Formalization of component substitutability

Meriem Belguidoum, Fabien Dagnat

Formalization of Component Substitutability

Meriem Belguidoum[1] and Fabien Dagnat[2]

Computer Science Department
ENST Bretagne, Technopole Brest-Iroise
Brest, France

Abstract

Component-Based Software Engineering (CBSE) is increasingly used to develop large scale software. In this context, a complex software is composed of many software components which are developed independently and which are considered as black boxes. Furthermore, they are assembled and often dependent from each other. In this setting, component upgrading is a key issue, since it enables software components to evolve. To support component upgrading, we have to deal with component dependencies which need to be expressed precisely. In this paper, we consider that component upgrade requires managing substitutability between the new and the old components. The substitutability check is based on dependency and context descriptions. It involves maintaining the availability of previously used services, while making sure that the effect of the new provided services do not disrupt the system and the context invariants are still preserved. We present here a formal definition and a verification algorithm for safe component substitutability.

Keywords: Component software, Safety, Substitutability, Upgrading.

1 Introduction

Component based software has gained recognition as the key technology for building high quality and large software. In this setting, sharing collections of components has become common practice for component oriented applications. These components are independently produced and developed by different providers and reused as black boxes making it necessary to identify component dependencies to guarantee interoperability.

[1] Email: meriem.belguidoum@enst-bretagne.fr
[2] Email: fabien.dagnat@enst-bretagne.fr

According to Szyperski's definition [11], a component is a unit of composition with contractually specified interfaces and explicit dependencies. An interface describes the provided and the required services of a component. Software consists of the assembly of components in an architecture, by binding a required interface of one component to an offered interface of another component.

In this context, upgrading a component is difficult because this component may be used by several software applications. More generally, replacing a component C_{old} with a component C_{new} in a system $\mathcal{S}$ requires that it does not disrupt $\mathcal{S}$. This property is often described as substitutability [5].

Several techniques exist to ensure substitutability between components, see for example [12,13]. All these approaches are built upon the substitution principle of Liskov introduced in [1] in the context of object oriented programming. They use the interface type to define a subtyping relation between components and then authorize C_{new} to replace C_{old} only if C_{new} is a subtype of C_{old}. Various forms of those types exist, starting with the classical interface type [10] and enhancing them with behavioral description such as automata for example [6]. Some related research [12,13] show that the resulting condition of pure subtyping ensures *safety* of the replacement but is too restrictive. Recent work [5] has shown the limits of this approach and proposes a less restrictive notion of substitutability depending on the context. In this setting, C_{new} may safely replace C_{old} in certain systems. In fact, all interfaces not used in the context are ignored when ensuring the subtyping.

To extend our previous work on the formalization of safe component installation and deinstallation [2], we tried to define contextual substitutability to build a safe replacement operation following the previously cited approaches. But it appeared that the resulting rule needs to be enhanced to reach *safety*. While in other work the new services provided by C_{new} do not have to meet any requirement, in our setting they may conflict with the context requirements.

Generally, replacing a component C_{old} by a new one C_{new} has an effect on the context and to maintain *safety*, we have to check that effect will not break system invariants. In previously mentioned work, the only effect taken into account is the services that C_{new} provides. Therefore, ensuring substitutability consist in ensuring compatibility of C_{new} provided services with the component requirements that were previously using C_{old} services. This compatibility can have different contract levels (syntactical, behavioral, synchronisation and quality of service) as described in [4]. This paper advocate adding the verification of component upgrading effects (upgrade) on the target system. In the deployment, it appears that upgrade effect must not disturb the target system. For this, we propose a substitution principle ensuring that (1) the

new component still provides all the services used in the context (as usual) and that (2) new provided services do not conflict with this context (effect verification). The formalization of this substitutability enables us to provide a safe and flexible replacement operation for our deployment system.

The paper is organized as follows. Section 2 introduces our dependencies description and illustrates it with the example of a mail server in Linux GNU. In Section 3, we present our substitutability approach with a progressive refinement of substitutability definitions. Section 5 describes the substitutability checking algorithm. Section 4 illustrates some substitutability examples. Section 6 discusses related work. Finally, Section 7 concludes and discusses future work.

2 Dependency description

In this section, we present the precise definition of the relation between a required and a provided service, either of the same component or of two different components. Such a relation is called a *dependency*.

