and deployment phases so as to achieve both component reuse and design flexibility. We demonstrate the feasibility of our approach in an industrial-strength case study - Automatic Train Protection system, and compare them with the closely-related existing composition approaches.

1. Introduction

Component-Based Development (CBD) [14] attempts to construct software systems by assembling pre-existing components, possibly supplied by third parties. In such a development process, the life cycle of components [4] can be divided into different phases [8]. The two main phases are *Design* and *Deployment*. In the design phase, component designers design, construct and deposit components in repositories. In the deployment phase, system developers retrieve appropriate components from repositories and compose them into their target systems.

Component composition, or assembly, plays a fundamental role in CBD, and so it does in both design and deployment phases. In the design phase, components can be composed into composite components, which are components that can be deposited in repositories as such. Building composites from pre-existing components is of course a kind of component reuse. Furthermore, composite components represent sub-systems or sub-parts, and are useful as sub-designs in any systematic approach to design. In the deployment phase, binaries of components (composites) are composed into an executable system in such a way that the system's behaviour is as desired. Composing components is again a kind of component reuse as well as a design activity.

The above is the ideal scenario whereby both component reuse and design flexibility can potentially be maximised. In current composition approaches, this ideal is currently not realised, primarily because composite components cannot be composed again in the design phase. In this paper, we propose an approach to the composition in design phase that allows composite components created in the design phase for further composition in both the design and deployment phases. The key idea of our composition approach is using the exogenous connectors [7]. In [7] we introduced exogenous connectors and presented their use in deployment phase composition only. This paper presents the definition and implementation of design phase exogenous connectors, and shows how they are used in the design phase composition. In section 2, we present the definition of design phase composition and describe how components are composed in this phase using exogenous connectors. Then section 3 presents our implementation, followed by an industrial-strength case study, the Automatic Train Protection system, to demonstrate the feasibility of our approach in section 4. Finally, we compare our approach with the closely-related composition approaches in existing literature and discuss the advantages and potential drawbacks of our approach.

2. Design Phase Composition

In CBD, composition is a central issue, since the nature of components is as building blocks from a repository that are assembled or plugged together into larger blocks or systems. So components are pre-existing reusable software units that can be produced and used by independent third-parties. Components can be composed into composite components which in turn can be composed with (compos-

Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06)
0-7695-2594-6/06 $20.00 © 2006 IEEE

ite) components into even larger composite components (or subsystems), and so on.

Composition can take place during different stages of the *life cycle* of components. An idealised life cycle of software components [8] includes the *design* phase, the *deployment* phase and the *run-time* phase.

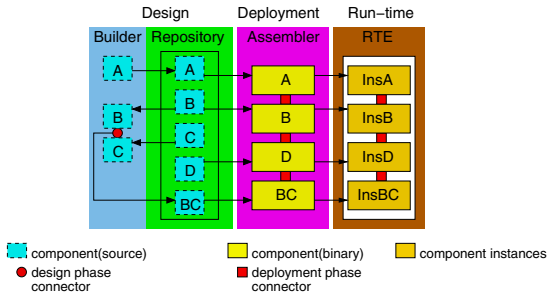


Figure 1. Component Life Cycle.

- *Design Phase*
  In this phase, components are designed, constructed, and then deposited in a repository. Components constructed in this phase are in *source code*, so they are not executable before compilation and deployment. Components can be composed into *composite* components that are also in source code. Composites from subparts of a system and as such are useful for designing a system. The constructed components, including composites, are then catalogued and stored in the repository in such a way that they can be retrieved later, as and when needed.
- *Deployment Phase*
  In this stage, components are retrieved from the repository and compiled into their binary code, so that they are ready for execution. Binary components can be composed into a complete system that is executable.
- *Run-time Phase*
  In this phase, there is no new composition. Components of a system are instantiated with data and the whole system is executed.

2.1. Exogenous Connectors

In our composition approach, components are reusable building blocks that are assembled by exogenous connectors into composite components in design phase. The distinguishing characteristic of our composition approach is on exogenous connectors that provide us a structured way to systematically compose components in design phase.

