Toward a rewriting logic framework for safe and distributed component installation

Meriem Belguidoum
LIRE Laboratory,
Computer Science Department,
University Mentouri
Constantine, Algeria
TELECOM Bretagne,
Computer science department
Brest, France
meriem.belguidoum@telecom-bretagne.eu

Faiza Belala
LIRE Laboratory,
Computer Science Department,
University Mentouri
Constantine, Algeria
belalafaiza@hotmail.com

Fateh Latreche
LIRE Laboratory,
Computer Science Department,
University Mentouri
Constantine, Algeria
fateh_lat@yahoo.fr

Traditionally, software components are developed independently and often considered as black boxes. However, they might be dependent from each other at assembly and deployment stages and used by a third party in various environments. In a previous work (Belguidoum and Dagnat 2007), we have proposed an intra-dependency language and a predicate logic based framework for safe component deployment. In this paper, we aim to extend this work to take into account dynamic changes of concurrent and distributed systems in installation phase and overcome some predicate logic limitations. Therefore, we propose an integration of predicate logic based framework within a rewriting logic based framework. Finally, we execute it in Maude which is a high-performance implementation of rewriting logic enabling both the deployment execution and its formal analysis. We show the relevance of the Maude based deployment approach relating to the interesting features of Maude such as: genericity, concurrency, distribution and formal tools.

Component deployment, Formal specification, Rewriting logic, Maude

1. INTRODUCTION

Nowadays, software are used on heterogeneous terminals and in different and complex communication infrastructures. The best way to develop and deploy this complex software with good properties in terms of flexibility, reliability and scalability is the Component-Based Software Engineering (CBSE) (Grunske et al. 2010). One of the main contributions of CBSE is the reuse of software components to save out development effort. In this context, software is defined as components assembly in an architecture, by binding a required interface of one component to an offered interface of another component. Thus, a software component is considered as a unit of composition with contractually specified interfaces and explicit dependencies (Szyperski 1998). Since, software deployment is a collection of dependent and complex activities that form the component software deployment life cycle (Carzaniga et al. 1998) (from its development to its deinstallation), these dependencies must be handled in a convenient way (Vieira and Richardson 2002).

A variety of tools and techniques that support different activities of the deployment process have been introduced (Heydarnoori 2008; Parrish et al. 2001). Currently there is no deployment tool or technique that can guarantee the success and the safety of all deployment activities. For instance, installation may not be *successful* if an installed component does not work properly and it may also not be *safe* if it disrupts the proper functioning of the existing system.

To face the evolution towards component based systems, our aim is to build a tool with formal foundations ensuring the success and safety of deployment. This is chiefly due to the lack of support for formally specifying and verifying component software deployment activities. To address this requirement, some works are achieved. In (Liu and Smith 2006), authors define a formal Labeled Transition System (LTS) for the application target system (the place where software is developed and deployed) with transitions for deployment operations and establish formal properties of the LTS, including the fact that if a component is

shipped with a certain version dependency, then at run time that dependency must be satisfied with a compatible version. In (Mouakher et al. 2006) dependencies between a required and a provided interfaces are guaranteed through an adapter by the use of the B formal method with its underlying concept of refinement, and its powerful tool support, the B prover. However, checking the deployment dependencies in a distributed and concurrent configuration is note taken into account in these works.

Consequently, when studying these proposed works, several challenges need to be addressed at the same time to ensure a correct deployment in concurrent and distributed component based systems. The most important challenges are: (1) component dependencies are not *explicitly* and *precisely* described and their management becomes a very complex task, (2) the *success* and the *safety* of deployment operations have to be checked in a formal and rigorous way, (3) component should be dynamically adapted depending on the user context and (4) the *concurrency* and the *distribution* of deployment have to be considered in component software deployment.

The two first challenges are addressed in a previous work (Belguidoum and Dagnat 2007), indeed, we have proposed a formal description of component dependencies specifying the precise relation between each provided service and its requirement in the same component. We proposed a predicate logic based framework for component deployment to ensure the success and the safety of component installation, deinstallation (Belguidoum and Dagnat 2007) and substitution (Belguidoum and Dagnat 2008). However, this framework has some limitations when applied to dynamic, concurrent and distributed systems. In fact, the predicate logic imposes a centralized interleaving semantics of concurrent computations, hence the two last challenges can not be addressed.

