HAL
archives-ouvertes.fr

Unanticipated Connection of Components based on their State Changes Notifications

Luc Fabresse, Christophe Dony, Marianne Huchard

▶ To cite this version:

Luc Fabresse, Christophe Dony, Marianne Huchard. Unanticipated Connection of Components based on their State Changes Notifications. EECC: Evaluation and Evolution of Component Composition, Jul 2006, San Francisco, United States. 2006. <lirmm-00102873>

HAL Id: lirmm-00102873
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00102873

Submitted on 2 Oct 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Unanticipated connection of components based on their state changes notifications

Luc Fabresse, Christophe Dony and Marianne Huchard
LIRMM – Université Montpellier II
161, rue Ada
F-34392 Montpellier Cedex 5, France
E-mail: {fabresse,dony,huchard}@lirmm.fr

Abstract

Component-based software development is a promising track in software engineering to improve reuse. This paradigm is based on the unanticipated connection of independently developed black-box components. However, any existing proposals enable connections of components based on their state changes notifications without requiring that specific code related to the connection is integrated into components. In this article, we propose a solution to support these kinds of connections. Our solution introduces component properties *and special connectors. We show that properties ease component programming and connectors enforce strict separation between functional code and code dedicated to connection. We develop a prototype in Squeak named* SCL *(Simple Component Language) to give a concrete form to our proposition.*

1 Introduction

Software engineering focuses on component-based models and languages [1, 2, 3, 4, 5, 6, 7] in order to increase reuse as stated by component software development [8]. This new paradigm is based on the following definition: *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"* [9]. Starting from this definition, component-based languages have been built with different or adapted abstractions and mechanisms to provide unanticipated connection between two or more components. To support unanticipated connections, a component definition sets up which services are provided and which services are needed and should make no assumptions about its possible connections or other components to be connected with.

In this paper, we focus on connections based on notifications of component state changes. These kinds of connections are really useful to trigger a component service whenever another component notifies that its state changes. Although these *state connections* are very useful, they are not possible in actual component-based languages in an unanticipated way i.e without writing specific connection code in components. We introduce component properties to avoid this constraint and provide a mechanism based on connectors to support unanticipated *state connections* between components. Properties represent external state of components and connectors represent connections between components. We enhance the actual prototype of our language SCL (Simple Component Language), a simple dynamically typed component-oriented language to support properties and connectors. SCL offers a unified mechanism to build standard and based on properties notifications connections.

The paper is organized as follows. Section 2 discusses why connections based on component state changes notifications are interesting. Section 3 presents the basis of component-oriented programming in SCL. Section 4 presents component properties and how to connect components using properties in SCL. Section 5 discusses related works. Finally, section 6 concludes and presents future work.

2 Motivation

Triggering operations as a consequence of state changes in a component is not a new idea. It is a related idea to daemons or procedural attachments [10] in frame languages, where it was possible to attach procedures to an attribute access which is then executed each time this attribute is accessed, or the Observer design pattern [11]. These kinds of interactions are particularly used between "views" (in the MVC sense [12]) and "models".

In component-based languages, this must be done in an unanticipated way and with strict separation between the component code and the connection code to enable components reuse. However, existing proposals fail to solve these two main constraints. Connecting components based on their state changes notifications always requires that component programmers add special code (like event signaling) in components. For example, Figure 1 shows Archjava [4] code of a CHATCLIENT component that has specific code to enable "state connections".

```
component class ChatClient {
  private String chatText;

  public port Accessing {
    provides String getChatText() {
      return chatText;
    }
    provides void setChatText( String s ) {
      String oldV = Accessing.getChatText();
      chatText = s ;
      Accessing.chatTextHasChanged( oldV, s );
    }
    broadcasts void chatTextHasChanged(
      String oldValue,
      String newValue );
  }
  ...
}
```

Figure 1. An incomplete CHATCLIENT component declaration in Archjava.

At connection time, the broadcast service `chatTextHasChanged` of a CHATCLIENT component may be bound to provided services of other components. For example, it can be bound to a service of a CHATCLIENTGUI component that refreshes the GUI. This connection is only possible because the CHATCLIENT component has a broadcast service which is invoked in its code. This constraint of integrated special code in notifier components is inconsistent with unanticipated connection and strict separation between component code and connection code. It has been illustrated in Archjava but similarly appears in other approaches. In fact, a software architect must be able to construct a software by choosing existing components, adapting them and finally build connections between them without requiring that they have been defined in a convenient way. With our approach, we provide component properties coupled with special connectors to support unanticipated "state connections" of components.