In the literature of design phase composition, existing approaches usually adopt message passing, and generally fall into two main categories: (i) composition by *direct* message passing; and (ii) composition by *indirect* message passing. In direct message passing scheme, there are generally two distinct roles: the sender and the receiver of a message (Fig. 2(a)). The identity of the receiver is known as a

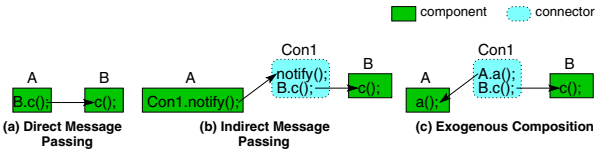


Figure 2. Composition Approaches.

priori by the sender of the message, as exemplified by direct method calls, i.e. the caller Objects have to know the identities of the callee Objects. EJB [5], CCM [12], COM [2], UML2.0 [11] and KobrA [1] adopt direct message passing as a composition approach and their components are usually Objects. In indirect message passing scheme, components are glued together by some scripts that pass messages between them indirectly. A component is connected to another component by a connector that when notified by the former invokes a method in the latter (Fig. 2(b)), as exemplified by ADL connectors [9] that pass messages between components indirectly. Like ADLs, Koala [15], SOFA [13], PECOS [10], PIN [6] and Fractal [3] adopt indirect message passing as the composition approach and their components are usually architectural units.

In direct message passing scheme, Objects are composed by direct method calls. There is no explicit code for connectors that can be reused. Components are tightly coupled with each other and thus are impossible to be reused independently. Moreover, direct message passing scheme does not support composite components to be created in the design phase. Whereas, with indirect message passing scheme, architectural units are composed by connectors that are separate entities and composite components can be created in the design phase. However, they are still difficult to be reused, because those connectors just pass control from one component to the others. They are strongly coupled with those components they connect with.

It is clear that in existing composition approaches by message passing, control is originated from components and mixed up with computation, which results in components tightly coupled to each other and causes difficulties in reuse and composing components in design phase.

In contrast to existing composition approaches, our approach adopts exogenous connectors that initiate method calls in the components, and handle any accompanying data flow, so that any control flow between the components is encapsulated by the connectors, as illustrated by Fig. 2(c).

In our approach, components react to their connector only, rather than directly with each other. Computation is encapsulated by components, whilst control is encapsulated by connectors. Because of well-encapsulation, our composition approach is structured and hierarchical. In the design phase, components are composed by exogenous connectors to composite components that are similar to their constituent components. Thus exogenous connectors provide us an approach to systematically compose components

in a hierarchical way.

This hierarchy of composition requires a properly defined type system for our connectors. In [7], We have introduced the type system for exogenous connectors in deployment phase. In this paper, we focus on composing components by exogenous connectors in design phase and we define the type system of design phase exogenous connectors in Fig. 3.

Basic types **Invocation, ComputationUnit;**

Component types

$$Component \stackrel{def}{=\!=} Atom \mid Composite;$$
$$Atom \stackrel{def}{=\!=} Invocation \times ComputationUnit;$$
$$Composite[n \in Nat, n > 1] \stackrel{def}{=\!=} Connector[n] \times Component \times \cdots \times Component;$$
$$Connector[n \in Nat, n > 1] : Component \times \cdots \times Component \longrightarrow Composite;$$

Figure 3. Type System of Design Phase Exogenous Connectors.

There are two types of components: (i) atomic component (Atom, Fig. 4 (a)); (ii) composite component (Composite, Fig. 4 (b)). Every atomic component has an Invocation and a Computation Unit. The Invocation serves as an interface for the component, while the computation unit

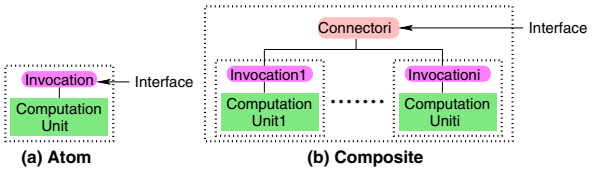


Figure 4. Atomic Component and Composite Component.

provides methods the component could offer. Components in design phase are composed by design phase exogenous connectors to composite components that are similar to their constituents.

2.2. Composing Components in Design Phase

In this paper, we propose a novel approach for composing components in design phase, such that composite components constructed in design phase can be stored and retrieved for further composition. Therefore both component reuse and design flexibility are maximised.

Exogenous connectors in design phase serve as composition operators that connect existing components and encapsulate control over them, so as to build new composite components. These connectors are generic and therefore are pre-built and stored in a repository. The reason for that is connectors can be reused, as well as components in design phase.

Components are constructed by exogenous connectors in design phase in a systematic way. Every atomic component has a Computation Unit together with an Invocation that provides an interface to the component as defined in the type system (Fig. 3). The Computation Unit contains implementation of the methods of the component, while the Invocation provides a way to invoke those methods in the Computation Unit. As depicted in Fig. 4(a), an atomic component is constructed by applying an Invocation to a Computation Unit. The Invocation exposes the component interface and serves as an access point to the component. A component can only be invoked through its interface, i.e. its Invocation.

