Electronic Notes in Theoretical Computer Science 82 No. 5 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 12 pages

Online reconfiguration of component-based applications in PacoSuite

Pieter Schollaert [a,1], Wim Vanderperren [a,2,3], Davy Suvee [a,4], Viviane Jonckers [a,5]

[a] *System and Software Engineering Lab*
Vrije Universiteit Brussel
Brussels, Belgium

Abstract

In this paper, we present an original approach for enabling online reconfiguration of component-based applications. This research fits into our component composition methodology PacoSuite, that makes use of explicit connectors between components, called composition patterns. Both components and composition patterns are documented by making use of a special kind of MSC. We propose an algorithm to check whether a new component can fulfill the role of an old component in a given composition pattern, without the need to revalidate the entire composition all over again. To enable online reconfiguration, we extend the documentation of a component with a new primitive that specifies when a component reaches a safe state. This approach enables to swap a component at run-time, while maintaining a consistent application.

1 Introduction

Software systems are subject to evolution. When a system is deployed for the first time, its design reflects the requirements that were imposed by its end users on that precise moment in time. If a system however addresses real world activities, it will be subjected to changes in domains, needs and expectations over time. Software maintenance is defined as all activities associated with the modification of a software product to meet new requirements, to correct faults, or to adapt a product to a different environment [3]. Software maintenance is the most expensive part of all software costs, estimated at 80%. This high

[1] Email: pscholla@vub.ac.be
[2] Email: wvdeperre@vub.ac.be
[3] Supported by the FWO
[4] Email: dsuvee@vub.ac.be
[5] Email: vejoncke@vub.ac.be

percentage is mainly due to the fact that maintaining software is usually more difficult than the original development, since unanticipated requirements need to be fit into the old system design. Additional problems arise, when systems are encountered for which the down-time should be minimal or even nil, e.g. web services and critical applications such as power plant controllers. Here, updates that take place at run-time are required. Two important aspects need to be considered when live updates are performed. Firstly, the state of the application needs to be preserved. All data, stored in a database, is considered to be safe, but the global state of the application itself should not be lost either. Secondly, the moment of modification must be carefully chosen, since interrupting the system during a critical operation might have serious consequences.
Maintaining applications which are deployed using component-based software development (CBSD) should be easier, as CBSD develops full-fledged software systems by assembling a set of pre-manufactured independently deployable components. Maintaining an application in CBSD, means replacing one or more of its components or its entire design. A component can be modified to correct faults or to meet new performance/functional requirements. When one or more bugs of a component are fixed or its implementation is improved to fulfill new Quality of Service (QoS) requirements, the impact on the system is minimal, as the external interface of the component is not changed. As a result, the replacement of a component with its improved version does not harm the system. However, if the functional requirements of a component do change, problems may arise, because the possibility exists that the modified component might not fit in the application anymore as its specification changed. In addition, the replacement of components needs to occur at runtime, if the uptime of the application is critical. It is also essential that the application remains consistent at all times.
In this paper, we focus on the online reconfiguration of component-based applications which are deployed using PacoSuite. PacoSuite is our approach to visual component composition. This paper discusses our approach for component replacement and proposes a technique to enable online reconfiguration. The next section introduces our component-based methodology. Section three discusses an algorithm to validate the replacement of components. Section four describes the technique we propose to enable online reconfiguration of component-based applications. The visual component composition tool PacoSuite, is presented in section five. Finally, we state our conclusions.

2 Research context : PacoSuite

We mainly focus our component-based research on lifting the abstraction level for component-based development. We want to enable the plug and play concept of component-based development. Therefore, we propose to document components with usage scenarios that specify their use. A usage scenario is