3 Component-oriented programming in SCL

In this section, we present SCL, a simple and dynamically typed component-oriented language, upon which we propose a solution to support unanticipated connection of components based on their state changes notifications. SCL integrates the common features of component-based languages: *component*, *service*, *port*, *property* and *connector* and offers standard mechanisms of *service invocation* and *unanticipated connection* with some variations.

A component has internal state and services. Services represent component behavior like object methods in object-oriented paradigm. Ingoing services are services defined in a component and invoked by other components. Outgoing services are invoked by a component (in its code) but not defined on it and this service may be bound at connection time. Required services are those that must be bound at connection time to an ingoing service of a connected component in order to solve the outgoing service invocation.

Ports are interaction points of components and therefore support connections and service invocations. Figure 2 shows a CHATCLIENT component with two ports.

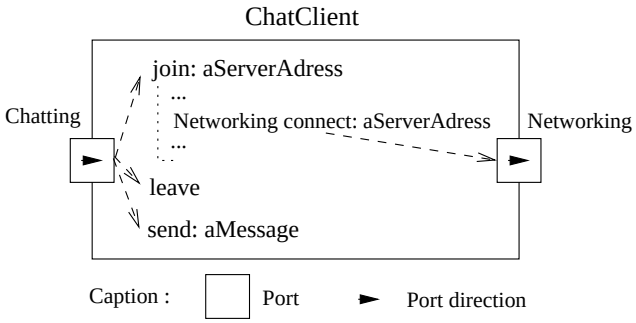

Figure 2. A CHATCLIENT component

`Chatting` is an ingoing port of the CHATCLIENT component providing a set of services that the other components can invoke. In this example, `Chatting` provides the services `join:`, `leave` and `send:`. `Networking` is an outgoing port used in the code of the CHATCLIENT component to invoke services provided by other components. A service invocation is syntactically alike message sending in object-oriented languages. The receiver of a service invocation is a port and the selector is a service name. If the receiver is an ingoing port, the executed service is the matching name service defined on the component whose this port belongs to. If the receiver is an outgoing port, the effectively executed service depends on connections. The outgoing service `connect:` used by CHATCLIENT to make network connection to a chat server has to be bound to a provided service of another component at connection time. Figure 3 shows SCL code of the definition of a CHATCLIENT component.

```
SCLCOMPONENTBUILDER create: #ChatClient
  outPorts: 'Networking'
  inPorts: 'Chatting'.

CHATCLIENT>>init
  (self port: #Chatting)
    addServiceSelector: #join: ;
    addServiceSelector: #leave ;
    addServiceSelector: #send:.

CHATCLIENT>>join: serverAdrs
  (self port: #Networking) connect: serverAdrs.
```

Figure 3. SCL **declaration of** CHATCLIENT

We choose to represent connections between components by *connectors* [13] in SCL in order to provide a good separation of components code and connections code. Connectors connect components through their ports and help solving adaptation problems [14] without using any Adapter pattern [11]. In a connector, outgoing ports are called *source ports* because service invocations come from these ports. *Target ports* are ingoing ports used to invoke services of components. Figure 4 shows the connection of a CHATCLIENT component with a NETWORKMANAGER component that provides services `openConnectionTo:`, `closeConnection` and `sendData:` through its port `Networking`.

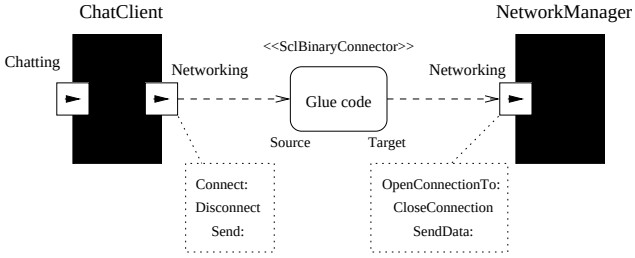

Figure 4. Connection of two components

At connection time, components are like black boxes and ingoing or outgoing services of a component are known by reading the component specification or by introspecting the component. Figure 5 shows the code of the connection of Figure 4 that deals with a frequent adaptation problem which is the non-matching service name problem [15].