Atomic components are composed to form composite components by the connectors that compose their interfaces. For each atomic component, the interface is exposed by the Invocation, so a connector connects the Invocations to construct a composite and then exposes an interface for it (Fig. 4(b)). The composite component interface is defined in terms of the interfaces of the constituent components. Like atomic components, a composite component also has a top level connector as an interface, because it is the access point to services provided by the component. Composite components can again be composed by connectors to a larger composite component, as shown in Fig. 5. The connection points of the constituent components are the interfaces, i.e. the top level connectors.

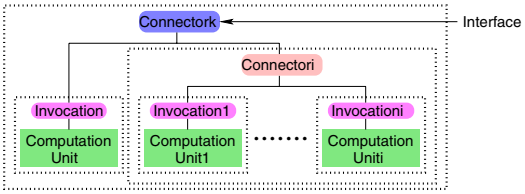


Figure 5. Composing components in Design Phase.

So far we have defined and implemented *Pipe*, *Sequencer*, *Selector* and *Condition* connectors. For a Pipe, when a method on the composite component is called, it invokes methods on the constituent components sequentially, obtains the result from the previous component and pipes it to the next one. For a Sequencer, the methods of components connected are executed sequentially as the order defined by the Sequencer connector. For a Selector, when a method on a composite component is invoked, it can switch to the correct constituent component to accomplish the request. A Condition is just a special selector, which connects only one component and executes the methods of the component based on the satisfaction of the condition. In this case, the control flow over the constituent components is encapsulated entirely by the connectors.

A composite component built in the design phase can be further reused in building new composite components in the design phase. For example in Fig. 6, a composite component *AB* is retrieved from the repository. Components *AB* and *D* are composed by a Selector connector that connects their top level connectors, Pipe and Invocation, so as to build a larger composite component *ABD*. *ABD* can again be deposited back into the repository for future reuse. In the deployment phase, components in the repository are re-

Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06)
0-7695-2594-6/06 $20.00 © 2006 IEEE

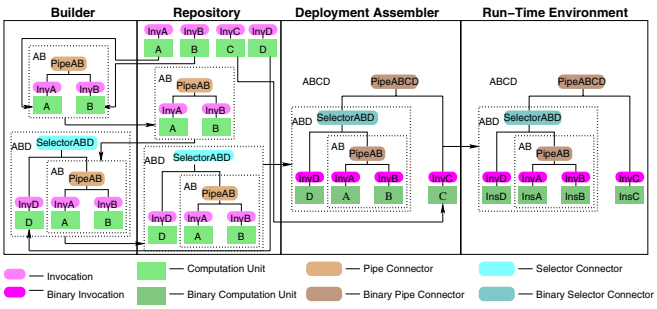


Figure 6. Reuse Composites in both Design and Deployment Phases.

trieved and compiled into binaries. They are composed by deployment phase exogenous connectors (as presented in [7]) to executable systems. For example in Fig. 6, the composite component *ABD* is retrieved from the repository and compiled into its binary code and then composed with the binary component *C* to an executable system *ABCD*. Thus the results of design phase composition (composites) can be reused in both design and deployment phases, and hence design phase composition maximises design flexibility.

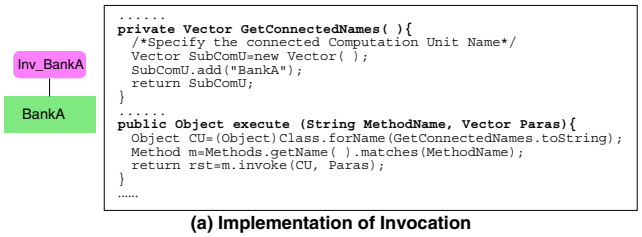
3. Implementation

Having defined the design phase composition, in this section we present the implementation of components and the design phase exogenous connectors in Java, and illustrate how composite components are built in design phase using exogenous connectors.

In our implementation, an atomic component is a set of Java compilation units. It consists of source code of an Invocation and a Computation Unit. For example in construction of an atomic component *BankA*, Invocation *Inv_BankA* is connected to the Computation Unit *BankA* (Fig. 7). In the implementation of Invocation, we specify the connected computation unit name in the *GetConnectedNames()* method of the Invocation source code, as illustrated in Fig. 7(a). As we have explained in 2.1, an Invocation is the interface of an atomic component, i.e. it is the access point for invoking the services provided by the component. In the implementation of Invocation, it has an *execute* method:

```
execute(String MethodName,Vector Paras)
```

It is used to invoke a given method in an atomic component with the parameters. For example, the interface of the atomic component *BankA* is illustrated in Fig. 7(b). From the interface, the user who wants to use the component gets to know the methods provided by the component, e.g. *Withdraw*, *Deposit* and *CheckBalance* and the corresponding parameters, and the way to invoke methods is calling the *execute* method as shown on the interface. When the execute method on the Invocation is invoked, Invocation calls



