

Paths and Matrices of Propagation

A concept to trace the impact of modifications on software components ¹

Caroline Berthomieu, Ralf-Detlef Kutsche, Stefan Mann

caroline.berthomieu@isst.fhg.de, rkutsche@cs.tu-berlin.de, stefan.mann@isst.fhg.de

64/02
December 2002

¹ This work has been supported by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung BMBF) as part of the research project *Continuous Engineering for Evolutionary I&C Infrastructures* (Kontinuierliches Engineering für evolutionäre IuK-Infrastrukturen KONTENG) under grant 01IS901C

Abstract

Because of growing market requirements, software systems have become complexer than ever. Recently, component-based software engineering has been presented as a solution to face to this problem. A well defined component model makes component based software systems robust and long-lasting. Nevertheless, due to their complexity, those systems are still difficult to maintain or evolve. Indeed, the impact of a modification grows proportionally with the complexity of the system.

In this paper, paths of propagation and the concept of matrix of propagation will be introduced as a technique to trace the impact of required modifications on the system. More precisely, it guides the engineer by tracing the impact of a given modification of a component on other components of the system.

Contents

1	Introduction	4
2	Component concepts	5
2.1	Structure of a component	5
2.2	Views on component	6
2.3	Component dependencies	8
2.4	Compositionality of components	11
3	The concept of »matrix of propagation«	12
3.1	Notations and semantics	12
3.2	The matrix of propagation as assembly of several concepts	14
3.3	Propagation paths	16
3.4	Example	17
3.4.1	Addition of a service	18
3.4.2	Deletion of a service	19
3.4.3	Modification of a service	22
4	Conclusion	27
5	Bibliography	28

List of Figures

1	Structure of a component	5
2	Blackbox, greybox and whitebox views	7
3	Detailed whitebox view	7
4	Detailed greybox view	8
5	»Use-relation« between two components	9
6	Direct and indirect dependencies in a component	9
7	Compositionality of components	11
8	How the matrix can be read	13
9	The matrix of propagation	15
10	Subsystem S	17
11	Subsystem S in a Greybox view	18

1 Introduction

Software systems have become more complex than ever. Today engineers face the biggest challenge to bring out dependable software systems with new technologies and features, but within a short time. These systems should also be easily modifiable without to become any harm. One promising solution is component based software engineering(CBSE). CBSE advocates the production of software systems by using standardized, prefabricated, stable components. Using CBSE brings down the overall time and cost without having to compromise on quality.

It is well known that software systems often have a longer life than expected. Since the market requirements change continuously, software systems need to be modified appropriately to keep in pace with this development. This is also valid in the case of systems built out of components. A minor change or replacement of a component could produce undesirable impacts on the system. It is therefore inevitable to closely study the effects of the modification of any component on the whole system.

This paper discusses the problems faced during the lifecycle of a component based software system. Here the concept of the matrix of propagation is presented as a solution to diagonalize the problems faced while modifying components of a system.

2 Component concepts

A component model defines a set of standards for the structure of components and the interaction between them. In component model considered in the project »«, components will also be considered to be »composable« [2, 3].

In this section, we will first discuss about the structure of components, then three different views of components will be presented. In a third part, the notion of relation between components or between parts of components will be developed. Finally, the compositionality of components will be discussed.

2.1 Structure of a component

Components are coherent software units which can be put together in order to constitute a complete »composable« software system. They encapsulate a specific functionality of the system. In this way, they have well defined interfaces which describe their functionality, i.e. the services they provide. A service can be defined as a type offered by a specific component. Each component implements a type and manages a set of instances of this type [12]. The interfaces allow to hide the implementation details of the component. Due to this implementation hiding, it is easier to modify or to replace a component.

In the CSE component model, a component consists of 3 parts, namely an export interface, an import interface and a body.

Export
Body
Import

Figure 1

Structure of a component

The export interface

The export interface holds the functionality of the component. It contents the services that the component provides to its environment, i.e. the services that it offers to

other components. Those services are implemented in the body of the component, or in another component. In the second case, the service is imported from the other component via the import Interface.

The import interface

The import interface states the requirements of the component on other components. It specifies services that are needed by the body in order to implement properly the services provided by the component (i.e the exported services). In this way the import interface shows the dependency of the component on other components in its environment.

Note that every service of the import interface is considered to be used in the body.

The body

The body of a component contains the implementation of the services that are provided in the export interface. In this way, the implementation details of the component are transparent to the user.

2.2 Views on component

Generally speaking there exist 3 views on components, namely the blackbox, the whitebox and the greybox view. These views are based on the visibility of the implementation details of a component to a user.

The blackbox view

In this view, the user does not get any information about the body of the component. He only sees which services the component offers and requires, i.e. only the export and the import interfaces are visible.

The whitebox view

In contrast to the blackbox view, the implementation details of the component appears to the user. Here, the export and import interfaces, as well as the body are completely visible (see figure 3).



Figure 2 Blackbox, greybox and whitebox views

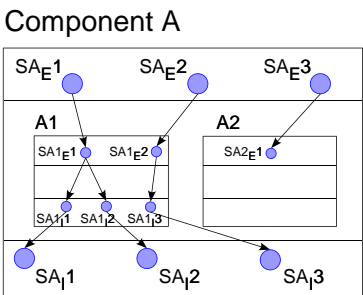


Figure 3 Detailed whitebox view

The greybox view

This view provides more informations about the component than the blackbox view, but it hides some details of the implementation of the component which are shown in the whitebox view. Indeed, the user can see the relations between the services of the export and the import interface of a component independently from the content of the body (see figure 4).