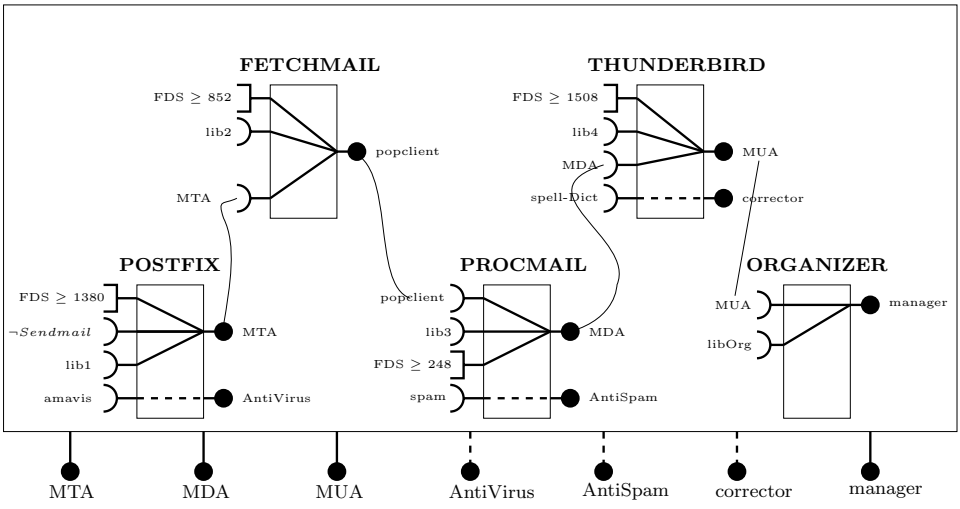


Fig. 1. A mail server assembly

Fig . 1 illustrates a simplified architecture of a mail server on a Linux system. It is composed of five components: `POSTFIX`, an SMTP server playing the role of a Mail Transport Agent (`MTA`), `FETCHMAIL` that recovers mails from a distant server like Pop or IMAP using a mail transport protocol, `PROCMAIL`, a Mail Deliver Agent (`MDA`) that manages received mail and enables, for example, mail to be filtered. `THUNDERBIRD`, a mail manager for reading and composing mail called a Mail User Agent (`MUA`). Finally, `ORGANIZER` is an inbox organizer, which allows mailing lists, web pages and users e-mails to be managed. Each

component is represented in Fig. 1 by a rectangle with the required interfaces (half circles and square brackets on the left side) and the provided interfaces (black circles on the right side). Requirements in the left hand side of a component, may be of two kinds: (1) software requirements (i.e., services provided by other components), for example libraries (half-circles: $lib1$) or (2) system requirements expressed by comparison of variables with values represented by square brackets, for example requiring a certain amount of free disk space ($FDS \geq 1380Ko$).

In this example[3], the two forms of dependencies, respectively *intra-dependencies* and *inter-dependencies*, are represented respectively by lines inside components and links between components (like `PROCMAIL` and `THUNDERBIRD`, `PROCMAIL` provides `MDA` and `THUNDERBIRD` requires it). There are three main kinds of dependencies, either *mandatory*, *optional* or *negative*:

- a mandatory dependency (represented by a solid line) is a firm requirement. If it is not fulfilled, installation is not possible. For example, `POSTFIX` needs a terminal with a specific libraries ($lib1$), an amount of free disk space ($FDS \geq 1380ko$).
- an optional dependency (represented by a dashed line) specifies that the component may provide optional services. Such services may not be provided (if their requirements are not fulfilled) without preventing the installation. For example, `POSTFIX` may provide a service for scanning messages against viruses if the service *amavis* is available. Otherwise `POSTFIX` can be installed and provides the `MTA` service, but the service $AntiVirus$ is not provided.
- a negative dependency (expressed by a negation) specifies a conflict forbidding installation. The conflict may be due to a service or a component. For example, as presented in Fig. 1, `POSTFIX` cannot be installed if the component `SENDMAIL` (another component providing `MTA`) is already installed in the target system.

Intra-dependencies are defined by the producer of the component and used to perform installation. Inter-dependencies result from installation and are used to perform deinstallation and replacement. The two notions are briefly presented below, more details on these concepts are given in [2].

2.1 Intra-dependencies

The intra-dependency description language uses the concepts of *dependency* and *predicate* defined by the following grammar where s represents the name

[3] To simplify the figure, only some interesting dependencies are represented.

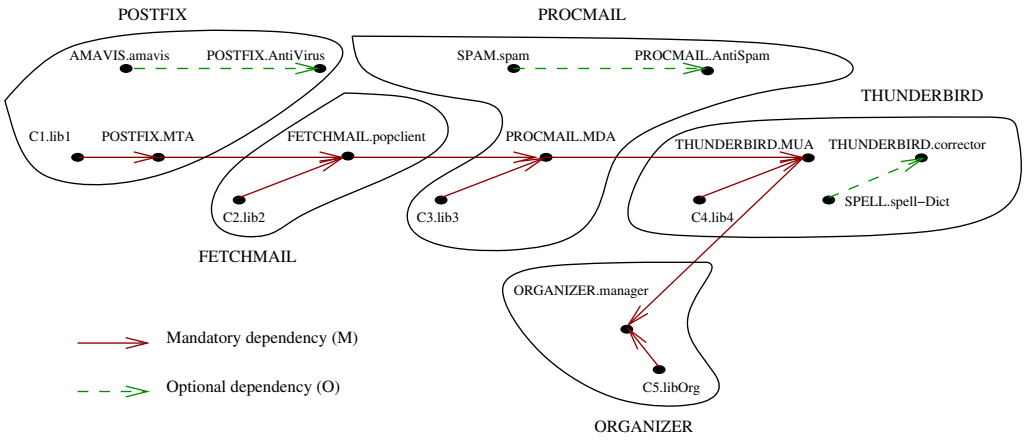


Fig. 2. Dependency graph of the mail server of Fig. 1

of a service and c the name of a component:

$$D ::= P \Rightarrow s \mid D \bullet D \mid D \sharp D \mid ?\, D \qquad P ::= \mathit{true} \mid P \wedge P \mid P \vee P \mid R$$

$$R ::= [v\ O\ \mathit{val}] \mid \neg\, s \mid \neg\, c \mid c.s \mid s \qquad O ::= > \mid \geq \mid < \mid \leq \mid = \mid \neq$$