(a) Implementation of Invocation

```
<Component>
  <Name> BankA </Name>
  <Operation_Specification>
    Object execute (String MethodName, Vector Paras)
      <Method_List>
          <MethodName>Withdraw
            <Para>..</Para> <Return>...</Return>
          </MethodName>
          <MethodName> Deposit ... </MethodName>
          <MethodName> CheckBalance ... </MethodName>
      </MethodList>
  </Operation_Specification>
</Component>
```

(b) Interface for the Atomic Component BankA

Figure 7. Invocation Implementation and Example of an Atomic Component Interface.

the desired method with the parameters in the Computation Unit as shown in Fig. 7(a).

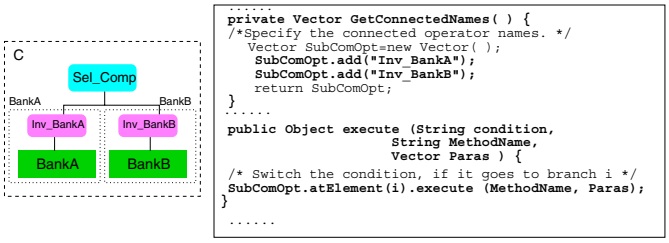
Atomic components can be composed by design phase exogenous connectors to composite components. A composite component is also a set of Java compilation units, which consists of the source code of the constituent components and its associated exogenous connectors. The implementation of the exogenous connectors is generic and therefore is deposited into a repository for reuse. So not only components, but also connectors can be reused in our composition approach. When a connector is retrieved from the repository and has the connecting components specified in its source code, the connector becomes specific for a composition. For example in Fig. 8, *Sel_Comp* is a selector retrieved from the repository and it composes two atomic components *BankA* and *BankB* to construct a composite component *C*, so that a user could use the services provided by *BankA* and *BankB* through *C*. The Invocations are the interfaces of the atomic components *BankA* and *BankB*, so that we specify their Invocation names in the *GetConnectedNames()* method of the *Sel_Comp* source code (Fig. 8(a)). The Selector connector has the *execute* method:

```
execute(String condition,String MethodName,Vector Paras)
```

The composite component constructed by Selector is invoked through this *execute* method. Internally the Selector needs a condition to switch to the correct constituent component, and invokes the *execute* method on the constituent component interface to accomplish the request as illustrated in Selector *execute* method in Fig. 8(a).

The interface of a composite is defined in terms of the interfaces of the constituents. For example, the interface of the composite component *C* (Fig. 8(b)) shows the *execute* method and the method list under each condition provided by *BankA* and *BankB* respectively. So the composite component *C* constructed by *Sel_Comp* can be used to work with both component *BankA* and *BankB* depends on the condition.

Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06)
0-7695-2594-6/06 $20.00 © 2006 IEEE



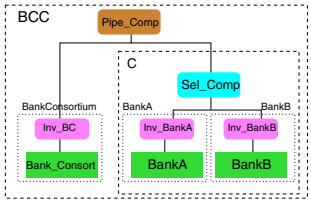

```
.....
private Vector GetConnectedNames( ) {
/*Specify the connected operator names. */
   Vector SubComOpt=new Vector( );
    SubComOpt.add("Inv_BankA");
    SubComOpt.add("Inv_BankB");
    return SubComOpt;
}
.....
public Object execute (String condition,
                       String MethodName,
                       Vector Paras ) {
/* Switch the condition, if it goes to branch i */
SubComOpt.atElement(i).execute (MethodName, Paras);
}
.....
```

(a) Implementation of Selector

```
<Component>
    <Name> Composite Component C </Name>
    <Operation_Specification>
        Object execute(String condition, String MethodName, Vector Paras)
     <Condition Case="Inv_BankA">
         <Method_List>
                <MethodName> Withdraw ...</MethodName>
                <MethodName> Deposit ...</MethodName>
                <MethodName> CheckBalance ...</MethodName>
         </Method_List>
     </Condition>
     <Condition Case="Inv_BankB">
         <Method_List>
                <MethodName> Withdraw ...</MethodName>
                <MethodName> Deposit ...</MethodName>
                <MethodName> CheckBalance ...</MethodName>
         </Method_List>
     </Condition>
    </Operation_Specification>
</Component>
```

(b) Interface of the Composite Component C

Figure 8. Selector Implementation and Example of a Composite Component Interface.




```
.....
private Vector GetConnectedNames( ) {
/*Specify the connected operator names. */
   Vector SubComOpt=new Vector( );
    SubComOpt.add("Inv_BankConsortium");
    SubComOpt.add("Selector_Composite");
    return SubComOpt;
}
.....
public Object execute (String[ ] MethodNames,Vector Paras) {
    SubComOpt.atElement(0).execute(MethodNames[0], Paras);
    rst = SubComOpt.atElement(0).result;
    SubComOpt.atElement(1).execute(MethodNames[1], rst);
    rst = SubComOpt.atElement(1).result;
    .......
    SubComOpt.atElement(i).execute(MethodNames[i], rst);
}
```

Figure 9. Pipe Implementation.