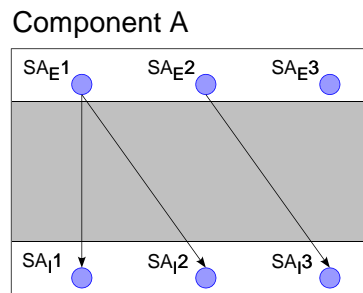


Figure 4

Detailed greybox view

2.3 Component dependencies

In this section, two kinds of dependencies will be developed, namely inter-dependencies between components, and intra-dependencies between component parts.

Inter-dependencies between components

Components are related to each other with »use-relations«. These relations are uni-directional and show the dependency of a component on another (see figure 5).

Most of the time, such a relation hides the utilization of connectors which have the task to »glue« components together.

More precisely, a connector binds the import interface of a component (i.e. its requirements) with the export interface of a component which provides the required services. The connector establishes the communication between components, or coordinates them. In this paper, connectors are considered to be components, therefore they will not be dealt in detail. For more details, see [2, 3, 5].

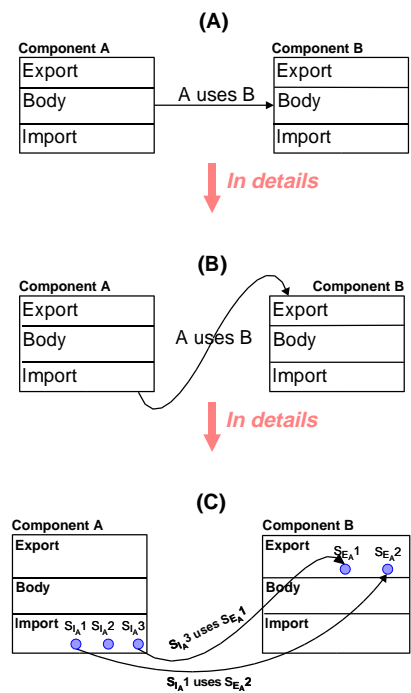


Figure 5 »Use-relation« between two components

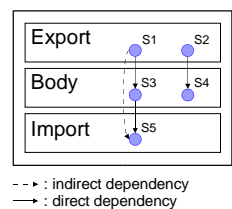


Figure 6 Direct and indirect dependencies in a component

Intra-dependencies between the parts of a component

There are two kinds of dependencies between the parts of a component (see figure 6).

Direct dependencies

The direct dependencies bind the services of the export or import interface with the body of a component. They represent the dependencies shown in the whitebox view of a component (see figures 6) and 3. There are two kinds of direct dependencies, namely:

- 1 *Direct dependencies between the export interface and the body:*
All the services provided by the component are defined in the export interface. Each of them is directly dependent on its implementation, which is made in the body of the component.
- 2 *Direct dependencies between the body and the import interface of a component:*
Some components have requirements on other components. These requirements are services which are imported through the import interface of the component. The body of the component needs the imported services to properly achieve the implementation of its functionality (i.e. the services in its export interface). So, the body is directly dependent on the services of the import interface.

Indirect dependencies

The indirect dependencies are used in the greybox view of components. In order to get a quick overview of a component, it is sometimes relevant to see the relations between the services of the export and the import interface, without considering the whole implementation made in the body. In this way, indirect dependencies can be seen as an abstraction of several direct dependencies which bind a service in the export interface to a service in the import interface (see figures 4 and 6).

2.4 Compositionality of components

The composition of components is the art of combining components with each other in order to create a bigger component (see figure 7). There are two categories of components in component systems:

- The *composite* components: they contain other composite components (subsystems) as well as non-compositional component. They can also be dependent on other components. In this way, a subsystem can be also seen as a composite component.
- the *non-compositional* components: their body contain the detail of their implementation (e.g. code), but no other component.

Note that in this paper, we will always consider non-compositional components in a greybox or blackbox view in order to hide the details of their body.

The compositionality of components is a good technique to achieve certain levels of abstraction of the system. The designer has the possibility to hide details of his model by considering a black box or a more precise greybox view (see chapter 2.2) of some of the composite components of the model.

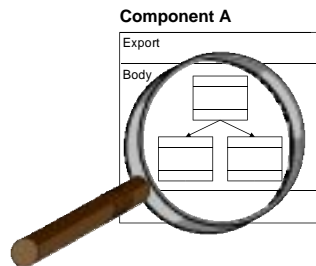


Figure 7

Compositionality of components

3 The concept of »matrix of propagation«

At a certain stage of a software lifecycle, designers or developers need to modify some components. Before the modification is actually done, they need to check whether the component they want to modify is dependent on other components or vice versa. This verification would avoid dire consequences on the whole component system. The aim of the matrix of propagation is to guide the tracing of dependencies between components, and to give to those persons a better view of the impacts of the modification of a component on the other components of the system. Thus the matrix helps the process of evolution of the whole software.

Note that in this chapter, the component-based systems which are taken into consideration (before modification) are assumed to be consistent and reduced:

- Their design must be free of ambiguities or errors (»consistent«),
- Dead code ¹ must be avoided (»reduced«).

3.1 Notations and semantics

The modal logic is the study of the deductive behavior of the expressions »it is necessary that« and »it is possible that« [9].