The precise semantics of these operators is described in detail in [2]. Intuitively, a dependency may be the conjunction $\bullet$ or the disjunction $\sharp$ of two dependencies, an optional dependency $?$ or a simple dependency $P \Rightarrow s$ specifying the requirements P of the service s. If these requirements are fulfilled the service s is available. The requirements are expressed in a first order predicate language with five conditions (R) expressing a comparison of an environment variable with a value ($[v\ O\ \mathit{val}]$), a conflict with a service ($\neg\, s$), with a component ($\neg\, c$), the requirement of a service provided by a precise component ($c.s$) or any component (s). Examples of such predicates appear in Fig. 1 on the required interfaces represented in the left-hand side of a component. For exemple the dependency description of the component `POSTFIX` is : $([FDS \geq 1380] \wedge (\mathit{sendmail}) \wedge \mathit{lib} \Rightarrow MTA) \bullet ?(\mathit{Amavis} \Rightarrow AV)$

2.2 Inter-dependencies

When a component is installed in a system $\mathcal{S}$, each of its requirements is fulfilled by binding it to any existing component of $\mathcal{S}$ satisfying the requirement. This binding is what we call an *inter-dependency*. It is the result of installation and is required to ensure safe deinstallation and replacement. We have chosen to represent inter-dependencies by a *dependency graph* (see [2] for more details). A node of a dependency graph is an available service s with its

provider ($c.s$) and an edge is a pair of nodes $n_1 \mapsto n_2$ meaning that n_2 requires n_1. Each edge is labeled (above the arrow) by the kind of dependency, either mandatory M or optional O. Fig. 2 presents the dependency graph of the mail server of Fig. 1. We can see that some solid (resp. dashed) lines inside components in the Fig. 1 (intra-dependencies) are reflected in Fig. 2 by solid (resp. dashed) edges. For example, `POSTFIX` depends on *lib*1 which is provided by a component `C1`, so the used service is `C1`.*lib*1. This dependency is a mandatory one (solid edge in Fig. 2 $C1.lib \mapsto POSTFIX.MTA$). `POSTFIX` has also an optional dependency, it can provide an Anti virus (`POSTFIX`.$AntiVirus$) if a service *amavis* provided by a component (for example `AMAVIS`) is available. This dependency is represented in the graph by a dashed edge. The system requirements (like $FDS \geq 1380$) and negative dependencies (`SENDMAIL`) are not represented in the graph. The inter-dependencies between components in Fig. 1 are represented in the graph as mandatory edges between services, for example the service *popclient* provided by `FETCHMAIL` is linked with service *popclient* required by `PROCMAIL`, the corresponding edge in Fig. 2 is $FETCHMAIL.popclient \mapsto PROCMAIL.MDA$.

3 What is substitutability?

In this section, we present and analyze progressively the substitutability problem, we propose definitions and rules to check the correctness and safety of substitutability. In general, two forms of compatibility between components can be defined: vertical compatibility and horizontal compatibility. The vertical compatibility is called substitutability, it expresses the requirements that allow the replacement of one component by another (C_{old} by C_{new} in Fig. 3). The horizontal compatibility expresses connexion between a provided service of a component and a required service of another component (C_{old} used by C_{client} in Fig. 3). When substituting the component C_{old} for the component C_{new}, we have to ensure that the component C_{client} can use the services provided by C_{new} as it used previously those provided by C_{old} and the new provided services do not conflict with C_{client} and all other client components.

3.1 Substitutability definitions

Following the current trend, we define two kinds of substitutability, one addressing substitutability in a particular context and the other independent of the context. The definitions of strict and contextual substitutability are given below and are inspired by those of Brada in [5].

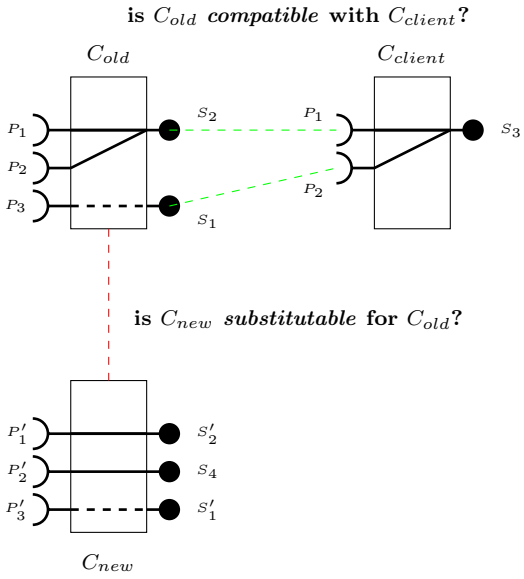


Fig. 3. Vertical and horizontal compatibility

Definition 3.1 (Strict substitutability)
A component C_{old} is *strictly substitutable* for a component C_{new}, if the latter can replace C_{old} in all contexts.

Definition 3.2 (Contextual substitutability)
A component C_{old} is *substitutable* in a context Ctx for a component C_{new} if the latter can replace C_{old} in the context Ctx.

Contextual substitutability is related to the context which represents the resources and the architecture of the target system. Ideally, it could be the union of the dependencies of all components (part of the system). The resulting description of the context would be a huge logical term. Its manipulation when deciding whether to authorize a deployment operation would be difficult and expensive (in calculation). Thus, we have chosen instead a safe approximation of the context description. The context definition is presented in [2]. It is summarized as follows:

Definition 3.3 (Context)
The *Context* is composed of (1) an environment $\mathcal{E}$ storing the values of environment variables (OS, disk space, etc.), (2) a set $\mathcal{C}$ of four-tuples $(c, \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c)$ storing for each installed component c its provided services $\mathcal{P}_s$, forbidden services $\mathcal{F}_s$ and forbidden components $\mathcal{F}_c$ [4] and (3) a dependency graph $\mathcal{G}$ storing

[4] The required services of a component are stored in the dependency graph not in the component tuple.

the dependencies (the required and the provided services of each component and the relation between them).