In this paper, we aim to propose a complementary approach to take into account dynamic changes of concurrent and distributed systems in installation phase. Therefore, we use the rewriting logic (Martí-Oliet and Meseguer 2002) which has been shown as an unified semantic framework, for concurrent and distributed computation (Martí-Oliet and Meseguer 2000; Meseguer 2004; Meseguer and Roşu 2007). The basic axioms of this logic, which are rewrite rules of the form $t \rightarrow t'$ where t and t' are terms over a given signature, can be interpreted into two ways (Aiguier et al. 2006): (1) as the local transition of a concurrent system, the rewriting logic then extends (equational) algebraic specifications to deal with dynamic and concurrent changes in systems or (2) as the inference rule, the rewriting logic is then a *universal* logic within which other logics can be translated. Furthermore rewriting logic has several high-performance implementations (Borovanský et al. 2002; Clavel et al. 2011; Diaconescu and Futatsugi 1998), the most comprehensive and expressive one is Maude (Clavel et al. 2011). Indeed, it is an efficient tool for both execution and formal analysis (a semi-decision procedure, an LTL model checker, a theorem prover, etc.). Consequently, to deploy efficiently software component in a concurrent and dynamic way, we propose an integration of predicate logic based framework (Belguidoum and Dagnat 2007) within rewriting logic and execute it in Maude which is a high-performance implementation of rewriting logic enabling both the deployment execution and its formal analysis. Indeed, Maude rewriting engine is highly optimized and can perform millions of rewrite steps per second, while its model checker is comparable with SPIN (Eker et al. 2002).

The rest of this paper is structured as follows. In section 2, we present the basic concepts of rewriting logic and Maude. In section 3, we present our transcription approach from predicate logic based models to Maude one, while recalling the formalization principle adopted to describe the component installation. The proposed Maude modules are successfully executed and illustrated in section 4 through an example to show how deployment can be easily specified and implemented in a concurrent way using Maude. Section 5 evaluates the Maude specification of component installation which overcomes the limitations of predicate logic based framework and presents their complementarity. Finally, section 6 concludes the paper and discusses future work.

2. REWRITING LOGIC AND MAUDE

Rewriting logic is a promising logical and semantic framework in which many different logics and models can be naturally represented and interrelated (Martí-Oliet and Meseguer 2000). Using a rewriting logic implementation such models can then be used to generate a wide range of formal tools (Meseguer 2004).

2.1. Basic concepts

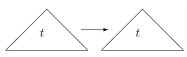
In rewriting logic, a dynamic system is represented by a rewriting theory, describing the complex structure of its states and the various possible transitions between them. A rewriting theory is defined by $R = (\Sigma, E, L, \mathcal{R})$, where (Σ, E) is an equational membership theory, L is a set of labels

and $\mathcal{R}$ is a set of labeled conditional rewriting rules of the following form:

$$r : (\forall X) t \rightarrow t' \; if \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \rightarrow t'_l$$

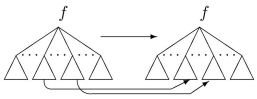
Where r is a labeled rule, all terms $(p_i, q_i, w_j, s_j, t_l, t'_l)$ are Σ-terms belonging to $T_{\Sigma,E}$ and the conditions can be rewriting rules, membership equations in (Σ, E), or any combination of bothN i and j are subsets of natural numbers $\mathbb{N}$. As a consequence, the relevant sentences that may or may not be provable by the above theory R are sequents of the form: $t \rightarrow t'$. The provable sentences are exactly those derivable by the inference rules presented bellow:

Reflexivity : for any state t there is an idle transition in which nothing changes



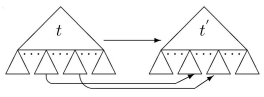
for each $[t] \in T_{\Sigma,E}(X),\ [t] \rightarrow [t]$

Congruence : each operator f can be seen as a parallel state constructor, allowing its non-frozen arguments to evolve in parallel



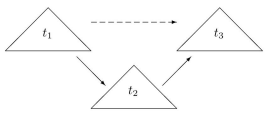
$$\text{for each } f \in \Sigma_n, n \in N, \quad \frac{[t_1] \rightarrow [t'_1] \ldots [t_n] \rightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \rightarrow [f(t'_1, \ldots, t'_n)]}$$

Replacement : supports a different form of parallelism, the state fragments in the substitution of the rules variables can also be rewritten



$$\text{for each rewrite rule } r : [t(x_1, \ldots, x_n)] \rightarrow [t'(x_1, \ldots, x_n)] \; in \; R, \quad \frac{[w_1] \rightarrow [w'_1] \ldots [w_n] \rightarrow [w'_n]}{[t(\bar{w}/\bar{x})] \rightarrow [t'(\bar{w'}/\bar{x})]}$$

Transitivity : build longer concurrent computations by composing them sequentially



$$\frac{[t_1] \rightarrow [t_2] \;\; [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

Computationally, the provable sequents describe all the complex concurrent transitions of the system axiomatized by R. Logically, they describe all the possible complex deductions from one formula to another in the logic axiomatized by R. Moreover, rewriting logic has a theory model with natural computational and logical interpretations (Martí-Oliet and Meseguer 2002) in addition to its inference system.