During their lifecycle, component-based software systems have to be frequently modified in order to respond to a more and more challenging demand. The matrix of propagation has been created to trace the impact of such modifications on the system. More precisely, it shows if a given modification on a component has necessarily, probably, or even no impact on other components. Therefore, the symbols »necessity« and »possibility« of the modal logic have been also used to specify the matrix of propagation (see figure 8).

¹ Dead code can be found in the import interface, as well as in the body of a component. (e.g. in the import interface: if any imported service is not used by the body, in the body: if there is any element or piece of code in the body which does not play any role by the specification of any service of the export interface.

□ : At least one element must be changed. (In the modal logic, this symbol means »it is necessary that...«)

◇ : An element (or more) could need a modification. (In the modal logic, this symbol means »it is possible that...«).

Note: Here, only a person who has a certain comprehension of the system is able to decide of the propagation of the considered modification

— : not affected

Note : The possible modifications mentioned just before are also associated with the notion of dead code creation. Here, three cases are possible. Some Modifications will necessarily bring out dead code, whereas it is by other ones only a possibility. In the last case, the modifications don't bring out any dead code.

A prop B: It is the field of the matrix where line A and column B meet together. Figure 8 shows how the matrix can be read. An action² will be done on a part³ of a component. We can see in the matrix which implied effect has this action on an affected component part, see the following formula:

◇_{Action Part1 prop Part2}:
will_be_modified(Part1) ⇒ ◇ is_affected(Part2)

K: in the Matrix of propagation, K is the component which has to be modified.

Initial Change on a Component	Affected Component Part (Export, Body or Import)
Action on Component Part (Add, Remove, or Modify)	Effect: □ : The affected component part must be changed ◇ : The affected component part could be changed — : No impact on the affected component part

Figure 8 How the matrix can be read

² i.e. add, remove or modify

³ i.e. export interface, body, or import interface

3.2 The matrix of propagation as assembly of several concepts

With this matrix, the designer or the developer can gain an overview of the mechanisms in order to trace all kinds of dependencies in the software he has to modify.

The component model introduced in chapter 2 shows that a component consists of three parts, namely the export interface, the body and the import interface. Modifications can be required in each of those parts. Three kinds of modifications have been defined in the matrix of propagation:

Add: A service has to be added in one of the tree component parts.

Remove: A service has to be deleted from one of the tree component parts.

Modify: A service has to be syntactically as well as semantically modified in one of the tree component parts.

Note that in this paper, only modifications on services will be considered. Modification of properties, as well as behavioral modification have been omitted.

The first step before tracing the impact of a given modification is to identify which kind of modification it is about (e.g. »deletion of an element from the body of a component«). Several kinds of modification have been classified in the first column of the matrix. For example, the »deletion of an element from the body of a component« can be repaired in the section »Body«, line »Modify« (see figure 9).

The matrix of propagation covers two main concepts to trace the impact of a modification. First, it can help to trace the impact of a modification into the modified component. For that, the modified component must be in a whitebox, or in a greybox view (see chapter 2.2).

The impact is shown in the second column (entitled »Modified Component«). If the modified component is in a greybox view, only the light grey parts into the column are relevant. Else, in a whitebox view, only the white parts have to be considered.

Secondly, the matrix helps by tracing dependencies between components. Here, there are two cases to consider:

Modified Component \ Modified Component		Modified Component			Importer Components		Exporter Components	
		Export	Body	Import	Single Importer	Set of Importers	Single Exporter	Set of Exporters
Export	Add		<input type="checkbox"/>	<input checked="" type="checkbox"/>	—	—		
	Remove		<input checked="" type="checkbox"/> *	<input checked="" type="checkbox"/> *	<input type="checkbox"/>	<input type="checkbox"/>		
	Modify		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Body	Add	—		<input type="checkbox"/>				
	Remove	<input type="checkbox"/>		<input checked="" type="checkbox"/> *				
	Modify	<input type="checkbox"/>		<input type="checkbox"/>				
Import	Add	—	—				<input type="checkbox"/>	<input type="checkbox"/>
	Remove	<input checked="" type="checkbox"/>	<input type="checkbox"/>				—	—
	Modify	<input checked="" type="checkbox"/>	<input type="checkbox"/>				<input type="checkbox"/>	<input type="checkbox"/>

Figure 9 The matrix of propagation

* These modifications may imply a dead code creation (see chapter 3.1).

- When the export interface of a component has to be modified, the modification can have an impact on the components which use it. In this case, the matrix shows either if a certain component is affected by the modification (column Importer components, subcolumn I), or the impact on the set of all the component which use the functionality of the modified component at all(column Importer components, subcolumn I').
- Symmetrically, the modification of the import interface of a component can have an impact on the component it uses ⁴. Here also, the matrix considers the case of one certain component (column Exporter components, subcolumn E), or the set of components used by the modified component (column Exporter components, subcolumn E').

⁴ because the export interface of these components could also have to be changed in order to cover the demanded functionality

3.3 Propagation paths

Step by step, the matrix shows the impact of an initial modification of a given element in a component on other elements in the same component and in others. For example, let us consider that »n« ($n \in \mathbb{N}$) is the number of the elements concerned by the initial modification. Due to the initial modification, a given number of them (»m«, where $m \in \mathbb{N}$), have also to be modified.

Thus the initial modification induces »m« new modifications in the system. Each of those induced modifications must also be verified with the matrix of propagation, and so on.

In order to get an overview of the impact of the initial modification, all the induced modifications will be registered in the form of a path. This path is called »Propagation Path«.

Since the system is considered to be consistent and reduced (see introduction of chapter 3), no induced modification will be left unhandled. A cycle could appear in the propagation path. In such a case, the engineer should be able know how the propagation goes on.