expressed by making use of a special kind of MSC. The main difference with a regular MSC is that the signals are taken from a limited set of predefined semantic primitives. Each of these signals is mapped on the concrete API that performs them. As a result, the documentation of a component is both abstract and concrete at the same time. Fig. 1 illustrates a usage scenario of a generic TCP/IP network component. One participant of a usage scenario represents the component itself and the other participants represent the environment the component expects. In this case, there's only one environment participant, namely the NetworkUser. This usage scenario specifies that the network component first expects to be created by the NetworkUser environment participant. Afterwards, the Network component either expects data to be send over the network or it submits events to the NetworkUser environment participant, when it receives data, when a connection is established or when it is disconnected.
We introduce explicit composition patterns that are also expressed by making use of MSC's. A composition pattern is an abstract specification of the interaction between a number of roles. The signals between the roles originate from the same limited set of semantic primitives. This allows to compare the signals in a usage scenario of a component with these in a composition pattern. Fig. 2 illustrates a generic game composition pattern. This composition pattern specifies the interaction between three roles: the Network, GameGui and Checker roles. One of the applications of this game composition pattern is a distributed scrabble game. The checker role is then filled by a dictionary component that is used to verify the validity of a word. The GameGUI role is filled by a dedicated Scrabble user interface component. The network role can be filled by the network component of Fig. 1.
The documentation of components and composition patterns allows to automatically check the compatibility of a component with a role. The glue-code that constraints the behavior of the components and that translates syntactical incompatibilities is automatically generated. Both these algorithms are based on finite automata theory. In this paper, we do not go into the details of these algorithms. The interested reader is referred to [8,6,7].

3 Component Replacement in PacoSuite

Replacing a component in a given composition means that compatibility of the composition has to be completely rechecked and glue-code has to be generated anew. Revalidating a composition pattern with filled-in components using our current algorithms can take up quite some time since the algorithms are of exponential nature. A lot of work is duplicated, as the compatibility of the entire composition, except for the new component, is already validated. In this section, we describe our approach to minimize the work to validate the correctness of a component replacement and to generate new glue-code. Fig. 3 illustrates the decision chart for determining whether a replacing component

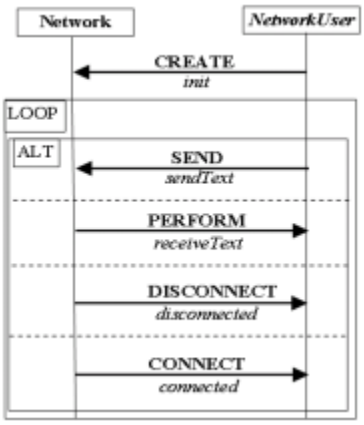

Fig. 1. Usage scenario of Network component.

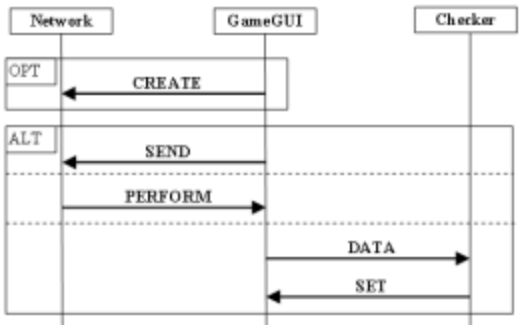

Fig. 2. Generic game composition pattern.

is compatible with an application and whether the glue-code needs changes. We divide component replacement into several logical cases depending on the "difference" between the new and the old component. The cases range from new components that are completely similar as the old components to new components which are not compatible with the composition at all. Before explaining these cases into more detail, we define "difference" between components and how it is calculated.

3.1 Calculating difference between new and old component

The interactions of a component are documented by making use of abstract semantic primitives and the concrete API that performs them. As a consequence, two levels are taken into account for defining difference between two components. First of all, we define difference on level 1 as difference on the primitive level, disregarding the concrete implementation. Both components share the same abstract protocol, if no difference is detected on level 1. If one component replaces another "equal" component, the resulting composition remains valid. Even if two components differ on level 1, the resulting composition does not automatically become invalid. The new component could for instance be completely identical, except that it introduces some new optional fragment of protocol. In that case, the composition is still valid, as the optional functionality of the new component isn't used. Secondly, we define the difference on level 2 as the difference on both the API and primitive level. If no difference is detected on both the API and primitive level, they both share the abstract protocol, but also the same concrete implementation for the abstract protocol. This should for instance be the case for newer versions of a component.
Calculating the difference between two components on both levels described above is very straightforward, as our component usage scenarios can be transformed into deterministic finite automata (DFA). The calculation of the differ-