A composite component can be reused to build up a larger composite. As exemplified in Fig. 9, a composite component *BCC* is constructed by the Pipe connector *Pipe_Comp* that connects an atomic component *BankConsortium* and the composite component *C*. When the banking service on *BCC* is called, the Pipe connector first invokes *BankConsortium* to find out the bank ID according to the account number, and then passes the bank ID as the condition and the service request to composite component *C*. The connection point for the composite component is the top level connector, which is *Sel_Comp* for the composite component *C*. The connecting interfaces of the constituents, i.e. the *Inv_BC* and *Sel_Comp* are specified in the *GetConnectedNames()* method in the *Pipe_Comp* source code(Fig. 9).

The Pipe connector has the *execute* method:

```
execute(String[ ] MethodNames,Vector Paras)
```

It invokes the constituent components with the given methods sequentially, and passes the result of the predecessor to the successor. Internally, it calls the *execute* method on each constituent component, gets the results and pipes them to the next constituent component as shown in Fig. 9.

The *execute()* method for the Sequencer connector has the same signature as the *execute()* method of the Pipe connector. However, a Sequencer sequentially invokes the given methods with individual parameters, instead of piping the predecessor's result to the successor. The Condition connector is a special case of Selector. It connects to only one constituent component. When the input condition is satisfied, the Condition connector will invoke the constituent with the given method name and parameters.

4. A Case Study - Automatic Train Protection System

In this section, we use an Automatic Train Protection (ATP) system to demonstrate the feasibility of our design phase composition approach. The ATP system is located onboard of a train to ensure safety. The system consists of five subsystems: the sensors (SNRS), speedometer (SPDO), brakes (BRKS), alarm (ALRM) and reset (RSET). The SNRS contains three sensors and they are attached to the side of the train and detect information on the track-side signals. Each sensor generates a signal in the range of DANGER, CAUTION, PROCEED respectively. The majority of signals from the three sensors is sent to the rest subsystems. On receiving DANGER, both the alarm and the brakes must be enabled, as the train must be stopped. CAUTION indicates that the alarm must be enabled and if the train speed is not decreasing the brakes will be enabled too. PROCEED allows the train to continue. The fourth signal is UNDEFINED which means there is no majority among the signals. But in order to ensure safety, it is handled as DANGER. A RESET signal is generated by the ATP system. In receiving this signal, the whole system is reset, i.e. the train's brakes and alarm are disabled.

Originally in the repository, we have atomic components *SNRS*, *SPDO*, *RSET*, *BRKS* and *ALRM* which provide functionalities for each subsystem respectively, and the connectors for reuse. Considering in the ATP system, the alarm and brakes react to all the signals, we firstly construct a composite component that consists of *ALRM* and *BRKS*. The interface of the atomic component *ALRM* is shown in Fig. 10. The *ALRM* component has methods: *Enable*, *Disable* and *IsEnabled*, which can be called through the *execute* method on the Invocation. To implement the functionalities of the composite component, we retrieve the components *ALRM* and *BRKS* as well as the Condition, Sequencer and Selector connectors from the repository, and they are composed hierarchically to the composite component *ALRM_BRKS_Control*(ABC), as illustrated in Fig. 11.

The interface of the composite component *ABC* is shown in Fig. 12. It can be invoked through the *execute* method on

```
<Component>
 <Name> ALRM </Name>
 <Operation_Specification>
    Object execute (String MethodName, Vector Para)
    <Method_List>
      <MethodName> Enable
          <Para>...</Para>   <Return>...</Return>
      </MethodName>
      <MethodName> Disable ... </MethodName>
      <MethodName>  IsEnabled ... </MethodName>
    </Method_List>
 </Operation_Specification>
</Component>
```

Figure 10. The Interface of the Atomic Component ALRM.