3.4 Example

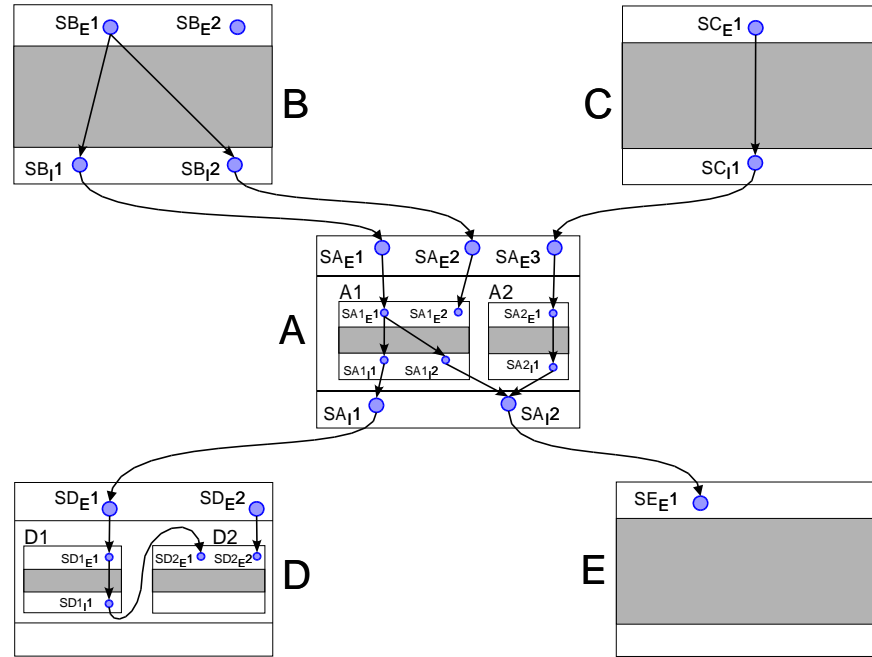


Figure 10

Subsystem S

The example of figure 10 illustrates a subsystem S which contains 9 components. Components A and D are compositional components which contain each of them 2 components, respectively A1 and A2, and D1 and D2. Components B, C, E, A1, A2, D1, and D2 are in a greybox view. For example, we can imagine that they are non-compositional, or still in the design phase, and only their interfaces are defined. In the following sections, we will trace the impact of the three kinds of modifications on this system, namely:

- 1 Addition of the service SC_I2 in the import interface of the component C
- 2 Deletion of the service SA_E3 of the export interface of the component A
- 3 Modification of the service SA_E1 of the export interface of the component A

In the first case (addition of SC_I2), we will trace the dependencies from a greybox view. The second and third example (deletion of SA_E3 , and modification of SA_E1) will be studied from both greybox and whitebox view.

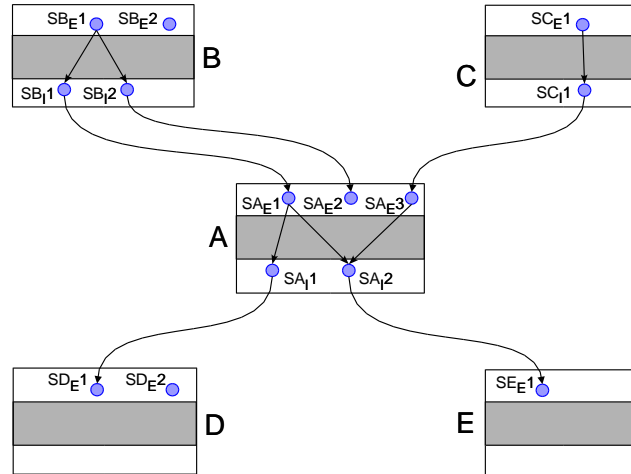


Figure 11 Subsystem S in a Greybox view

3.4.1 Addition of a service

Let us consider that a service named SC_I2 has to be added in the import interface of the component C (see figure 11).

The matrix of propagation shows that the addition of a service in an import interface can have an impact on an Exporter component.

Indeed, if the new required service is not already exported by any component, either an existing component has to be modified in order to provide it (Column »Exporter Components«, part »Single Export« of the matrix), or a new component which will export it has to be added in the system (Column »Exporter Components«, part »Sum of Exports« of the matrix). For the example, we will decide to add a new service named SE_E2 in the export interface of E on which will depend SC_I2 .

Thus, the propagation path of the addition of SC_I2 will start as follow:

$$Add_{C_I}(SC_I2) \rightarrow E_E(SE_E2)$$

Now, we must trace the impact of the addition of SE_E2 .

The matrix shows that the addition of a service in the export interface of a component could have an impact on the import interface of the same component. It means that, in order to provide the new service SE_E2 , E can possibly require the import of any service from another component. Let us consider that E requires a new service

» SE_I1 « in order to provide SE_E2 .

Thus, the propagation path of the addition of SC_I2 will continue as follow:

$$Add_{C_I}(SC_I2) \rightarrow E_E(SE_E2) \rightarrow E_I(SE_I2)$$

We are now in a similar case as at the beginning of this example: we must check the addition of SE_I2 in the import interface of E. After verifying in the matrix like in the case above, we know there are two possibilities: The service SE_I2 depends on an already exported service, or on a new one. Let us consider that SE_I2 matches with the service SD_E2 . In this case, the addition of SE_I2 requires no further addition or modification of any other service. Thus, the propagation path of the addition of SC_I2 is:

$$Add_{C_I}(SC_I2) \rightarrow E_E(SE_E2) \rightarrow E_I(SE_I2)$$

3.4.2 Deletion of a service

In this paragraph, we will trace the impact of the deletion of the service SA_E3 .