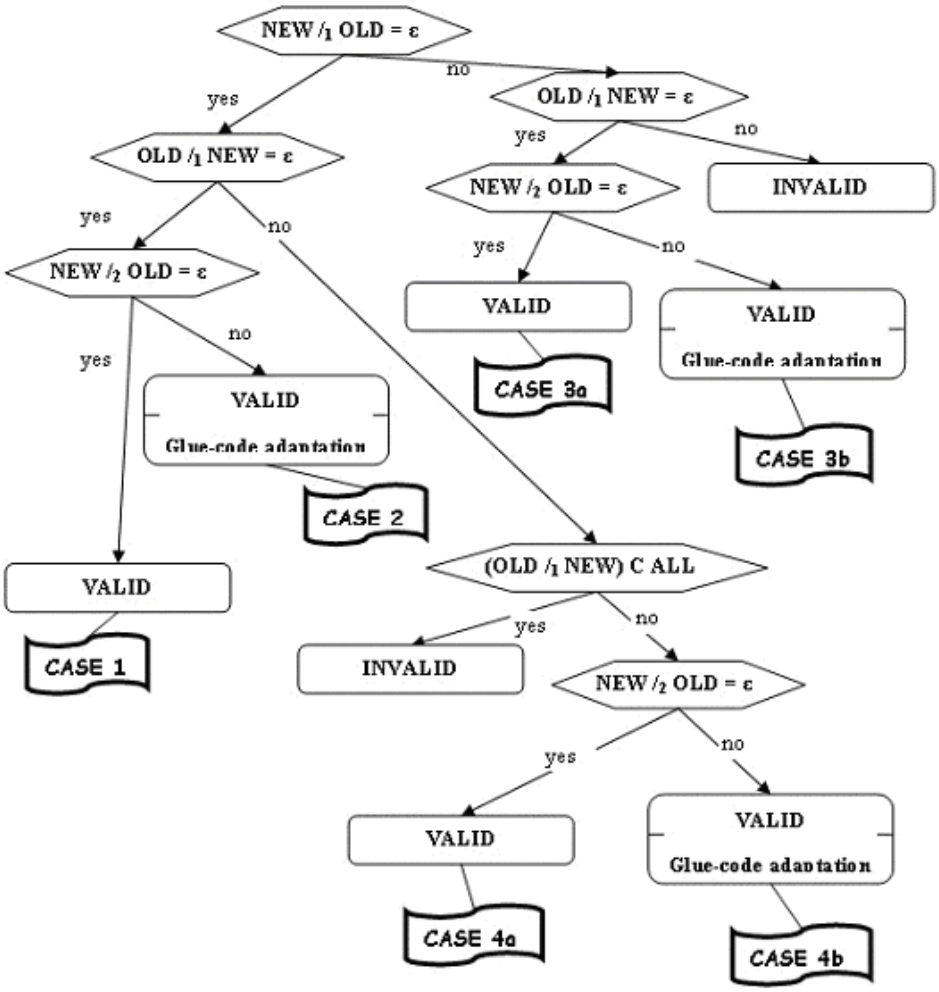

Fig. 3. Decision chart for determining whether the new component is valid or not and whether the glue-code needs changes. Branches that correspond to a logic case are tagged with a case number. We define difference between components on two levels: primitive level only, disregarding concrete API (level 1) and primitive plus API level (level 2).

ence between DFA's is a standard process and described in literature [4]. The only consideration we have to take in mind is that if we check the difference on level 1 only, the concrete implementation (API) needs to be ignored to verify equality of transitions between states.