2.2. Maude system

Theoretical concepts of rewriting logic are implemented through Maude language, a high-performance declarative language, supporting both equational and rewriting logic specification of concurrent systems. It has been influenced by the OBJ3 equational logic language. Besides, Maude has been also used to develop, program and prototype a wide range of applications. Indeed, it offers a comprehensive toolkit for the analysis of specifications, such as LTL model checker, Inductive Theorem Prover (ITP), Maude Termination Tool, Church Rosser Checker, Coherence Checker, etc (Clavel et al. 2011).

The basic units of specification and programming in Maude are *modules*. A program in Maude supports, three types of modules mainly: *functional* modules to define the static aspects of a system, they form a Maude sub-language (extension of OBJ3) based on the equational logic; *system* modules specify the dynamic aspect of the system using rewriting rules; while *object oriented* modules specify the object oriented systems.

Functional and system modules contain a common signature declaration consisting of:

- *sorts*, giving names for data types,
- *subsorts*, organizing the data types hierarchically,
- operators, providing names for the operations that will act upon the data and allowing us to build expressions (or terms) referring to such data.

The difference between them resides in their manipulating statements:

- *functional module*: is an equational-style functional program with user-defined syntax in which a number of sorts, their elements, and functions are defined. From a specification viewpoint, a functional module is an equational theory (Σ, E) with initial algebra semantics.

- *system module*: is a declarative-style concurrent program with user-defined syntax in which rules, transitions between states, equations and memberships are defined. From a specification viewpoint, it is a rewrite theory $(\Sigma, E, \phi, \mathcal{R})$ (where ϕ specifies the frozen arguments[1] (Clavel et al. 2011) of operators in Σ) with initial model semantic. We do not present the *object oriented module* which is a special case of *system module* since it is not used in the context of this paper.

There are two main framework in Maude system: (1) the Core Maude interpreter implemented in C++ and providing all Maude basic functionalities and (2) the Full Maude, an extension of Maude, written in Maude itself, that endows the language with an even more powerful and extensible module algebra. More details of Maude can be found in (Clavel et al. 2011).

3. A FORMAL APPROACH FOR COMPONENT INSTALLATION

In a previous work we have proposed a formal deployment framework to ensure the success and the safety of the software system, we have been interested by the component installation formalization (Belguidoum and Dagnat 2007, 2008). Thus, component dependencies are specified in predicate logic based language. These dependencies are verified according to the target system specification using the sequential derivation of the sequent calculus. Our dependency description approach is based on the parameterized contracts (Reussner 2001) presented as a generalization of interoperability checks between components. It allows to perform adaptation of component provided or required interfaces. A component will offer less functionalities if its environment offers not all functionalities the component requires, and a component will require less functionalities, if not all offered functionalities will be used by its clients.

In this paper, we focus on how to translate this formal specification of component and its deployment operation into a more suitable executable specification model for a generic solution. Therefore, we have chosen Maude language, a high performance rewriting logic implementation. This contribution is intended to prototype and execute our predicate logic based model, using Maude system and its analysis tools (LTL model checker, etc.). Besides, it will extend the previous formalization to take into account dynamic and concurrent changes in considered systems.

[1]declaring certain argument positions in an operator with the frozen attribute blocks rule rewriting anywhere in the subterms at those positions.

$$D ::= P \Rightarrow s \mid D \bullet D \mid D \# D \mid ?\, D$$

$$P ::= true \mid P \wedge P \mid Q$$

$$Q ::= Q \vee Q \mid [v\ O\ val] \mid \neg\, s \mid \neg\, c \mid c.s \mid s$$

$$O ::= > \mid \geq \mid < \mid \leq \mid = \mid \neq$$

Figure 1: *The intra-dependency grammar*

3.1. Intra-dependencies Maude description

Intra-dependencies are considered as parametrized contracts (Reussner 2001), they represent the relation between each provided service (s) and its requirements (P) in the same component. They are defined in the grammar of Figure 1, where D denotes a dependency, P a predicate (requirements), s a service and c a component.

As it is described in (Belguidoum and Dagnat 2007) intra-dependencies can express a simple dependency ($P \Rightarrow s$) or a composed one ($D \bullet D$, $D \# D$, $?\, D$): $P \Rightarrow s$ is the couple of preconditions (P) representing the predicate and a post-condition (s) representing a potential provided service, $D_1 \bullet D_2$ (respectively $D_1 \# D_2$) is the conjunction (respectively the disjunction) of the dependencies D_1 and D_2 and $?\, D$ is an optional dependency.