Before deleting this service, it is preferable to overview the consequences of such a deletion.

3.4.2.1 Deletion of a service from a greybox view

In this paragraph, we will trace the impact of the deletion of the service SA_E3 , considering A from a greybox view (see figure 11).

From a greybox view, the matrix of propagation shows that the deletion of an exported service may have an impact on the import interface of the same component, and on at least one of the Importer components.

First, let us consider the list of the Importer components of A. We must search in the example the components which import the service SA_E3 from A. Figure 11 shows that the service SC_I1 of the import interface of the component C depends on SA_E3 . Thus, if SA_E3 is deleted, the service SC_I1 required by component C is not being performed anymore. The designer has now 3 possibilities:

- 1 There is some component in the system providing a service which matches with SC_I1 . In this case, the use-relation between SC_I1 and SA_E3 must be deleted and a new relation between SC_I1 and the new service must be created. This case corresponds to a modification of SC_I1 .
- 2 The designer modifies any component in the system in order to provide a service which matches with SC_I1 . In this case also, the use-relation between SC_I1 and SA_E3 must be deleted and a new relation between SC_I1 and the new service must be created. This case corresponds to a modification of SC_I1 .
- 3 The designer modifies the component C in the way that it does not require the functionality of SC_I1 anymore. In this case, SC_I1 will be deleted.

Figure 11 shows that SC_I1 is the only service depending on SA_E3 . So, the propagation path of the deletion of SA_E3 will begin as follows:

$$Remove_{A_E}(SA_E3) \rightarrow C_I(SC_I1)$$

We know that SC_I1 has to be either deleted or modified. We must now consider the services depending on SC_I1 . The matrix of propagation shows that in the case of a deletion or modification of an imported service, the services of the export interface of the same component may be affected.

From a greybox view, we can see that the only service depending (indirectly) on it is SC_E1 . In this way, SC_E1 could be affected by the modification. This implies that SC_E1 must also figure in the propagation path.

Moreover, the example does not show which component depends on the service SC_E1 . In this way, the tracing of the impact of the deletion of SA_E3 on the Importer components of A can be closed. The resulting propagation path is:

$$Remove_{A_E}(SA_E3) \rightarrow C_I(SC_I1) \rightarrow C_E(SC_E1)$$

In the second step, we must consider the impact of the deletion of SA_E3 on the import interface of A. The matrix shows that the deletion of an exported service may create dead code in the import interface of the same component.

Figure 11 shows that SA_E3 only depends (indirectly) on SA_I2 . Moreover another service named SA_E1 depends on the same service. Thus, if SA_E3 is deleted, there will be no dead code creation in the import interface of A. The tracing of the impact of this deletion is in this case closed.

To summarize, the propagation path of the deletion of SA_E3 by considering S in a greybox view is the following:

$$Remove_{A_E}(SA_E3) \rightarrow C_I(SC_I1) \rightarrow C_E(SC_E1)$$

3.4.2.2 Deletion of a service from a whitebox view

After having traced the impact of the deletion of SA_E3 from a greybox view, we arrived to the conclusion that another component would be affected, namely component C. The resulting propagation path was the following:

$$Remove_{A_E}(SA_E3) \rightarrow C_I(SC_I1) \rightarrow C_E(SC_E1)$$

Now, we can observe from the whitebox view what will happen in the bodies of the affected components, namely A and C. The matrix of propagation shows that the deletion of an exported service may create dead code in the body of the component. In the figure 10, we can see that component A is a composite component which contains two subcomponents A1 and A2. Component C on the other hand, is still shown as a greybox. Thus, we can see which impact has the deletion of SA_E3 into the body of A.

If we look in details, the deletion of SA_E3 implies that the service $SA2_E1$ on which it depends won't be used anymore into the system. In order to maintain a reduced⁵ system, the service $SA2_E1$ must be also removed from the export interface of A2. This deletion must figure in the propagation path.

Now, we are in the same case as before: the deletion of a service in the export interface of a component, namely $SA2_E1$. Since A2 only provides the service $SA2_E1$, the deletion of this one implies a deletion of the whole component (because the system must be maintained reduced).

We must now verify if the deletion of A2 has any impact on the import interface of A. The matrix of propagation shows that the deletion of an element in the body of a component may create dead code in its import interface. Figure 10 shows that the service $SA2_I1$ of the import interface of A2 depends directly on the service SA_I1 . Moreover $SA1_I2$ also depends on SA_I1 . So, the deletion of $SA2_I1$ has no impact on the service SA_I1 which is still needed by $SA1_I2$. In this way, the deletion of SA_E3 implies only the deletion of component A2 in the body of A.

If we summarize the whole in a propagation path, this one sees as follows:

$$Remove_{A_E}(SA_E3) \rightarrow C_I(SC_I1) \rightarrow C_E(SC_E1) \\ \searrow A2_E(SA2_E1) \rightarrow A2$$

⁵ It is a prerequisite for using the matrix of propagation (see introduction of chapter 3)

3.4.3 Modification of a service

The third kind of change we want to trace is the »modification« of a service. The word modification is quite abstract, so that the expression »modification of a service« can have a lot of significations. It can mean for example that the static specification, or the dynamic specification of the service has to be changed.