3.2 CASE 1 : components have the same abstract protocol and implementation

If two components have no difference on the second level (primitive+API), they are completely similar on the protocol level. As a consequence, the new component can safely replace the old one. The existing glue-code doesn't need any changes either. Notice that our approach only checks compatibility on the

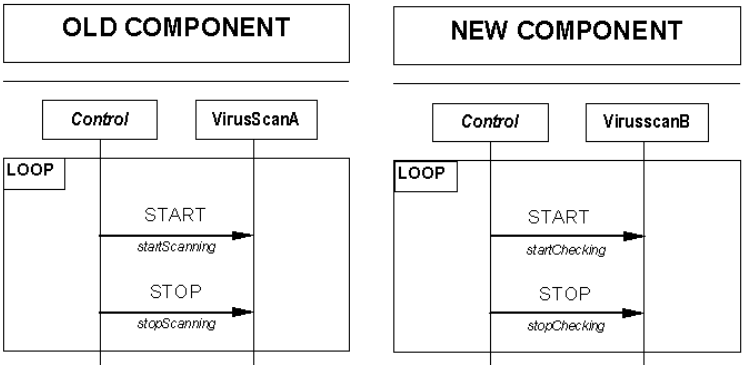

Fig. 4. This example illustrates two kinds of virus scanner components. Replacing the old virus scanner with the new one, is valid. The glue-code however needs to be adapted.

syntactic and synchronization level. The semantic level (i.e. what the component does) is not covered automatically. As a result, it is possible to replace a network component with a UI component if they share the exact same protocol. As a consequence, the application does not work as expected. Checking semantic compatibility is however a hard problem and although some approaches exist, it is difficult to generalize them. In our methodology, semantic compatibility checking is not really needed as we expect the application to be composed manually. The component composer selects the appropriate components based on their description. So, replacing a network component with a UI component would be nonsense for a human person. Our approach however, does allow component composition without in-depth technical knowledge of the components, because of the automatic protocol compatibility checks and glue-code generation.

3.3 CASE 2: components have the same abstract protocol but different implementation

If two components are different on the second level (primitive+API), but not on the first (primitive), both components share the same abstract protocol, but implement it in a different way syntactically. Fig. 4 illustrates a simple example, where a virus scanner component of company A, is replaced by a virus scanner component of company B. They both allow to send START and STOP signals to them. However, virus scanner A implements START by *startScanning* and the virus scanner B by *startChecking*. In this case, both components are compatible on the protocol level, so replacing them is allowed. The glue-code between the components however needs to be altered to cope with the different API of the new component.

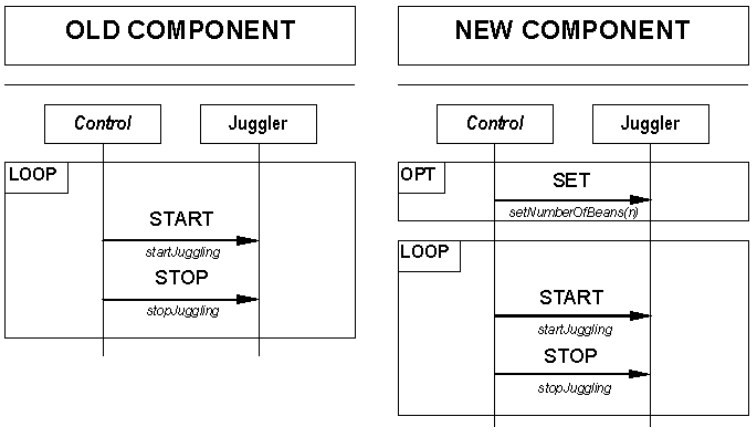

Fig. 5. The component on the right hand-side is a different version of the Juggler bean that allows to change the number of beans it throws. This behavior is not supported by the original Juggler bean. However, the original Juggler bean is checked to be compatible with the application. Therefore, the extra behavior is not used by the application and we can safely replace the new Juggler with the old one.