3.2 Component substitutability

To decide whether a component C_{new} can substitute a component C_{old}, it is necessary to compare what they provide and what they require. Indeed, the provided (or required) services of C_{new} can be the same or different from those of C_{old}. We therefore have to study all the possibilities. Fig. 4 depicts the different possible relations between the old and the new set of provided (resp. required) services:

- case 1: the set of provided (resp. required) services of C_{new} is included in the set of provided (resp. required) services of C_{old};
- case 2: the set of provided (resp. required) services of C_{new} and C_{old} are equal;
- case 3: the set of provided (resp. required) services of C_{old} is included in the set of provided (resp. required) services of C_{new};
- case 4: the two sets are different from each other and can have some services in common.

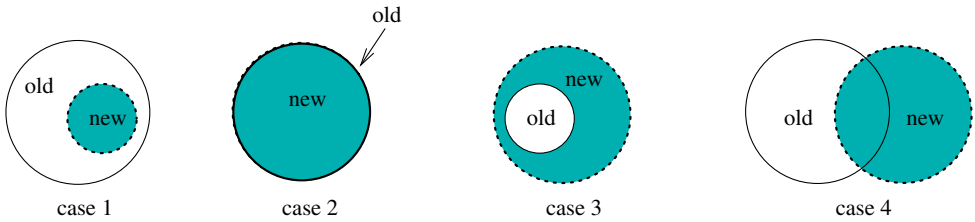


Fig. 4. Comparison according to the old and the new service sets

There are four cases for provided services combining with four cases for required services leading to sixteen possibilities. To illustrate these cases, we suppose that the component C_{old} provides the services PS_1 and PS_2 and requires the services RS_1 and RS_2. Table 1 represents the different forms that the component C_{new} may have, depending on its provided and required services. Each cell of the table corresponds to numerous possible components and is here represented by one possible component for illustrative purposes only.

In fact, the sixteen possible cases can be refined to table 2 below, containing eight possibilities combining only three conditions:

- ensure new requirements (**NR**) of C_{new}. For example, in line 1, is RS_3 satisfied?

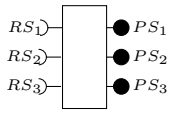
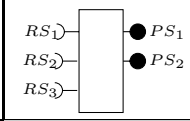
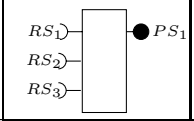
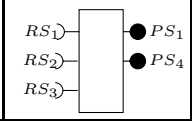
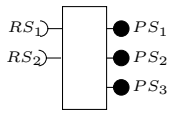
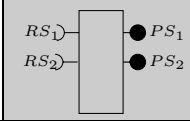
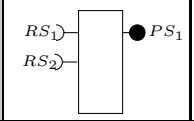
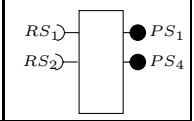
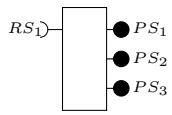
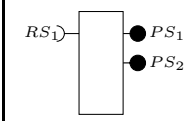
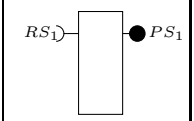
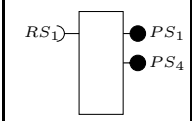
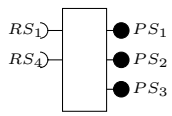
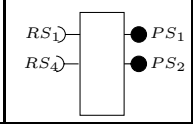
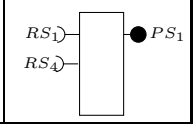
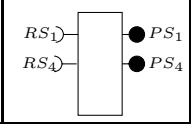
Requires \ Provides	more	same	fewer	different
more				
same				
fewer				
different				

Table 1
Substitutability possibilities

- ensure no conflicts (**NC**) between the new services of C_{new} and the system. For example, in column 1, is PS_3 in conflict with the system?
- ensure that all previously provided services which are not provided by C_{new} are not necessary for the system (**NON**). For example, in column 3, is PS_2 (previously provided by C_{old} and not provided by C_{new}) necessarily used?

Requires \ Provides	more	same	fewer	different
same / fewer	NC		NON	NC+NON
different / more	NR+NC	NR	NR+NON	NR+NC+NON

Table 2
Substitutability conditions

This table shows the different substitutability conditions on the context. The only cell corresponding to strict substitutability is the empty one. The condition is then that C_{new} requires the same thing or less than C_{old} and provides the same services. The seven other cells represent contextual substitutability. Necessary and sufficient conditions (NSC) for strict and contextual substitutability are defined as follows:

NSC 1 (Strict substitutability) *A component C_{old} is strictly substitutable for a component C_{new} iff they provide the same services and C_{new} has the same or fewer requirements than C_{old}.*

NSC 2 (Contextual substitutability) *A component C_{old} is substitutable for a component C_{new} in a context Ctx iff:*

- *all the new requirements of C_{new} are satisfied in Ctx (NR).*
- *none of the new provided services is in conflict with Ctx (NC).*
- *none of the services provided by C_{old} not provided by C_{new} is used necessarily within Ctx (NON).*

Compared to existing substitutability approaches, the condition (NC) is original because it enables to take into account various form of component effects on the context (potential conflicts that can occur due to the new component) and maintaining the safety of the system. In an extension of our system not presented here, we have the specification of non-functional properties. Replacing a component by another may have an impact on the system properties and therefore may be forbidden. An example of a such substitution is further discussed in Section 6.

3.3 Ensuring substitutability in our context

To ensure substitutability in our system, it is necessary to :

- determine which case is examined,
- evaluate the corresponding conditions among NR, NC and NON.