The goal of this paper is first to introduce the matrix of propagation as a method to trace dependencies in component-based software systems. In this way, we will consider the notion of »modification« in its global meaning.

In the following paragraphs, we will trace the impact of a modification of the service SA_E1 on the components A, B, C, D and E. We will start with a greybox view of both components, and then in more details with a whitebox view.

In this paper, we considered that if a given modification of an element has an impact on another element, this impact is a modification or a deletion of the second element (in case of deletion, the deleted element would be replaced by a new one). In this section, in order to simplify the example, we will consider that the impact of a modification is always modification (for the cases »addition« and »deletion«, see sections 3.4.1 and 3.4.2).

3.4.3.1 Modification of a service from a greybox view

From a greybox view, the matrix of propagation shows that the modification of service in an export interface may have an impact on the import interface of the same component, and has an impact on at least one of its Importer components (since the system is reduced, at least one component imports the modified service).

Figure 11 shows that one service called SB_I1 depends directly on SA_E1 , and that SA_E1 depends indirectly on the two services SA_I1 and SA_I2 . In this way, if SA_E1 is modified, all SB_I1 , SA_I1 , SA_I2 could be concerned, i.e they could also need a modification. The propagation path of the modification of SA_E1 will start with three cases, as follows:

$$Modify_{A_E}(SA_E1) \begin{matrix} \nearrow A_I(SA_I2) \\ \rightarrow A_I(SA_I1) \\ \searrow B_I(SB_I1) \end{matrix}$$

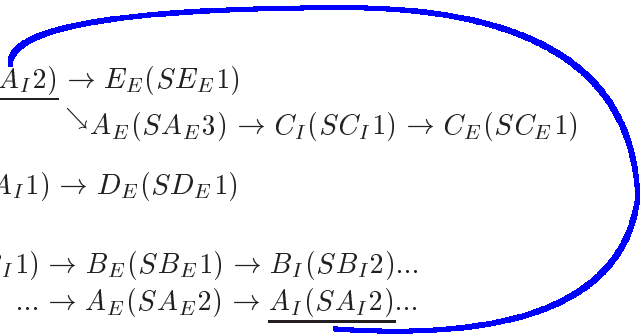
The matrix of propagation shows that the modification of service in an import interface may have an impact on the export interface of the same component, and has an impact on at least one of its Exporter components (since the system is reduced, the modified service depends on another service which will also have to be modified).

Now we must consider the three potential modifications of SB_I1 , SA_I1 , SA_I2 . The

example shows that only one service depends on SB_I1 , namely SB_E1 . Moreover, SA_I1 depends on SD_E1 , and SA_I2 on SE_E1 . Thus, the service SB_E1 could have to be modified, and SD_E1 , and SE_E1 will have to be modified. The propagation path of the modification of SA_E1 continues as follows:

$$\begin{array}{l} \nearrow A_I(SA_I2) \rightarrow E_E(SE_E1) \\ Modify_{A_E}(SA_E1) \rightarrow A_I(SA_I1) \rightarrow D_E(SD_E1) \\ \searrow B_I(SB_I1) \rightarrow B_E(SB_E1) \end{array}$$

In these three new states of the propagation, the tracing method is the same like in the beginning: a service of the import interface has to be modified, and the modification will probably have an impact on imported services... and so on. If we continue so on the basis of the example of figure 11, we will obtain the following propagation path:

$$\begin{array}{l} \nearrow \frac{A_I(SA_I2) \rightarrow E_E(SE_E1)}{\searrow A_E(SA_E3) \rightarrow C_I(SC_I1) \rightarrow C_E(SC_E1)} \\ Modify_{A_E}(SA_E1) \rightarrow A_I(SA_I1) \rightarrow D_E(SD_E1) \\ \searrow B_I(SB_I1) \rightarrow B_E(SB_E1) \rightarrow B_I(SB_I2) \dots \\ \dots \rightarrow A_E(SA_E2) \rightarrow \underline{A_I(SA_I2)} \dots \end{array}$$


As we can see in this propagation path, it can happen that an element appears several times (see blue line in the propagation path). In such a case, the designer has to decide if the modifications on the same element are compatible. In order to simplify the example, we considered that the modifications are always compatible. In this way, both modifications of SA_I2 have the same impact. Thus, the propagation of one of both cases would be stopped.

3.4.3.2 Modification of a service from a whitebox view

After tracing the impact of the modification of SA_E1 from a greybox view, we will go into more details with a whitebox view of the components.

The composition of components was hidden in the greybox view. Now, we have to repair the impact of the modification of their imported and exported services on their bodies. Here, two cases can appear:

- 1 The propagation path studied from a greybox view shows the direction of the propagation of a modification through indirect dependencies. For example $SA_E1 \rightarrow SA_I1$ shows that the modification of SA_E1 has indirectly an impact on SA_I1 , or $SA_I2 \rightarrow SA_E1$ shows that the modification of SA_I2 has indirectly an impact on SA_E3 . This indirect impact shows actually that »something« happens between SA_E1 and SA_I1 (or between SA_I2 and SA_E3) into the body of A. This »something« will be traced in details from the whitebox view.
- 2 The propagation path studied from a greybox view also shows standalone services of an interface of composite components (the last service of any branch of the path, e.g. SD_E1). The modification of such services must also be traced into the body of the component.