3.4 CASE 3: New component has extra functionality

In case three, the difference between the new component and the old component is not empty on both level 2 (primitive+API) and level 1 (primitive). The difference between the old component and the new component is however empty on level 1 and either empty or non-empty on level 2. In this specific case, the new component implements some extra functionality. Altough the old component does not have this extra functionality, it is still compatible with the application. The extra functionality is thus not required by the application and as a result the new component can be considered compatible with the given composition. A distinction can be made between two subcases. In case 3a, the difference between old and new component is empty on level 2. As a consequence, the glue-code does not need any changes. In case 3b however, the glue-code has to be adapted to cope with the different API of the new component. Fig. 5 illustrates a simple example of this case. An enhanced version of the Juggler bean replaces the original Juggler bean. The enhanced version supports an extra optional feature to be able to change the number of beans thrown by the Juggler. This behavior is not supported by the original Juggler bean. However, the original Juggler bean is checked to be compatible with the application. Therefore, the extra behavior is not used by the application and we can safely replace the new Juggler with the old one. As the new Juggler bean has the same implementation as the original bean for the behavior they share, the glue-code does not need any changes.

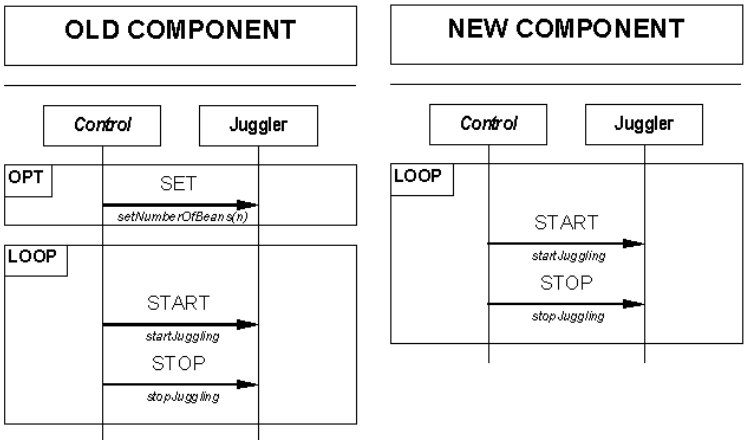

Fig. 6. The component on the left hand-side is a different version of the Juggler bean that allows to change the number of beans it throws. This behavior is not supported by the original Juggler bean on the right-hand side. Replacing the original Juggler bean for the enhanced Juggler is only allowed if the application doesn't use the extra behavior.

3.5 CASE 4: New component misses functionality, but not used by application

In case four, the difference between the new component and the old component is empty on level 1 (primitive) and either empty or non-empty on level 2 (primitive+API). The difference between the old component and the new component is however not empty on both level 2 and level 1. In this specific case, the new component misses some functionality. At first sight, one might decide to define the new component to be incompatible with the application because it lacks some behavior. However, it is very well possible that this behavior is not used by the application. To check this, we have to verify whether the difference between the old component and the new component is contained in the DFA that represents the entire composition. If so, the missing behavior is indeed used by the application. As a consequence, the new component is not compatible with the application. Else, the missing behavior is not used by the application and thus the old component can be safely replaced by the new one. Again, a distinction can be made between 2 subcases. In case 4a, the difference between new and old component is empty on level 2. As a result, the glue-code does not need any changes. In case 4b, the glue-code has to be adapted to cope with the different API of the new component. In the example of Fig. 6, the component composer wants to replace the enhanced version of the Juggler bean with the original bean. However, changing the number of beans it throws is not supported by the original bean. As a consequence, replacing the original Juggler bean for the enhanced Juggler bean is only allowed if the application doesn't use the extra behavior.

3.6 CASE 5: New Component is incompatible with the application

In all other cases the new component is not compatible with the existing application and the replacement should be denied.