Using our dependency descriptions presented in section 2 it is easier to calculate and compare provided services than required ones. In our approach, we do not consider requirements because the conditions are described in predicate logic and it is rather complex to compare requirements for each provided service. Therefore, we check substitutability according to provided services only as follows:

(i) **NR** and **NC**: to check the new requirements (NR) and prevent new conflicts (NC), we reassess the installability condition of the new component C_{new}. This condition ensures, on the one hand, that all the component requirements are fulfilled (NR) and, on the other hand, that the provided services are not in conflict with context (NC). Therefore, the NR and NC conditions correspond to ensuring installability as presented in [2]: $(Ctx \vdash C_{new} : D_{new})$.

(ii) **NON**: this condition is based on the calculation of provided services from the right-hand member of the dependencies. So, for each provided service of C_{old} which is not provided by C_{new} we have to check that it has no mandatory dependency (i.e., it is a leaf in the dependency graph) or it is only used (directly or indirectly) by optional services (in the graph,

all paths coming from it must be optional, see definition 3.4). In fact, it corresponds to the deinstallation requirements of [2].

Definition 3.4 (Mandatory dependencies (MD)) The set of mandatory dependencies (MD) of a service s provided by a component c ($c.s$) in a dependency graph ($\mathcal{G}$) is the set of nodes which use necessarily this service. It is defined as :
$MD(\mathcal{G}, c.s) = \bigcup\{\{c'.s'\} \cup MD(\mathcal{G}, c'.s') \mid c.s \overset{\mathsf{M}}{\longmapsto} c'.s' \in \mathcal{G}\}$

The condition NON can be expressed as follows:
$\bigcup\{(MD(\mathcal{G}, C_{old}.s) \mid s \in (C_{old}.\mathcal{P}_s \setminus C_{new}.\mathcal{P}_s)\} = \varnothing$

We summarize the different substitution conditions for the four cases illustrated in Fig. 4 and table 2 as follows:

- providing more (case 3): C_{new} is installable (NR+NC);
- providing the same (case 2): C_{new} is installable (NR);
- providing less (case 1): C_{new} is installable (NR) and services from $C_{old}.\mathcal{P}_s \setminus C_{new}.\mathcal{P}_s$ are not used necessarily (NON);
- different (case 4): C_{new} is installable (NR+NC) and services from $C_{old}.\mathcal{P}_s \setminus C_{new}.\mathcal{P}_s$ not used necessarily (NON).

4 How substitutability is checked?

The substitutability handled in our system is only a contextual one. We have to calculate the context denoted Ctx without C_{old} ($Ctx \setminus C_{old}$), i.e., simulate the effect of removing from the context the component C_{old} with its four-tuple $(C_{old}, \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c)$. Then, we have to check the installability of C_{new} in the resulting context ($Ctx \setminus C_{old}$). The formal definition of contextual substitutability is presented below:

Theorem 4.1 ***Contextual substitutability***
A component C_{old} is substitutable for a component C_{new} in a context Ctx if:

- *C_{new} is installable in $Ctx \setminus C_{old}$;*
- *all provided services of C_{old} which are not provided by C_{new} ($C_{old}.\mathcal{P}_s \setminus C_{new}.\mathcal{P}_s$) **must** not be used necessarily in the context (NON condition):*
  $\bigcup\{(MD(\mathcal{G}, C_{old}.s) \mid s \in (C_{old}.\mathcal{P}_s \setminus C_{new}.\mathcal{P}_s)\} = \varnothing$

Substitutability is checked as depicted in the diagram of Fig. 5. First, the new context Ctx' is calculated without the old component C_{old}. So, Ctx' is

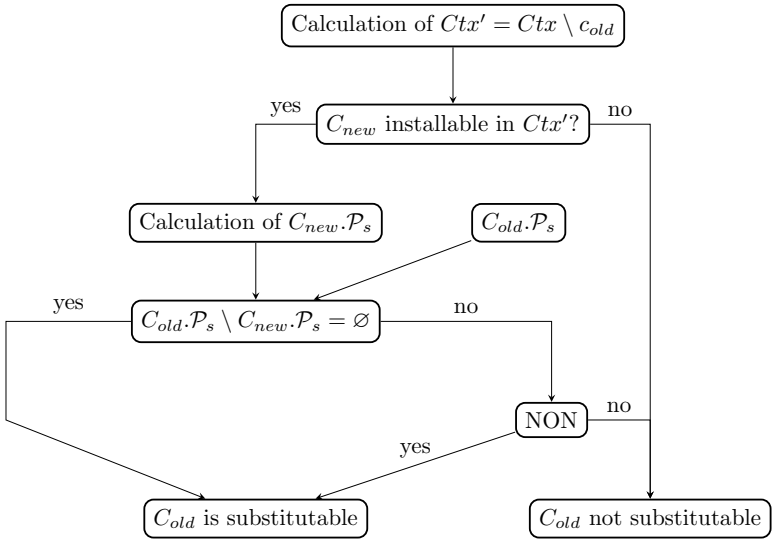


Fig. 5. Substitutability phases

Ctx without the set of all provided services of C_{old} and without its forbidden services and forbidden components. Then, we check whether the new component can be installed using installability rules in the new context, i.e., all C_{new} requirements are fulfilled in Ctx' and its provided services does not conflict with Ctx' (the rules are described in [2]). Once the installation of the new component is possible in the new context, we calculate the effect of its installation in the context from its dependency description using installation rules described in [2], i.e., its provided services, forbidden services, forbidden component and the new dependency graph (it is illustrated in the diagram of the Fig. 5 by the calculation of $C_{new}.\mathcal{P}_s$).