Let us begin with the modifications hidden by the indirect dependencies of compositional components. For that, we have to repair such dependencies in the propagation path of paragraph 3.4.3.1. They are the following: $A_E(SA_E1) \rightarrow A_I(SA_I1)$, $A_E(SA_E1) \rightarrow A_I(SA_I2)$, $A_E(SA_E2) \rightarrow A_I(SA_I2)$, and $A_I(SA_I2) \rightarrow A_E(SA_E3)$. The matrix of propagation shows that the modification of a service in the export interface has necessarily an impact into the body. After seeing figure 10, we can conclude that $A_E(SA_E1) \rightarrow A_{1E}(SA_{1E}1)$. Since A1 is in a greybox view, we can apply the same rules as in chapter 3.4.3.1, and obtain the following result:

$$A_E(SA_E1) \rightarrow A_{1E}(SA_{1E}1) \rightarrow A_{1I}(SA_{1I}1) \searrow A_{1I}(SA_{1I}2)$$

Here, the matrix shows that the modification of an element in the body of a component may create dead code into the import interface. Thus, with the same example, we will obtain:

$$A_E(SA_E1) \rightarrow A_{1E}(SA_{1E}1) \rightarrow A_{1I}(SA_{1I}1) \rightarrow A_I(SA_I1) \searrow A_{1I}(SA_{1I}2) \rightarrow A_I(SA_I2)$$

If we do the same for the indirect dependency $A_E(SA_E2) \rightarrow A_I(SA_I2)$, we will have: $A_E(SA_E2) \rightarrow A_{1E}(SA_{1E}2) \rightarrow A_{1I}(SA_{1I}2) \rightarrow A_I(SA_I2)$.

We will now study the case of the indirect dependency corresponding to $A_I(SA_I2) \rightarrow A_E(SA_E3)$. The matrix shows that the modification of a service in the import interface of a component has necessarily an impact into its body. Indeed, figure 10 indicates that $SA2_I1$ depends directly on SA_I2 . In this way, we obtain $A_I(SA_I2) \rightarrow A2_I(SA2_I1)$. Since $A2$ is in a greybox view, we can apply the same rules as in chapter 3.4.3.1. The result is the following:

$$A_I(SA_I2) \rightarrow A2_I(SA2_I1) \rightarrow A2_E(SA2_E1)$$

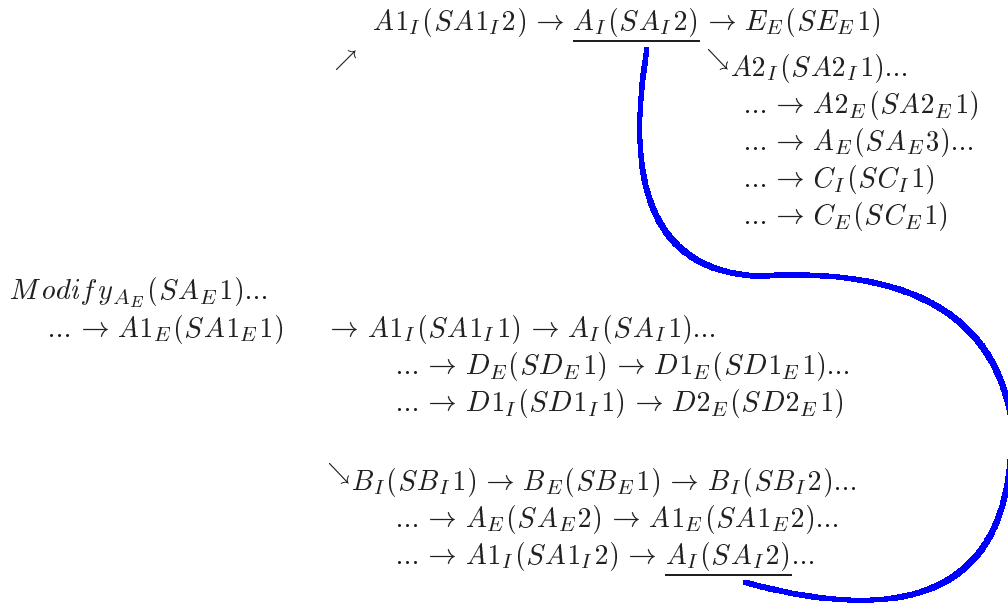
The matrix shows that the modification of an element into the body of a component has necessarily an impact on its export interface. For the case $A_I(SA_I2) \rightarrow A_E(SA_E3)$ (which was studied from a greybox), view we will have in a whitebox view the following path:

$$A_I(SA_I2) \rightarrow A2_I(SA2_I1) \rightarrow A2_E(SA2_E1) \rightarrow A_E(SA_E3)$$

As a last step, we will trace the impact of modifications of services mentioned at the end of a path (like explained in number 2). In the propagation path from paragraph 3.4.3.1, we can find such a service, namely SD_E1 . If we use the matrix like just before, we can see that the modification of this exported service has an impact on the service $SD1_E1$. Since $D1$ is in a greybox view, we can adopt the same method like in paragraph 3.4.3.1 to find out that the modification of $SD1_E1$ has an impact on $SD1_I1$, and that if $SD1_I1$ is modified, $SD2_E1$ will be also concerned. Moreover, since SD_E2 directly depends on $SD2_E1$, the modification of $SD2_E1$ concerns also SD_E2 . The modification of SD_E1 will be traced from a whitebox view as follows:

$$D_E(SD_E)1 \rightarrow D1_E(SD1_E)1 \rightarrow D1_I(SD1_I)1 \rightarrow D2_E(SD2_E)1 \rightarrow D_E(SD_E)2$$

By summarizing the results of paragraph 3.4.3.1 and the results obtained in the above paragraphs, we can say that the propagation path of the modification of the service SA_E1 of the component A is the following:



4 Conclusion

The matrix of propagation was introduced to trace the impact on modifications on components. In this paper, various aspects of using the Matrix have been evaluated. However, it has to be taken into consideration that the Matrix of Propagation is in an early stage of research.