4 Online Component Replacement

One of the major challenges of online component replacement is deciding **when** to replace, as replacing a component at a random moment might cause the application to become inconsistent. Therefore, we need to make sure that it is "safe" to replace a component. Kramer and Magee [1] propose an approach where the components have an explicit interface to turn them into passive mode. In passive mode, the components don't engage any interactions with the other components anymore. In [1] a complete theory is developed to decide which components to passivate in order to replace a certain component. A drawback of this approach is that it requires components to have an explicit interface to be able to passivate/activate them. As a consequence, components that do not implement this interface can not be used. In addition, the logic to activate and passivate a given component is not always straightforward to implement. Furthermore, it makes components more heavy-weight as they contain a concern that is not part of their main functionality. Wermelinger et al [5] introduce another approach to be able to determine the safe moment for replacement. They model not only the protocol, but also the interior logic of the components in a formal language. By explicitly modeling the components in a formal language, it is possible to automatically determine when the system will be in a safe state. On the other hand, formal languages are not so user-friendly as graphical representations. In addition, duplicating the component both in the formal language and in the concrete implementation language imposes extra work and it is difficult to keep both the implementation and the formal model consistent.
For the reasons above, we propose a novel strategy to be able to determine the suitable moment for component replacement. Because we already have an explicit documentation of a component, it is very natural to augment this documentation with information that specifies the moment a component reaches a safe state. Fig. 7 illustrates a usage scenario of the Juggler bean that is augmented with safe state information. In this case, the Juggler declares itself safe to be replaced after a consecutive START and STOP have been received. From this augmented documentation, glue-code can be generated that knows when a component is in a safe state. Opposite to [1], we do not have to take the other components into account, because of the glue-code that sits in between the components. Components never communicate directly to each other, so the glue-code is able to remember the interactions that occured during replacement. When a system is running and component X is replaced by component Y, the four steps of algorithm 1 are performed.

Algorithm 1 *Step 1: Verify whether component Y is compatible with the application using the algorithm described in the previous section, if not stop.*
Step 2: Wait until component X is in a safe state.
Step 3: Replace component X by component Y.
Step 4: Continue.

One of the main problems with this approach in general is that it is possible that a component never reaches a safe state or that it takes too long for reaching a safe state. It is however possible to determine if a component is not able to reach a safe state. This way, the component composer is at least warned that the component never reach a safe state.
At first sight, the replacement of a component itself might seem obvious, however, some problems exist. First of all, the glue-code maintains hard links to the components it instantiated, so replacing a component with another component means that we have to adapt the running glue-code to use another component. Second, if the glue-code logic - which is actually a DFA - changes, we have to be able to somehow replace the running DFA with the new one. Our first naive attempt to solve these issues consists of generating new glue-code and stopping the old glue-code. The PacoSuite application and the previous glue-code communicate using a simple network protocol to negotiate a suited moment for replacement. When the replacement is able to occur, all the components except the one that should be replaced are serialized and sent over the network along with the current state of the DFA. PacoSuite then generates new glue-code that uses the new component and initializes the other components with the serialized state. Afterwards, the new glue-code is started. This approach is however quite cumbersome and serializing components might cause problems. Therefore, we propose a more flexible solution that requires changing the glue-code generation process. Instead of generating static glue-code, we plan to generate more dynamic glue-code that is able to change the components it uses and employs reflection to call the correct methods on the components. The new glue-code also allows reading in a new DFA from file or network if necessary. PacoSuite and the glue-code still communicate using a network protocol to be able to determine a safe state for replacement. Afterwards, the new component and possibly the new DFA are sent to the dynamic glue-code. This approach allows switching components safely without the side-effects that might be caused by serializing components. Also, it does not require generating new glue-code, so the replacement of a component itself should be faster. A drawback of this new idea is that using reflection poses an extra performance overhead at run-time during the whole lifecycle of the application.

5 Tool support

PacoSuite [2] is a visual component composition environment, which provides tool support for our component-based methodology introduced in section two.

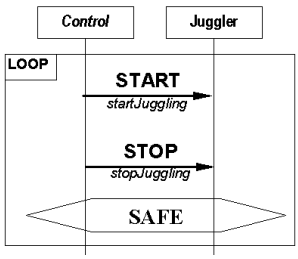

Fig. 7. Juggler component usage scenario augmented by safe states.