This work result gives good example of formal specification of component intra-dependencies and formal verification of its installation, but the obtained model can not be directly executed nor checked using a formal tool. Our approach here consists in implementing the proposed predicate logic based model for component installation in Maude language to benefit from its formal execution and formal analysis tools. For this reason, we identify for each installation basic concepts involved in predicate logic semantic framework, its executable Maude module that inherits from the rewriting logic all its theoretical aspects. Consequently, the deployment that was described statically, may consider in the Maude based framework the dynamic features of the environment and the concurrent changes of the application. Additionally, proofs of the *success* and the *safety* of this operation (the deployment) are simply and naturally deduced, we use directly the model checker tool of Maude system, we do not need to learn how to use tools of automatic proof that often use quite complex languages and requires enough time to manipulate them.

In order to give an algebraic semantics to component intra-dependencies, we first define the corresponding syntax in Maude for every part in

Basic installation entities	Predicate logic specification	Maude specification
a component	a proposition c and a dependency D	a term of sort component in a functional module COMPONENT
a service	a proposition s	functional module: SERVICE
an intra-dependency	dependency language D with its predicates P	functional modules: • INTRA-DEPEND-LANGAGE (see Listing 1) • PREDICATES • SIMPLE-PREDICATE
a target system	Context properties: • its environment properties (hardware and software)$\mathcal{E}$ • the set of its components $\mathcal{C}$ • the dependency graph $\mathcal{G}$	functional module: CONTEXT (see Listing 2)

Table 1: *Mapping between Maude and predicate logic specifications*

their generating grammar (see table 1) (component, service, intra dependency and target system). Then, we define dynamic semantics for the component installation operation. Components and services are specified here as algebraic terms having sorts Components and Services respectively.

3.1.1. Intra-dependency description

In the INTRA-DEPEND-LANGAGE module, we define terms of sort Dependencies explaining how component intra-dependencies are described in terms of predicates. It provides specification of all possible notations of dependency thanks to its declared operators: _orD_ to specify disjunction of dependencies, _andD_ for the conjunction of dependencies, etc. (see Listing 1).

Listing 1: Maude intra-dependency specification

```
fmod INTRA-DEPEND-LANGAGE is
protecting PREDICATES .
sorts Dependency Dependencies .
subsort Dependency < Dependencies .
op _=>_ : Predicates  Service ->
          Dependency [ctor prec 29] .
op _andD_ : Dependencies Dependencies ->
          Dependencies [ctor comm assoc prec 30] .
op _orD_ : Dependencies Dependencies ->
          Dependencies [ctor comm assoc prec 30] .
op  ?_  : Dependencies ->
                    Dependencies [ctor prec 30] .
endfm
```

We note that this module imports the module PREDICATES which specifies semantic of any simple predicate term (SimplPredicate) formed thanks to the following operators: notS_ for forbidden service, notC_ for forbidden component, _\/_ the disjunction of simple predicates. Since the predicate grammar (see Figure1) follows a CNF (*Conjunctive Normal Form*) notation, we use a super sort Predicates to form a predicate conjunctions first ($\_ \wedge \_$) then we use the following operators:

```
    op notS_ : Service ->  SimplPredicate .
    op notC_ : Component ->  SimplPredicate .
    op _._ : Component Service ->  SimplPredicate .
    op _\/_ : SimplPredicate SimplPredicate ->
 SimplPredicate [ctor assoc comm id: nilspprec 27] .
```

We note that this module imports also SERVICE and COMPONENT modules, since predicates or dependencies are related to these entities.

Semantical equivalence between components may be established on their corresponding dependencies thanks to some additional equations.

3.1.2. Target system description

The description of the target system is called a *context*. A context is composed of (1) its environment properties (hardware and software) denoted by $\mathcal{E}$ or $ctx.\mathcal{E}$, (2) the set of its components denoted by $\mathcal{C}$ storing for each installed component c its provided services $\mathcal{P}_s$, forbidden services $\mathcal{F}_s$ and forbidden components $\mathcal{F}_c$ $(c, \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c)$, (3) the dependency graph storing dependencies between these components denoted by $ctx.\mathcal{G}$. A node of $\mathcal{G}$ is an available service s and its provider c $(c.s)$ 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.