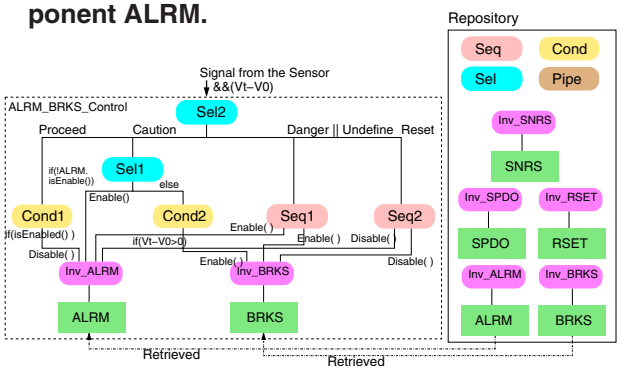


Figure 11. Construction of a Composite Component in Design Phase.

```
<Component>
    <Name> Alarm_Brakes_Control </Name>
    <Operation_Specification>
        Object  execute (String condition, String MethodName,  Vector Paras)
        <Condition Case="Proceed">
          <If Case="Inv_ALRM.IsEnabled()">  Inv_ALRM.Disable ( ) </If>
        </Condition>
        <Condition Case="Caution">
           <Condition Case="! ALRM.IsEnabled( )"> Inv_ALRM.Enable() </Condition>
           <Condition Case="ALRM.IsEnabled( )">
              <If Case="(Vt-V0)>0"> Inv_BRKS.Enable() </If >
           </Condition>
        </Condition>
        <Condition Case="Danger" or "Undefine">
           <Seq> Inv_ALRM.Enable( ) </Seq>
           <Seq> Inv_BRKS.Enable( ) </Seq>
        </Condition>
        <Condition Case="Reset">
            <Seq> Inv_ALRM.Disable( )</Seq>
            <Seq> Inv_BRKS.Disable( ) </Seq>
        </Condition>
    </Operation_Specification>
</Component>
```

Figure 12. The Interface of the Composite Component Alarm_Brakes_Control.

the *Sel2*, which is the top level Selector connector. It receives signals and the train speed as input. When the signal is PROCEED, it invokes sub Condition connector *Cond1* that checks whether *ALRM* is enabled. If so it disables the alarm. When the signal is CAUTION, *Sel2* invokes sub Selector *Sel1* that checks whether *ALRM* is enabled or not. If not it enables the *ALRM*, otherwise it calls sub Condition connector *Cond2* that checks whether the train speed is decreasing or not. If not it enables brakes. When the signal is DANGER or UNDEFINED, *Sel2* calls sub Sequencer connector *Seq1*, which enables the *ALRM* and *BRKS* sequentially. When the signal is RESET, *Seq2* is chosen by *Sel2* and it disables both the *ALRM* and*BRKS*.

As the outcome of the design phase composition, the composite component *ABC* can be deposited back into the repository to facilitate component reuse. As shown in Fig. 13, we retrieve the composite component *ABC* and

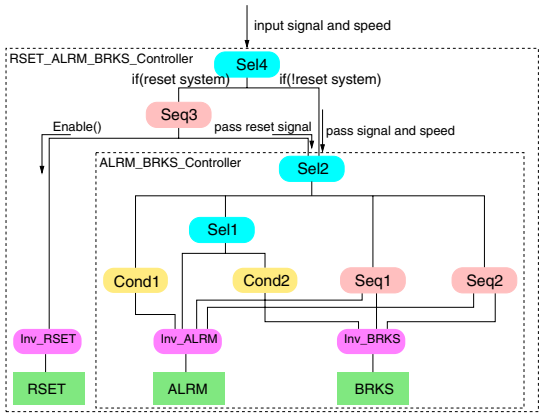


Figure 13. Reuse the Composite Component in Design Phase Composition.

compose it with the atomic component *RSET* to construct a larger composite component, which includes a reset mechanism. A Sequencer connector *Seq3* composes *RSET* and *ABC* by connecting their interfaces, i.e. their top level connectors, which are *Inv_RSET* and the *Sel2* respectively. Then a Selector connector *Sel4* connects *Seq3* and *Sel2* and it becomes the top level connector of the composite component *RSET_ALRM_BRKS_Control*(RABC). *RABC* takes all signals and the train speed as input through the interface *Sel4*. When the signal is RESET, *Sel4* calls *Seq3*, which first enables the component *RSET* and then passes RESET signal to component *ABC*. When the input signal is not RESET, *Sel4* passes the signal and train speed to component *ABC* and the alarm and brakes within composite component *ABC* will respond accordingly.