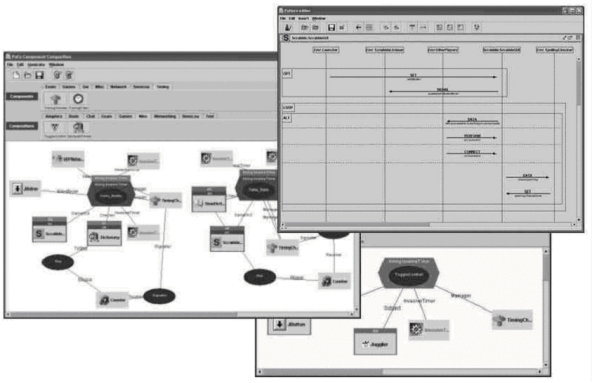

Fig. 8. Screenshots of PacoSuite. The middle left and bottom right screenshots illustrate the visual component composition environment PacoSuite. The rectangles represent components, the ovals stand for composition patterns. The top-right screenshot shows the documentation of a Scrabble component in the PacoDoc tool.

PacoSuite consists of two visual applications, PacoDoc and PacoWire. The fist one, PacoDoc, is a visual editor, used for documenting individual components and composition patterns. PacoWire on the other hand, is a visual composition environment, which allows assembling an application by visually applying components onto composition patterns. This drag-and-drop action is refused when the component is detected to be incompatible with the composition pattern. To perform this check, PacoWire makes use of the documentation of both the components and the composition patterns. When all roles of the composition patterns are filled with their appropriate components, the composition is checked as a whole. Afterwards, glue-code between de various components is generated and a running application is obtained.
We developed a prototype which extends our current implementation of PacoSuite to enable online reconfiguration of component-based applications. The component composer is able to visually replace a component using the PacoWire tool. PacoSuite checks whether this new component is able to perform the same task as the previous one. If so, PacoSuite waits for a save state to occur, before replacing the existing component with its replacement. The component replacement strategy that is currently implemented consists of the first naive idea described in the previous section. Fig. 8 illustrates some screenshots of this tool suite.

6 Conclusions

In this paper, we propose an original approach to enable online reconfiguration of applications, which are build using our component composition tool PacoSuite. Our approach should simplify the maintenance of component-based applications that are subject to evolution over time. We propose an algorithm, which enables to check whether a new component is able to fulfill the role of an old component, without having to revalidate the entire composition all over again. To enable the maintenance of component-based applications at run-time, we extend the documentation of each component with a new primitive that specifies when a component reaches a safe state. This strategy enables online reconfiguration, without making the state of an application inconsistent. The main advantages over other approaches are that we avoid to duplicate the application in a full formal model and that we do not require our components to implement an explicit activate/passivate interface. On the other hand, some drawbacks exist. First of all, it takes an undetermined period of time before the swap takes place because we have to wait until the component reaches a safe state. Second, at this moment we do not provide a solution for automatically transferring the interior state of the old component to the new one.

References

[1] Magee, J., and J. Kramer, *The Evolving Philosophers Problem: Dynamic Change Management.*, IEEE TSE 16, **11**, November 1990.

[2] Pacosuite website, URL: `http://ssel.vub.ac.be/pacosuite`.

[3] Schach, S. R., "Object-Oriented and Classical Software Engineering.", Mc Graw-Hill, ISBN 0072395591.

[4] Ullman, J. D., J.E. Hopcroft and R. Motwani, "Introduction to Automata Theory, Languages and Computation.", Addison-Wesley, Second ed. 2001.

[5] Wermelinger, M., and J.L. Fiadeiro, *A Graph Transformation Approach to Software Architecture Reconfiguration.*, Science of Computer Programming **44**(2):133-155, August 2002.

[6] Wydaeghe, B., "PACOSUITE: Component Composition Based on Composition Patterns and Usage Scenarios.",PhD Thesis, URL: `http://ssel.vub.ac.be/Members/BartWydaeghe/research/PhD/phd.htm`.

[7] Wydaeghe, B., and W. Vandeperren, *Visual Component Composition Using Composition Patterns.*, Proceedings of Tools 2001, July 2001.

[8] Wydaeghe, B., and W. Vandeperren, *Towards a New Component Composition Process.*, Proceedings of ECBS 2001, April 2001.