The target system description is given in Maude thanks to the ctx(_,_,_) operator declared in the functional module of Listing 2. The triplet: (1) system environment (ENV-VARS), (2) the set of installed components (INSTALLED-COMPONENTS) and (3) the dependency graph (DEP-GRAPH), represents the

Component installation	Predicate logic based framework	Maude based framework
installability phase	first order logic installability rules	The rewriting rule (`rl [install]`) in the system module : `CONFIG-INSTALL`
installation phase	first order logic installation rules	

Table 2: *Component installation in Maude*

Listing 2: Maude component context specification

```
fmod CONTEXT is
protecting ENV-VARS .
protecting INSTALLED-COMPONENTS .
protecting DEP-GRAPH .
sort Context .
op ctx‘(_‘,_‘,_‘) :  EnvVars InstalledCs Graph
                        -> Context [ctor prec 30 ] .
op evalDep : Dependencies Context -> Bool .
op eval : Predicates Context -> Bool .
...
endfm
```

Listing 3: Maude component installation specification

```
mod CONFIG-INSTALL is
protecting CONTEXT DECLAR-COMPONENT.
sort Declaration .
sort Configuration .
subsort Context Declaration < Configuration .
op _:_ : Component Dependencies ->
                    Declaration [ctor prec 31 ] .
op modify : Graph Component Dependencies Services
                                          -> Graph .
var C : Component .
var E : EnvVars .
var I : InstalledCs .
var G : Graph .
var X : Context .
...
rl [install] : ctx( E , I , G ) C : D =>
      if  ( not ( C BelongsTo ForbiddenC(I)) and
            evalDep( D , ctx( E , I , G ) )) then
        ctx( E , < C , PSDep( D, ctx( E,I,G )),
        FSDep(D,ctx(E,I,G)),FCDep(D ,ctx(E,I,G))> I,
        modify(G,C,D,PSDep(D,ctx(E,I,G))))
      else ctx(E,I,G ) fi  .
endm
```

actual context while specifying the context structure as it was done in predicate logic based approach. Each of the triplet element is specified separately in a Maude module. Terms of sort `Context` respecting the underlined signature define the actual environment in which components will be installed. Then, the set of installability rules presented in (Belguidoum and Dagnat 2007) are implemented in Maude using two particular operators declared in this module: `eval` to evaluate the predicates and `evalDep` to evaluate the dependencies. The verification process of these rules is specified by a set of equations.

3.2. Component installation formalization in Maude

In general, the installation process is divided into two phases: a first one ensures component *installability* (the possibility of component installation) and a second phase calculates the *result* of the component installation in the target system (see table 2) by providing and/or forbidding new services (or new components).

The table 2 describes the mapping between Maude based installation description and predicate logic based installation description. We note that in the Maude based framework, the effect of the two rule types is defined by a single rewriting rule: `install`. This local transition rule is applied to any component and context couple, and shows in its right hand side the achieved changes in the considered context (see Listing 3). By the same way, the concurrent application of this rewriting rule specifies the possible concurrent and distributed component installation.

Dynamic semantics of component installation operation is given through the two main modules: `CONFIG-INSTALL` and `DECLAR-COMPONENT`. The former (`CONFIG-INSTALL`) extends the previous functional modules in order to describe the state change of a system (`configuration`) consisting of a given context (`Context`) and a component (`Declaration`) that will be installed in this context. The Component declaration is obtained using the following operator :

`op _:_ : Component Dependencies -> Declaration .`

The `Configuration` sort represents a multi set of declared components and predisposed contexts (`Context Declaration < Configuration`). This multiset allows us to define distributed structure of considered components as follows:

`__ : Configuration Configuration -> Configuration .`

Now, we are able to define not only the installation of one component in a given context but also the installation of a set of components using a meta rewrite rules of the form: `crl: init-config -> final-config if cond . init-config` and `fin-config` are algebraic terms of sort Configuration that may design several components and some predisposed contexts.

The rewrite rule `install` is defined to give the semantics of the execution of installation operation. When the declared component does not belong to the forbidden components set, the installation operation is carried out and it consists in updating: the set of component (adding the component `C`, its provided services `PSDep`, its forbidden services

PSDep, its forbidden components FCDep) and the dependency graph G using the operator modify which calculates the subgraph of the component C by binding its requirements (already available in the context graph) with its new provided services.

In the DECLAR-COMPONENT Maude module, we specify the actual relation between a given installed component and its context. For instance, we define the PSDep operator and some corresponding equations to get all provided services of a given component C. Other similar operators FCDep, FSDep, RSDep, etc. are defined to get respectively Forbidden services, Forbidden components, Required services, etc. Thus, the programmer of such component installation operation has this model in mind. So, the essential asset of this logic is the so-called agreement between the mathematical semantics (the models) and the operational semantics (the computations). For this semantic level, the mathematical category model, inherited from rewriting theories (system modules for the Maude language), associates a precise definition in terms of mathematic morphisms to system state changes and algebraic terms to predicate static concepts. The state transitions, expressing concurrent components installation, are the (possibly complex) concurrent rewrites possible in the system given by application of the local rules in CONFIG-INSTALL and rewriting logic deduction rules.