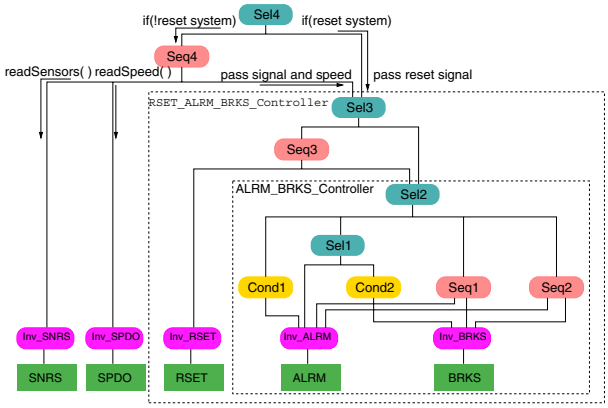


Figure 14. ATP System assembled in Deployment Phase.

The composite component *RABC* can also be deposited back in the repository and be reused for further the design phase and deployment phase composition. In the deployment phase, the whole ATP system is assembled as shown in Fig. 14. Components *RABC*, *SNRS*, *SPDO* are compiled into binaries and composed by a deployment phase Sequencer connector *Seq4*. The connection points for the

atomic components are the Invocations and the one for the composite component is the top level connector, which in *RABC* is the *Sel3*. Further we have another deployment phase Selector connector *Sel4* which connects *Seq4* and the composite component *RABC*. The system is executed through the top level connector *Sel4*. If signal from the system is RESET, *Sel4* will pass forward the signal to the composite component *RABC* to respond. Otherwise *Sel4* invokes the *execute* method in *Seq4*. *Seq4* invokes the *readSensors()* method in *SNRS* and *readSpeed()* in *SPDO*, then passes the signal and speed to the composite component *RABC*.

5. Evaluation and Discussion

The ATP example we presented in the previous section illustrates how exogenous connectors are used to construct composite components in the design phase, which can be reused in further compositions in both design and deployment phases, i.e. composite components can then be composed into larger composite components in design phase or can be composed into executable system in deployment phase. It shows that composite components built in design phase are reusable in further compositions, thus our composition approach maximise design flexibility. In this section we evaluate our composition approach and discuss its advantages and potential drawbacks with respect to closely-related composition approaches.

The distinguishing characteristic of our composition approach is on the exogenous connectors that provide us a structured way to systematically compose components in design phase. The implementation of our connectors is generic and we have demonstrated its use in construction of some other applications such as a banking system. In the current implementation, we implemented exogenous connectors in Java, however, they could be implemented in other programming languages such as C# in .NET. The exogenous connectors that we have implemented are deposited into a repository, and they can be retrieved to compose components into composite components in design phase . Composite components built in the design phase can also be composed to larger composite components, and compiled into binaries that can then be composed to executable systems in deployment phase.

As we have already analysed in section 2.1, existing design phase composition approaches mainly fall into two categories: (i) direct message passing; (ii) indirect message passing. In the following we will compare our composition approach with both of them.

In direct message passing scheme, components are hard-wired together by direct method calls. For a component to communicate with another component, the methods of the callee component have to be specified in the source code of the caller component. For example in EJB, as depicted in Fig. 15, for an enterprise bean to communicate with another enterprise bean, it needs to look up the target bean's home object via JNDI, then calls the *create()* method on the home object to create an instance of the target bean and call the methods on this instance. The connections are specified in the source code of the caller enterprise bean. As

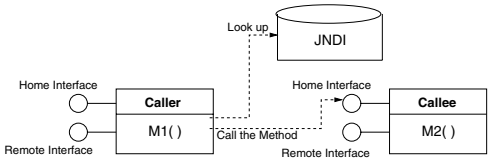
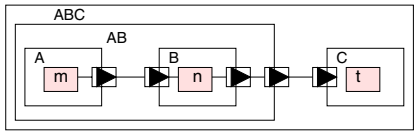


Figure 15. Direct message passing example - EJB.

demonstrated by the EJB example, in direct message passing scheme, there is no explicit code of connector. Thus connectors are not reusable. The caller components are tightly coupled with the callee components. The result of design phase composition in direct message passing scheme is simply not reusable. There is even no such thing as a *composite*, since components are hard-wired by direct method calls, which can not result in a separate entity that could be reused. Thus neither components nor connectors could be reused independently in direct message passing.