In this example, the connection is achieved with the special connector SCLBINARYCONNECTOR which has only one source port and one target port. Glue code is written in the connector to deal with each service invocation and solve adaptation problems: the invocation of the `connect:` service in the CHATCLIENT component code results by executing the `openConnectionTo:` service of the NETWORKMANAGER. This connection mechanism is the basis of component-oriented programming but it can not express

```
chat := CHATCLIENT new.
netManager := NETWORKMANAGER new.

SCLBINARYCONNECTOR new
  source: ( chat port: #Networking )
  target: ( netManager port: #Networking )
  glue: [ :source :target :message |
    ( message selector == #connect: ) ifTrue: [
    ^target openConnectionTo:
      (message arguments first) ]
  ]; connect.
```

Figure 5. SCL **connection code**

unanticipated connections of components based on their external state changes notifications. As in Archjava, a component has to integrate an outgoing port and notifying services to enable these kinds of connections. Our goal is to discard this constraint using component properties.

4 Component properties

To support components connection based on notifications of their state changes, we introduce the concept of *property*. This property concept enhances the idea of property in the Javabeans component model [16] with strict separation between component code and connection code. For example, a CHATCLIENT component has a property named `chatText`. This means that it is possible to get and set a `chatText` value (string messages from chat users) to a CHATCLIENT component. The two first lines of code on Figure 8 show the declaration of the CHATCLIENT component with its `ChatText` and `NickName` properties. The `ChatText` property declaration does not enforce the use of an instance variable named `chatText` to implement this property: other internal implementations can be choosen. When a programmer declares a property, the component is automatically equipped with two ports: an *access port* and a *notifying port*. The property access port is an ingoing port that provides at least getter and setter services. This port avoids the services to respect particular syntactic name convention. The *notifying port* is an outgoing port, which is used to invoke services during property accesses. These services are defined in the SCL component model. For example, the service `nac:value:oldValue:` (nac is an acronym for Notify After Change) is invoked after a property is modified with the new and the old value of the property as parameters. Another service, the `nbc:value:newValue:` (nbc is an acronym for Notify Before Change) service, is invoked before the property is modified with the current value and the next value of the property as parameters. In fact, all defined services have two main characteristics: when they are invoked (before or after the property modification) and what

a connected component is able to do (nothing, prevent the modification or change the property value). An example of connection using properties is depicted on Figure 6 and the corresponding SCL code is shown on Figure 7.

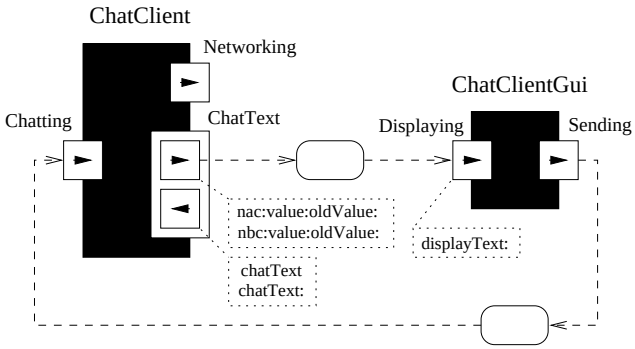

Figure 6. Connect components using properties changes notifications

```
client := CHATCLIENT new.
clientGui := CHATCLIENTGUI new.

SCLBINARYNACCONNECTOR new
  source: ( client notifyPortOf: #ChatText )
  target: ( clientGui port: #Displaying )
  glue: [ :source :gui :message |
    gui displayText: (message arguments second).
  ]; connect.
```

Figure 7. Connecting two components based on a property changes notifications

In this example, each time the `chatText` property of the CHATCLIENT is changed, this results in changing the displayed text on the CHATCLIENTGUI due to the SCLBINARYNACCONNECTOR that only considers `nac:value:oldValue:` invocations on the source port. Special connectors like SCLBINARYNACCONNECTOR ease connection writing. Moreover, a software architect is able to build reusable connectors that can be included in a library of generic connectors. Actually, SCL provides different kinds of connectors like SCLBINARYNACCONNECTOR, SCLBINARYNBCCONNECTOR, PROPERTYBINDERCONNECTOR ensuring that the value of the target property is always synchronized with the value of the source property.

Figure 8 shows the complete code of the CHATCLIENT. Figure 9 shows the complete connection code necessary to build the application. Figure 10 shows a simulation code and Figure 11 shows the screenshot of this simulation[1] execution.