4. A COMPONENT INSTALLATION EXAMPLE

Let us take the example of POSTFIX (Postfix 2009) an SMTP server playing the role of a Mail Transport Agent (MTA). The Figure 2 illustrates the installation principle of POSTFIX. To be able to install this component, we have to make available its requirements (dependencies). POSTFIX has mandatory dependencies (Free Disk Space (FDS), libraries (S_{lib}), conflicting component (SENDMAIL: C_{SM}) and optional dependencies (anti-virus Amavis: S_{Amavis}). The detailed description of POSTFIX and its installation proof according to predicate logic based framework are given in (Belguidoum and Dagnat 2007).

In table 3, we present intra-dependency of POSTFIX and the target system in predicate logic based description and in their corresponding Maude code.

In the predicate logic based framework, the installation is carried out in two stages. First we check whether an installation is possible (*installability*) by evaluating the component dependency in the current context. Then if an installation is possible, we calculate its effect on the context. This effect is used to update the context once the concrete installation has been carried out.

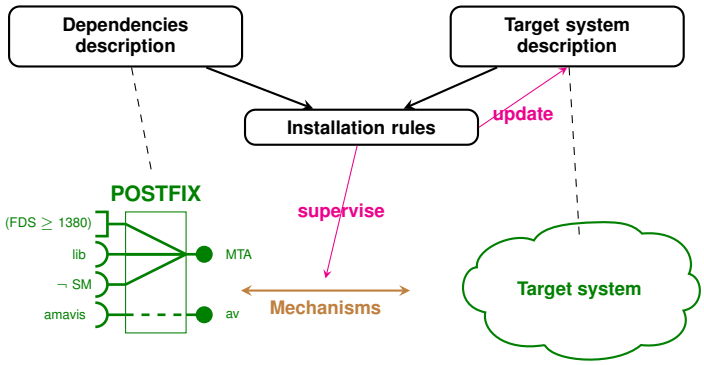


Figure 2: *Installation principle of POSTFIX component*

The installability of POSTFIX is deduced by the proof presented in Figure 3. This proof represents a set of inference rules denoted by the derivation symbols: $\vdash_C$ for checking dependencies and $\vdash_P$ for evaluating predicates. Initially, the rule CAND is used to check D_1 and D_2. Then, the rule CTRIV is used to check D_1 and verifies that its provided service S_{MTA} does not belong to the set of forbidden services of the context ($FS(Ctx) = \varnothing$), the rule PTRIV is used to evaluate its predicate $[FDS \geq 1380] \wedge \neg C_{SM} \wedge S_{lib}$. Finally, the rule PVAR checks the availability of FDS in the context (the free disk space is greater or equal to the required one), the rule PNOTC verifies that the conflicting component C_{SM} does not belong to the available components of the context ($AC(Ctx)$) and the rule PSERV checks the availability of S_{lib} in the set of available services of the context ($AS(Ctx)$). Note that as D_2 is optional, it is not explored in this stage.

As POSTFIX is installable, the installation stage calculates the effect of installing POSTFIX (C_{PX}) according to some installation rules: IAND3, IOPT$_2$ and ITRIV using the derivation symbol $\vdash_I$ to evaluate the effect of installation and the rules $GAnd$ and $GServ$ using the derivation symbol $\vdash_G$ to calculate the resulting dependency graph. The installation proof is presented in Figure 4 (the requirement of S_{MTA} is denoted by the predicate P). Initially, the rule $IAnd3$ is used to calculate the effect of installation of POSTFIX by evaluating D_1 and D_2.

The same reasoning is applied to the dependency ($[FDS \geq 1380] \wedge \neg C_{SM} \wedge S_{lib} \Rightarrow S_{MTA}$) (see $\boldsymbol{Proof_{MTA}}$) using the rules ($ITriv$, $GAnd$, $GServ$). The function $CalcF$ is used to calculate the sets of forbidden components and forbidden services from a predicate.