In indirect message passing scheme, components are composed by connectors that are separate entities and composite components can be created in the design phase. However, they are still difficult to be reused, because those connectors just pass control from one component to the others. They are strongly coupled with those components that they connect with. We take Koala as an example of indirect mes-



(a) Component Composition in Koala

```
component AB {
  contains  component A a;
            component B b;
  requires  I r;
  connects  a.r = b.p ;
            b.r = r  ;
}
```

(b) CDL file for compound component AB

```
component ABC {
   contains component AB ab;
            component C c;
   connects ab.r = c.p ;
}
```

(c) CDL file for configuration ABC

Figure 16. Indirect message passing example - Koala.

sage passing. Components are composed by connectors that bind up their ports and method calls are routed from one port to the other. For example in Fig. 16(a), the out-port of component *A* is bound to the in-port of component *B* and the out-port of *B* is connected to the out-port of the composite component *AB*. Note that all components including the composites are specified in Component Description Language (CDL) [15] (Fig. 16(b) and (c)). The specification of the composite component *AB* is stored in a file system

that serves as a repository. Later the composite component *AB* is retrieved from the repository and composed with the component *C* by connecting its out-port to *C*'s in-port to a larger composite component *ABC*. In the deployment phase, there is no new composition and all Koala components are compiled into a programming language, e.g. C.

As demonstrated by the Koala example, in indirect message passing scheme, components are difficult to be reused, because they are tightly coupled by connectors. In particular, Koala only reuses components in their specifications rather than their implementations, so no components are reused in the deployment phase. Consequently, the result of design phase composition in indirect message passing scheme are difficult to be reused in general.

In contrast to the existing composition approaches, we adopt exogenous connectors in our design phase composition. Components are composed in the design phase to form composite components, which in turn can be composed again in both design and deployment phases. Hence the reusability of components is maximised. We reuse not only the specifications of components but also their implementations, i.e. components do not need to be implemented from scratch in the deployment phase. Also our approach makes system design more flexible, because developers could make choices on if components are composed in design phase or deployment phase, because they could be reused in both phases.

Our composition approach may have some potential drawbacks. One is that once constructed, the composite component is not flexible to changes, e.g. it is difficult to replace the constituents or extend the composite component. When making some updates or changes, we have to rebuild a new composite component.

Our work is only at the preliminary stage and so far we have not considered the issues of data. Currently, data is kept in the individual constituent components. In some cases, within a composite component, data needs to be shared between the constituents. So data flow is another important issue in design phase composition that we are tackling. We are also in the process of investigating solutions for updating and substitution of composite components. For large-scale software development, this may require much concern when it comes to system maintenance. Thus we need to test our theory on larger examples in future work.

6. Conclusion

In this paper, we present a novel approach for composing components in design phase using exogenous connectors. The ATP example is used to demonstrate the feasibility of our approach. The main contribution of this approach is that composite components built in design phase can be stored in a repository and retrieved for further composition in this phase, and also the binaries of the composite components are further composable in deployment phase. As a consequence, both the reusability of components and the design flexibility are maximised.

References

[1] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based product line engineering with UML*. Addison-Wesley, 2002.

[2] D. Box. *Essential COM*. Addison-Wesley, 1998.

[3] E. Bruneton, T. Coupaye, and M. Leclercq. An Open Component Model and Its Support in Java. In *Proceedings of 7th International Symposium on Component-Based Software Engineering*, pages 7–22. Springer -Verlag, 2004.

[4] B. Christiansson, L. Jakobsson, and I. Crnkovic. CBD process. In I. Crnkovic and M. Larsson, editors, *Building Reliable Component-Based Software Systems*, pages 89–113. Artech House, 2002.

[5] L. DeMichiel, L. Yalçinalp, and S. Krishnan. *Enterprise JavaBeans Specification Version 2.0*. Sun Microsystems, 2001.

[6] J. Ivers, N. Sinha, and K. Wallnau. A Basis for Composition Language CL. Technical Report CMU/SEI-2002-TN-026, CMU SEI, 2002.

[7] K.-K. Lau, P. V. Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proceedings of 8th Internation Symposium on Component Based Software Engineering*, pages 90–106. Springer-Verlag, 2005.

[8] K.-K. Lau and Z. Wang. A taxonomy of software component models. In I. Crnkovic and M. Larsson, editors, *Proceedings of Component-Based Software Engineering Track on 31st Euromicro Conference*, pages 88–95, 2005.

[9] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE transactions On Software Engineering*, 26(1):70–93, 2000.

[10] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. P. Black, P. O. Müller, C. Zeidler, T. Genssler, and R. van den Born. A component model for field devices. In *Component Deployment*, pages 200–209. ACM, 2002.

[11] OMG, `http://www.omg.org/cgi-bin/doc?ptc/2003-08-02`. *UML 2.0 Superstructure Specification*.

[12] OMG, `http://www.omg.org/technology/documents/formal/components.htm`. *CORBA Component Model, V3.0*, 2002.

[13] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *Proceedings of ICCDS98*. IEEE CS Press, 1998.

[14] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[15] R. van Ommering. The Koala Component Model. In I. Crnkovic and M. Larsson, editors, *Building Reliable Component-Based Software Systems*, pages 223–236. Artech House, July 2002.