[1]The whole code is available at http://www.lirmm.fr/~fabresse/scl

```
SCLCOMPONENTBUILDER create: #ChatClient
  properties: 'chatText nickName'.
  outPorts: 'Networking' inPorts: 'Chatting'.

CHATCLIENT>>init "presented in Figure 3"

CHATCLIENT>>primitivechatText
  "internal component accessor defined by the
  programmer and used by the generated chatText
  property accessor"
  chatText ifNil: [ chatText := '' ].
  ^chatText

CHATCLIENT>>primitivechatText: newV
  chatText := newV

CHATCLIENT>>primitivenickName
  nickName ifNil: [ nickName := 'anonymous' ].
  ^nickName

CHATCLIENT>>primitivenickName: n
  nickName := n

CHATCLIENT>>leave
  (self port: #Networking) disconnect.

CHATCLIENT>>send: aMessage
  (self port: #Networking)
    send: ('<', self nickName, '> ', aMessage)

CHATCLIENT>>receive: aMessage
  (self accessPortOf: #chatText) chatText:
    (self chatText, String crlf, aMessage).
```

Figure 8. The CHATCLIENT SCL code

Properties are a new feature that helps component programming by providing a higher abstraction to component programmers and software architects. Figure 12 illustrates this fact because a new functionality is added to our chat client with only one "state connection".

This connection allows our chat client application to automatically send the current title played by our music player to other chat users. In other words, each time the `CurrentTitle` property of the component MUSICPLAYER is modified, a message is sent to chat users using the `send:` service of the CHATCLIENT component through its `Chatting` port.

5 Related Work

Our approach is similar to Javabeans component model [16]. A Javabeans programmer declares properties through syntactic name conventions like `get` and `set` and writes the event signaling code to enable connection based on Javabeans properties event signals. The Javabeans model distinguishes different kinds of properties depending on signals like *bound* properties that notify connected Javabeans

```
ChatClientApp>>createChatClientFor: aNickName
  | chatClient chatClientGui |

  chatClient := SclChatClient new.
  chatClientGui := SclChatClientGui new.

  SclBinaryConnector new
    source: (chatClient port: #Networking)
    target: (NetworkManager new port: #Networking)
    glue: [ :chat :netM :message |
      (message selector == #connect:) ifTrue: [
        netM openConnectionTo:
          message arguments first.
      ] ifFalse: [
        (message selector == #disconnect)
        ifTrue: [ netM closeConnection. ]
        ifFalse: [
          netM sendData: message arguments first.
        ]
      ]
    ] ; connect.

  SclBinaryConnector new
    source: (chatClientGui port: #Sending)
    target: (chatClient port: #Chatting)
    glue: [ :gui :chat :message |
        chat send: ( message arguments first )
    ] ; connect.

  SclPropertyBinder new
    bind: (chatClient property: #chatText)
    with: (chatClientGui property: #displayText).

  SclPropertyBinder new
    bind: (chatClient property: #nickName)
    with: (chatClientGui property: #chatUserName).

  (chatClient accessPortOf: #nickName)
    nickName: aNickName.
  (chatClient accessPortOf: #chatText)
    chatText: 'Welcome in ChatClient app v 0.0.1'.
  (chatClientGui port: #Displaying) show.

  ^chatClient
```

Figure 9. Chat client application code

```
| chatClientApp chatServer bCli rCli |

chatServer := ChatServer new startOn: 8080.

chatClientApp := ChatClientApp new.
bCli := chatClientApp createChatClientFor: 'Bob'.
rCli := chatClientApp createChatClientFor: 'Rick'.

(bCli port: #Chatting) join: '127.0.0.1:8080'.
(rCli port: #Chatting) join: '127.0.0.1:8080'.

(rCli port: #Chatting) send: 'Hello guys'.
(bCli port: #Chatting) send: 'Hi Rick'.
(bCli port: #Chatting)
  send: 'Did you have a nice day ?'.
```

Figure 10. Simulation code

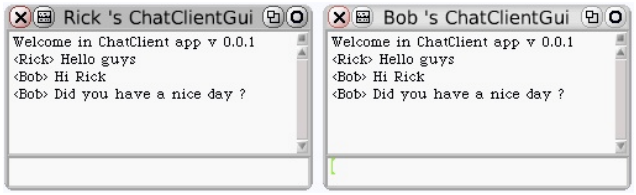

Figure 11. Screenshot of the simulation execution

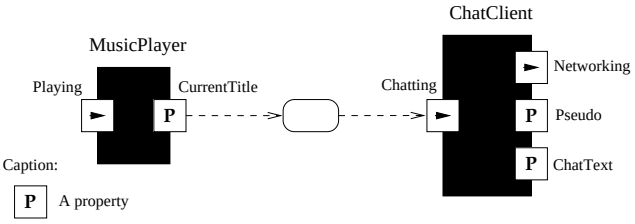

Figure 12. Evolving the chat client application using connection based on properties

after each value changes. A Javabean component programmer has to write a lot of code that is not relevant for the component but for its connection. The automatic and hidden use of Adapter pattern enables to create connections between Javabeans without requiring that notified components integrate specific code like in the Observer pattern (notified component have to define an update method).