During this phase, the optional dependency ($?(S_{Amavis} \Rightarrow S_{AV})$ is checked to decide whether it provides services or not ($IOpt_2$ in $\boldsymbol{Proof_{Amavis}}$). The rule ITRIV evaluates the effect of the installation of S_{AV} when the service (S_{Amavis}) is available. The

Predicate logic description	Maude description

Intra dependency : $D_1 \bullet D_2$
$\begin{cases} D_1 = ([FDS \geq 1380] \wedge \neg C_{SM} \wedge S_{lib} \Rightarrow S_{MTA}) \\ D_2 = ?(S_{Amavis} \Rightarrow S_{AV}) \end{cases}$
	Intra dependency: Algebraic term of sort Dependencies
	`POSTFIX : [ FDS >= 1380.0 ] /\ notC C-SM`
	`/\ S-lib => Smta andD`
	`? S-Amavis => Sav .`

Target system description
$\begin{cases} \mathcal{E} = \{FDS = 500000, OS = Linux, RAM = 256\}; \\ \mathcal{C} = \{(C_1, S_{lib}, \varnothing, \varnothing), (C_A, S_{Amavis}, \varnothing, \varnothing)\} \\ \mathcal{G} = \{C1.S_{lib}, C_A.S_{Amavis} ...\} \end{cases}$
	Target system description: algebraic term of sort Context
	`ctx((FDS=500000)(OS=Linux)(RAM=256.0),`
	`< C1, S-lib, nils, nilc >`
	`< CA, S-Amavis, nils, nilc >,`
	`| (C1.S-lib) (CA.S-Amavis), nile|...)`

Table 3: *POSTFIX and its target system specification*

Installability:

$$\textsc{CAnd}\ \dfrac{\textsc{CTriv}\ \dfrac{\textsc{PTriv}\ \dfrac{\textsc{PVar}\ \dfrac{500000 \geq 1380}{Ctx \vdash_P FDS \geq 1380} \quad \textsc{PNotC}\ \dfrac{C_{SM} \notin AC(Ctx) = \{C_1, C_A\}}{Ctx \vdash_P \neg C_{SM}} \quad \textsc{PServ}\ \dfrac{S_{lib} \in AS(Ctx) = \{S_{lib}, S_{Amavis}\}}{Ctx \vdash_P S_{lib}}}{Ctx \vdash_P [FDS \geq 1380] \wedge \neg C_{SM} \wedge S_{lib}} \quad S_{MTA} \notin FS(Ctx) = \varnothing}{Ctx \vdash_C [FDS \geq 1380] \wedge \neg C_{SM} \wedge S_{lib} \Rightarrow S_{MTA}} \quad Ctx \vdash_C ?(S_{Amavis} \Rightarrow S_{AV})}{Ctx \vdash_C ([FDS \geq 1380] \wedge \neg C_{SM} \wedge S_{lib} \Rightarrow S_{MTA}) \bullet ?(S_{Amavis} \Rightarrow S_{AV})}$$

Figure 3: *Installability proof of POSTFIX*

sub-dependency graph is constructed progressively by binding the appropriate provided and required services using the rule GSERV.

After the installation of `POSTFIX`, the MTA service (S_{MTA}) and the anti-virus (S_{AV}) are provided and the component `sendmail` (C_{SM}) becomes forbidden. The dependency graph $\mathcal{G}$ corresponds to the union of the sub dependency graphs deduced from the two sub-dependencies D_1 and D_2, that is:
$\mathcal{G} = \{C_A.S_{Amavis} \overset{O}{\longmapsto} C_{PX}.S_{AV}, C_1.S_{lib} \overset{M}{\longmapsto} C_{PX}.S_{MTA}\}$. Therefore, after the installation of `POSTFIX`, the context becomes:

$$\begin{cases} \mathcal{C} = \{(C_1, S_{lib}, \varnothing, \varnothing), (C_2, S_{Amavis}, \varnothing, \varnothing), \\ (C_{PX}, \{S_{MTA}, S_{AV}\}, \varnothing, \{C_{SM}\})\}, \\ \mathcal{G} = \{C_A.S_{Amavis} \overset{O}{\longmapsto} C_{PX}.S_{AV}, C_1.S_{lib} \overset{M}{\longmapsto} C_{PX}.S_{MTA}\} \end{cases}$$

The corresponding `POSTFIX` installation in Maude is simply defined by the rewrite command execution (see Listing 5). Maude executes this command relatively to the module `TEST-INSTALL` described in Listing 4.