Since the set of provided services depends on the availability of services in the context, we need to use installation rules to calculate it. The two main phases of substitutability are the installability and the calculation of provided services (installation) which depend on the context description and the component dependency description ($D_1 \bullet D_2, D_1 \sharp D_2$, or $?D$). The calculation of provided services is not obvious without evaluating the dependency description in the context. Even if the component is installable the provided services depend on fulfilled dependency conditions in the context. Therefore, we present the calculation of provided services depending on the dependency descriptions ($D_1 \bullet D_2, D_1 \sharp D_2$, or $?D$):

- For $D_1 \bullet D_2$, D_1 and D_2 must be verified in Ctx', and the set of provided services is the union of the provided services of D_1 and those of D_2. For example, considering the following description: $((C1.S_1 \Rightarrow S_2) \bullet (S_3 \wedge [FDS \geq 10]) \Rightarrow S_4)$, the installability conditions are :

· S_1 belongs to the set of provided services of C_1 and S_2 is not forbidden in the context **and**
· S_3 belongs to the set of available services of the context, the condition $[FDS \geq 10]$ is verified and S_4 is not forbidden in the context.

Thus, the provided services are $C.\mathcal{P}_s = \{S_2, S_4\}$ or $\varnothing$

- $D_1 \sharp D_2$ is verified if D_1 is verified in Ctx or D_2 is verified in Ctx. For example, for: $((C1.S_1 \Rightarrow S_{text}) \sharp (S_3 \wedge [FDS \geq 10]) \Rightarrow S_{graph})$, the installability conditions are :
  · S_1 belongs to the set of provided services of C_1 and S_{text} is not forbidden in the context `else`
  · S_3 belongs to the set of available services of the context, the condition $[FDS \geq 10]$ is verified and S_{graph} is not forbidden in the context.

  The set of provided services is $C.\mathcal{P}_s = \{S_{text}\}$ `else` $\{S_{graph}\}$ `else` $\varnothing$
- $?D$ is always installable, for example: $?(C1.S_1 \Rightarrow S_2)$, the set of provided services is $C.\mathcal{P}_s = \{S_2\}$ `if` $S_1 \in C_1.\mathcal{P}_s$ and S_2 is not forbidden `else` $\varnothing$

Next, we compare the provided services of C_{old} with those of C_{new}. C_{old} is substitutable in two cases: either the set of previously provided services which are no longer provided by C_{new} is empty or each of these services are not necessarily used by other components in the context.

5 Example

Let us illustrate component substitutability by examining two substitutability scenarios. First, the new component provides fewer services. Second, the new component provides more services. The first case may happen for optimizing purposes by replacing one component by another which requires fewer resources and provides fewer services. For example, we replace `THUNDERBIRD` by `SYLPHEED`. We suppose that the service `THUNDERBIRD`.*corrector* (a spell checker) is the only service which is not provided by `SYLPHEED`. The system must ensure the condition NON i.e., this service is not used by another component. According to the dependency graph of the mail server represented in Fig. 2 of section 2, the service `THUNDERBIRD`.*corrector* is a leaf in the graph. Thus, `THUNDERBIRD`.*corrector* is not used by another component and NON condition is ensured. The component `THUNDERBIRD` can be substituted by `SYLPHEED` which provides fewer services if `SYLPHEED` is installable, i.e., its required services are available in the context. The required services here are the libraries (`C6`.*lib*6) which means the libraries *lib*6 provided by any component, for example `C6` (see Fig. 6).

The second example addresses the substitution of a component providing

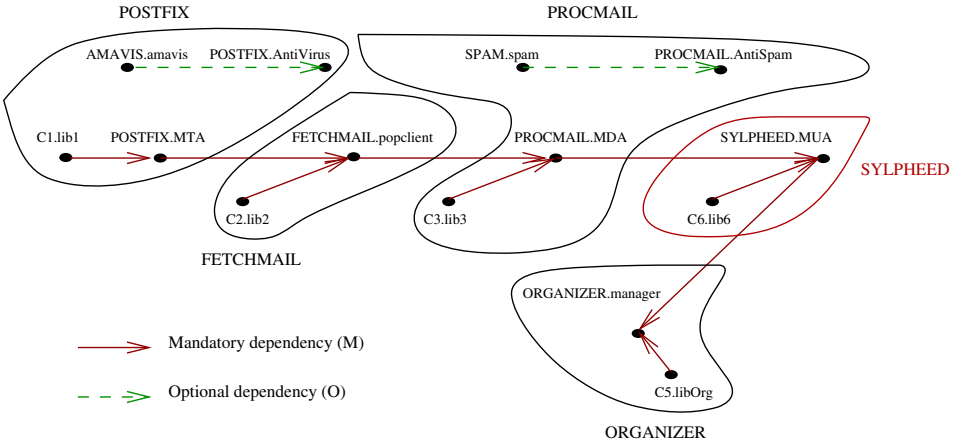


Fig. 6. Substitution of `THUNDERBIRD` by `SYLPHEED`

more services. For instance, replacing `THUNDERBIRD` with the mail user agent of `SEAMONKEY` which has numerous enhancements, for example: a Chat service. (see Fig. 7).