In the Corba Component Model [3], the notifier component must integrate an event source that emits an event notifying its value changes and notified components have to offer an event sink that receives compatible events. This is not an unanticipated connection although the Adapter pattern can also be used to avoid specific code in notified components.

In the Fractal model [5], components have interfaces (defining a set of services) that are server (provided) or client (required). Components are connected through primitive bindings or composite bindings. A primitive binding is a fixed interface connection mechanism that binds one client interface with one server interface. Binding components also called connectors represent composite bindings. Like in Archjava with broadcast services, in Fractal, notifying services have to be coded in components and put in client interfaces although they are not required by the component.

6 Conclusion and Future Work

In this article, we show that the unanticipated connection mechanism of components in most component-based languages is not enough to create connections based on component states notifications. We propose a solution to enable this kind of connections in an unanticipated way and with a strict separation between the component code and the connection code. To achieve this goal, we introduce component *properties* based on ones in the Javabeans component model. Properties allow programmers to declare external state of components that will be used by software architects at connection time to create connections among notifications emitted by these properties. The connection code is encapsulated into connectors allowing code separation and extensibility because the software architect is able to build new connectors to extend this connection mechanism. Our proposition is based on our SCL (Simple Component Language) language prototyped in Squeak.

On the one hand, we will extend the property concept to support multi-valued properties i.e. properties whose value is a collection of elements. These properties changes are different such as adding or removing an element and new connectors are needed. On the other hand, connections based on properties notifications have to be used carefully because of the possibility of infinite recursive notification loop. For example, glue code in a connector is executed each time a property notifies a change and this glue code must not change directly or indirectly (through other connections) this property otherwise there is an infinite notification loop during runtime. This problem has to be detected before runtime with program analysis.

References

[1] J. C. Seco and L. Caires, "A basic model of typed components," *Lecture Notes in Computer Science*, vol. 1850, pp. 108–129, 2000. [Online]. Available: citeseer.ist.psu.edu/article/seco00basic.html

[2] R. Monson-Haefel, *Enterprise JavaBeans*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999.

[3] Object Management Group, *Manual of Corba Component Model V3.0*, 2002, *http://www.omg.org/-technology/documents/formal/components.htm*.

[4] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation." in *ICSE*. ACM, 2002, pp. 187–197.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "An open component model and its support in java." in *CBSE*, ser. Lecture Notes in Computer Science, I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, Eds., vol. 3054. Springer, 2004, pp. 7–22.

[6] P. H. Fröhlich, A. Gal, and M. Franz, "Supporting software composition at the programming-language level," *Science of Computer Programming, Special Issue on New Software Composition Concept*, vol. 56, no. 1-2, pp. 41–57, April 2005. [Online]. Available: http://www.cs.jhu.edu/ phf/publications.shtml

[7] R. Marvie, "Picolo: A simple python framework for introducing component principles," in *Euro Python Conference 2005*, Göteborg, Sweden, june 2005.

[8] G. T. Heineman and W. T. Councill, Eds., *Component-based software engineering: putting the pieces together*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[9] C. Szyperski, *Component Software: Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley, 2002.

[10] M. Minsky, "A Framework for Representing Knowledge," in *The Psychology of Computer Vision*, P. Winston, Ed. ny: mgh, 1975, pp. 211–281.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, March 1995.

[12] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view-controller user interface paradigm in smalltalk-80," in *Journal of Object-Oriented Programming*, vol. 1, Août-Septembre 1988, pp. 26–49.

[13] M. Shaw, "Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status," in *ICSE '93: Selected papers from the Workshop on Studies of Software Design*. London, UK: Springer-Verlag, 1996, pp. 17–32.

[14] J. Sametinger, *Software engineering with reusable components*. New York, NY, USA: Springer-Verlag New York, Inc., 1997.

[15] P. H. Fröhlich, "Component-oriented programming languages: Messages vs. methods, modules vs. types," in *Proceedings of the Workshop on Programming Languages and Computer Architecture*. Bad Honnef, Germany: Technical Report 2007, Institute for Computer Science and Applied Mathematics, Christian-Albrechts-University, Kiel, Germany, May 2000. [Online]. Available: citeseer.ist.psu.edu/317429.html

[16] G. Hamilton, "JavaBeans," Sun Microsystems," API Specification, July 1997, version 1.01.