Therefore, efforts must be invested in terms of research in this field in order to develop and extend this idea. Since it is out of scope of this paper to investigate all possible aspects regarding the matrix, some areas can be proposed for future works. For example, the term »modify« in the matrix would be treated in its global meaning. In the future, several aspects of a modification will have to be identified, e.g. semantical modification of an element in a component, or modification of the behavior of a component etc. Additionally, the notion of component properties could be introduced in the future. Properties may be added, deleted, modified, but they can also be strengthened or weakened. Therefore a new matrix could be developed in order to describe the impact of the modification of properties in a better way.

After achieving a certain level of maturity, the matrix should be implemented in a software tool. This tool will automatically calculate the propagation path of a given modification of any component of a system. Thus, it could become a very helpful tool for any component based software developer.

On the other side the matrix cannot be considered as a universal solution for diagnosing all the problems arising inside a component based system. For instance, binary codes inside components have not been considered in this paper. In such cases, the matrix could be combined with tools and techniques which analyze these codes. Certainly more research has to be done in this direction to find out the ways to extend it.

The component model considered in this paper is still under way. Within the scope of the project »Continuous Software Engineering« [2][3][6], the component specifications of this component model are under research [6]. In this way, the technique of propagation matrices as such cannot be actually used at the implementation phase and after deployment. Future work should adapt this idea for technologies which are currently used, e.g. Enterprise Java Beans.

5 Bibliography

- [1] C. Berthomieu: *Matrix of propagation - A concept to trace dependencies in component-based software systems*.
Diplomarbeit, Technische Universität Berlin, 2002.
- [2] S. Mann, A. Borusan, H. Ehrig, M. Große-Rhode, R. Mackenthun, A. Sünbül, H. Weber: *Towards a Component Concept for Continuous Software Engineering*.
Technischer Bericht 55/00, Fraunhofer ISST, Berlin, 2000.
- [3] A. Borusan, M. Große-Rhode, H. Ehrig, R-D. Kutsche, S. Mann, J. Padberg, A. Sünbül, H. Weber: *Kontinuierliches Engineering: Grundlegende Terminologie und Basiskonzepte*.
Interner Bericht des Projekts Kontinuierliches Engineering für Evolutionäre IuK-Infrastrukturen, Fraunhofer ISST, 2000.
- [4] M. Große-Rhode, R-D. Kutsche, F. Bübl: *Concepts for the Evolution of Component-Based Software Systems*.
Technical Report 2000/11, Technische Universität Berlin, Dep. of computer science, 2000.
- [5] F. Bübl: *Introducing Context-Based Constraints*.
In proc.: Fundamental Approaches to Software Engineering (FASE), Grenoble, France, April 2002. Eds.: R. Kutsche, H. Weber.
Springer Verlag, Lecture Notes in Computer Science 2306.
- [6] U. Kriegel: *ComponentML - Eine Markup-Language zur Spezifikation von Komponenten , version 0.6*.
Interner Bericht des Projekts Continuous Software Engineering, Fraunhofer ISST, 2000.
- [7] S. Comella-Dorda , K. Wallnau, R. C. Seacord, J. Robert: *A Survey of Legacy System Modernization Approaches*.
Technical Report 2000, Carnegie Mellon University, Software Engineering Institute, 2000.
<http://www.sei.cmu.edu/publications/documents/00.reports/00tn003.html>
- [8] N. Weideman, D. Smith, S. Tilley: *Approaches to Legacy System Evolution*
Technical Report 1997, Carnegie Mellon University, Software Engineering Institute, 1997.
<http://www.sei.cmu.edu/publications/documents/97.reports/97tr014/97tr014abstract.html>

- [9] *Stanford Encyclopedia of Philosophy, Modal Logic*.
The metaphysics Research Lab, Center for the Study of Language and Information, Stanford University.
<http://plato.stanford.edu/>
- [10] H. Schumann, M. Goedicke, *Component Oriented Software Development with II*, paper, 1994.
- [11] G.T. Heineman, W.T. Council: *Component-Based Software Engineering, Putting the Pieces Together*.
Addison Wesley, 2001.
- [12] P. Herzum, O. Sims: *Business Component Factory*.
OMG Press, Wiley, 2000.
- [13] J. Rumbaugh, G. Booch, I. Jacobson: *The Unified Modeling Language Reference Manual*.
Addison Wesley, 1999.
- [14] J. Cheesman, J. Daniels: *UML Components*.
Addison Wesley, 2000.
- [15] M. Fowler: *UML Distilled*.
Addison Wesley, 1997.
- [16] D.F. D'Souza, A.C. Wills: *Objects, components, and frameworks with UML, The Catalysis Approach*.
Addison Wesley, 1998.
- [17] Ed Roman: *Mastering Enterprise Java Beans and the Java2 Platform*.
Wiley, 1999.
- [18] R. Burkhadt: *UML- Unified Modeling Language, Objektorientierte Modellierung Für die Praxis*
Addison Wesley, 1997.
- [19] C. Szyperski: *Component Software - Beyond Object-Oriented Programming*.
Addison Wesley, 1997.