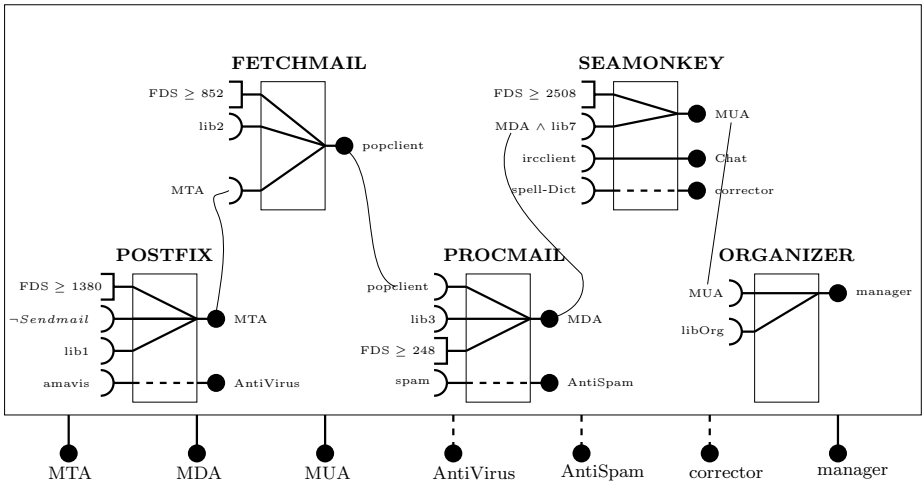


Fig. 7. The mail server with `SEAMONKEY`

Checking substitutability corresponds to ensuring the requirements of `SEAMONKEY` (`C7`.*lib7*, `IRC`.*ircclient*, `SPELL`.*spell* − *Dict*, etc.) which are different from those of `THUNDERBIRD` and ensuring that the additional provided services do not disrupt the context. The related dependency graph is presented in Fig. 8.

Now, let's illustrate our main contribution, the two substitution examples of `THUNDERBIRD` presented above may not be possible even if conditions on provided services are fulfilled ($C_{old}.\mathcal{P}_s \setminus C_{new}.\mathcal{P}_s = \varnothing$ and NON). Indeed,

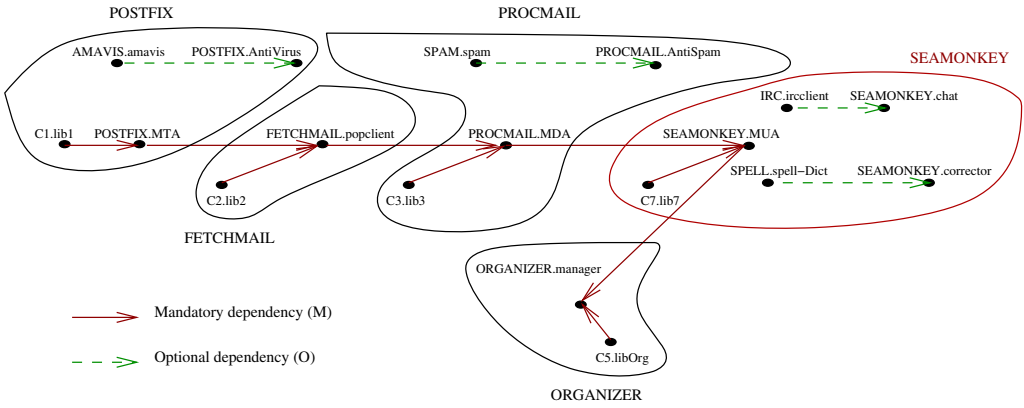


Fig. 8. Substitution of `THUNDERBIRD` by `SEAMONKEY`

according to the diagram of Fig. 5, we have to verify firstly the installability conditions of `SEAMONKEY` and `SYLPHEED`. One of the most important part in the installability condition is the verification of the **effect** of the component in the context. If we suppose that the service *chat* is forbidden in the context, then the condition of the installability of `SEAMONKEY` is not verified. Therefore the substitution is not possible before comparing provided services of each component.

Finally, we can have invariants in the context that we need to preserve. For example, we may have to preserve the security level of a system by forbidding the installation of any service that can decrease the security level of the system (like the service `ftp` for example). Let's suppose a component C_{new} which provides the service `ftp` and someone want to replace component C_{old} by C_{new}. The fact that the service `ftp` decreases the security level will forbid the replacement of C_{old} by C_{new} even if it provides all necessary services and does not require too much.

6 Related work

The issue of component substitutability has already been addressed in literature. We mention here only those which are the closest to our work and summarize the most common approaches dealing with the substitutability problem.

In oriented object programming, the substitution principle of Liskov is a particular definition of subtype which was introduced in [1]. This concept of subtype is founded on the concept of substitutability, i.e., if S is a subtype of T then we can substitute objects of the type S for objects of the type T without deteriorating the desirable properties of a program. However, although the

concept of subtype is approved in [10], the rules based on typing are rather restrictive. We therefore choose a more flexible approach that allows us to make choices according to the system and user requirements.

Substitutability is presented in [9] as a relation between the types of the old and the new components. This approach is based on a contract definition which is a consequence of the definition of the component type [8,7]. Substitutability is based on the definition of the compatibility between the component and all the elements with which it interacts. Compatibility is defined in terms of syntactic, semantic, and pragmatic contracts for operations, interfaces, ports and components. Thus, a component A is substitutable for another component B if its compatibility with other components is preserved after the substitution and the new required properties are checked. Our work follows the same principle but it is done on the interface signature only.

The concept of substitutability in [5] is defined for black box components. The principle is to check that the substitution of the component preserves the consistency of the preliminary configuration. The concepts of context of deployment, strict substitutability and contextual substitutability are defined. The representations of component specifications and the deployment context are based on the ENT model (*Export*, *Needs*, *Ties*). The definition of strict substitution is different from ours because it considers that the new component must provide at least the same thing and requires at most the same thing as the old component (generalization of the needs and specialization of the requirements). In the case of "strict" substitutability there is no check for new required interfaces since those are not supposed to exist and they are "forbidden" by the strict subtyping case. Nevertheless, new provided interfaces are allowed and therefore checked. However, in our work, the verification is done for the new requirements as well as for the new provided services without using subtyping rules. We think that the strict substitutability is not really interesting because the component needed functionalities depend on the environment in which it is used and it does not verify the effects on the context and its invariants.

Despite enhancements in substitutability specification at signature, semantics and protocol levels, we believe that these works do not take into account the **effect** of the component. Indeed, they do not verify potential conflicts that can occur after substitution, due to new services. Therefore, in our approach we impose more constraints on the new provided services and ensure that they do not conflict with the existing context and its *invariants* are preserved. For example, when we want to substitute a component which provides `http` with another providing `http` and `ftp` services and the system forbid non-secure services like `ftp`, such a substitution cannot occur. Furthermore,

taking into account the potential effect of the new component on the system can be generalized. It is applied not only to conflicts but also to other kinds of effect. For example, the substitutability verification of non-functional properties needs such a mechanisms as the new component may conflicts with the system invariants. Another example of use is the resource consumption. The new component despite being functionally equivalent may not work because it consumes too much resources for the system. For this reason, we have to control the effect of the new component on the context.

7 Conclusion and Future work

In this work, we have presented a formalization of component substitutability. Our formalization is based on dependencies and context descriptions which are also used for installation and deinstallation phases in [2]. It aims at providing a safe and flexible component upgrade. The key concept is the comparison between dependency descriptions of the new and the old component. The comparison concerns provided services and does not take into account required services. We have defined the *strict* and the *contextual* substitutability and we have concentrated only on *contextual* one. We have presented an analysis of different substitutability cases and summarized them into three key conditions. These conditions involve checking the installability rule of the new component (verifying requirement and ensuring that provided services will not conflict with the context of the system) and checking the effect of deinstallation of the old component using the dependency graph. A prototype implementing our proposal has been developed in Ocaml. Our objective is to ensure the safety of substitutability without being restrictive by authorizing all cases of substitutability. For example, replacing a component which has a lot of unused services with another which has fewer provided services (only those which are useful) is possible. This substitutability can also depends on a system policy or property models as described in [3]. We focus on ensuring the **safety** of the system, i.e., verifying the *requirements*, the *effect* of the substitution and preserving context *invariants*, component and service *properties*.

Now, we aim to parametrize the substitutability check by policies and properties (for example, if the policy tries to optimize resources we cannot replace a component by another one which requires a lot of resources). Furthermore, we are working on a dependency description extension to express properties on services and components. As future work we aim at extending our system to overcome its two main limitations, which are:

- to substitute a component assembly, we have to calculate the dependency of a composite component using the dependencies of its sub-component.

- in our current approach, components and services are identified by their names. This identity must be extended to include interface type and property information. This means changing from name equivalence to a form of subtyping when determining dependencies between services. In such an approach, we could reuse behavioral substitutability such as [6] for example.

References

[1] B. Liskov and J. Wing. Behavioral subtyping using invariants and constraints. MU CS-99-156, School of Computer Science, Carnegie Mellon University, july 1999.

[2] M. Belguidoum and F. Dagnat. Dependency management in software component deployment. In *FACS'06-International Workshop on Formal Aspects of Component Software*, Prague, Czech Republic, September 2006. ENTCS.

[3] M. Belguidoum and F. Dagnat. Dependability in software component deployment. In *DepCoS-RELCOMEX*, pages 223–230, Szklarska Poreba, Poland, June 2007. IEEE Computer Society.

[4] A. Beugnard, J. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Lecture Notes in Computer Science*, 32(7):38–45, July 1999.

[5] P. Brada. *Specification-Based Component Substitutability and Revision Identification*. PhD thesis, Charles University, Prague, August 2003.

[6] I. Cerna, P. Varekova, and B. Zimmerova. Component substituability via equivalencies of component-interaction automata. In *FACS'06-International Workshop on Formal Aspects of Component Software*, Prague, Czech Republic, September 2006. ENTCS.

[7] P. Champagnoux, Laurence Duchien, D. Enselme, and G. Florin. Cooperative abstract data type : A stack exemple. In T. Hruska and M. Hashimoto, editors, *IEEE 4th Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2000)*, pages 183–190, Brno, Czech Republic, sep 2000. IOP Press.

[8] P. Champagnoux, Laurence Duchien, D. Enselme, and G. Florin. Typage pour des composants coopératifs. In *Colloque International sur les NOuvelles TEchnologies de la REpartition, NOTERE 2000*, Paris, France, nov 2000. Revue EJNDP RERIR, Revue Eléctronique sur les Réseaux et l2019Informatique Répartie.

[9] F. Legond-Aubry, D. Enselme, and Gerard Florin. Assembling contracts for components. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS-DAIS)*, Lecture Notes in Computer Science, pages 35–43, Paris, France, November 2003. Springer-Verlag.

[10] J. Costa Seco and L. Caires. A basic model of typed components. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128, London, UK, 2000. Springer-Verlag.

[11] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[12] A. Vallecillo, J. Hernandez, and J. Troya. Component interoperability. Technical Report ITI-2000-37, Departmento de Lenguajes y Ciencias de la Computacion, University of Malaga, 2000. Available at http://www.lcc.uma.es/ãv/Publicaciones/00/Interoperability.pdf.

[13] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.