The current configuration `my-conf` is then declared via an axiom of this module, applying the meta rewrite rule `install` (declared in the `CONFIG-INSTALL` module in Listing 3 on page 6),

Listing 4: Installation of `POSTFIX` using Maude module

```
mod TEST-INSTALL is
including CONFIG-INSTALL .
ops S-lib S-Amavis Smta Sav  : -> Service [ctor] .
ops FDS OS RAM : -> EnvProp [ctor] .
ops LINUX : -> Val [ctor] .
op my-conf : -> Configuration [ctor] .
eq my-conf = ctx( ( FDS = 500000.0 ) ( OS = LINUX )
             ( RAM = 128.0 ),
         < C1 , S-lib , nils , nilc >
         < CA , S-Amavis , nils , nilc > ,
      | ( C1 . S-lib ) ( CA . S-Amavis ) , nile | )
POSTFIX :
   [ FDS >= 1380.0 ] /\ notC C-SM /\ S-lib => Smta
   andD ? S-Amavis => Sav .
endm
```

we will obtain the new configuration indicated as result of the rewriting process. This configuration shows that the installability of `POSTFIX` component is possible and has been done successfully since its context has changed. If it is not the case, the Maude engine will return the initial configuration with no changes. It is evident that rewriting process may deal with deployment of some components in parallel.

Installation:

$$\textsc{IAnd}_3\ \frac{Ctx, C_{PX} \vdash_I (P \Rightarrow S_{MTA}) \Rightarrow_{\boldsymbol{I}} \boldsymbol{Proof_{MTA}}\ \ Ctx, C_{PX} \vdash_I ?(S_{Amavis} \Rightarrow S_{AV}) \Rightarrow_{\boldsymbol{I}} \boldsymbol{Proof_{Amavis}}}{Ctx, C_{PX} \vdash_I (P \Rightarrow S_{MTA}) \bullet ?(S_{Amavis} \Rightarrow S_{AV}) \Rightarrow_{\boldsymbol{I}} \{S_{MTA}, S_{AV}\}, \varnothing, \{C_{SM}\}, \mathcal{G}}$$

$$\boldsymbol{Proof_{MTA}}\textsc{:ITriv}\ \frac{\textsc{GAnd}\ \dfrac{\textsc{GServ}\ \dfrac{S_{lib} \in AS(Ctx) = \{S_{lib}, S_{Amavis}\}}{Ctx, C_{PX}, S_{MTA} \vdash_G S_{lib} \Rightarrow \{C_1.S_{lib} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{MTA}\}}\quad Ctx, C_{PX}, S_{MTA} \vdash_G \neg C_{SM} \Rightarrow \varnothing \qquad Ctx, C_{PX}, S_{MTA} \vdash_G [FDS \geq 1380] \Rightarrow \varnothing}{Ctx, C_{PX}, S_{MTA} \vdash_G P \Rightarrow \{C_1.S_{lib} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{MTA}\}}\quad S_{MTA} \notin \varnothing \qquad CalcF(P) = \{FS(P) = \varnothing, FC(P) = C_{SM}\}}{Ctx, C_{PX} \vdash_I (P \Rightarrow S_{MTA}) \Rightarrow_{\boldsymbol{I}} \{S_{MTA}\}, \varnothing, \{C_{SM}\}, \{C_1.S_{lib} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{MTA}\}}$$

$$\boldsymbol{Proof_{Amavis}}\textsc{:IOpt}_2\ \frac{\textsc{ITriv}\ \dfrac{\textsc{GServ}\ \dfrac{S_{Amavis} \in AS(Ctx) = \{S_{lib}, S_{Amavis}\}}{Ctx, C_{PX}, S_{AV} \vdash_G S_{Amavis} \Rightarrow \{C_A.S_{Amavis} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{AV}\}}\quad S_{AV} \notin FS(Ctx) = \varnothing \quad CalcF(S_{Amavis}) = \varnothing, \varnothing}{Ctx, C_{PX} \vdash_I (S_{Amavis} \Rightarrow S_{AV}) \Rightarrow_{\boldsymbol{I}} \{S_{AV}\}, \varnothing, \varnothing, \{C_A.S_{Amavis} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{AV}\}}}{Ctx, C_{PX} \vdash_I ?(S_{Amavis} \Rightarrow S_{AV}) \Rightarrow_{\boldsymbol{I}} \{S_{AV}\}, \varnothing, \varnothing, \{C_A.S_{Amavis} \overset{\mathsf{O}}{\longmapsto} C_{PX}.S_{AV}\}}$$

Figure 4: *Installation proof of* `POSTFIX`

Listing 5: rewrite command execution of `POSTFIX` installation

```
Maude> rewrite in TEST-INSTALL : my-conf .
rewrites: 36 in 7621250285ms cpu (0ms real)
(0 rewrites/second)
result Context: ctx((FDS = 5.0e+5) (OS = LINUX)
(RAM = 1.28e+2),< C1,S-lib,nils,nilc >
< CA,S-Amavis,nils,nilc >,
|(C1 . S-lib) (CA . S-Amavis) (POSTFIX . Smta)
(POSTFIX . Sav),
 link((POSTFIX . S-lib) . (C1 . S-lib) . M)
 link((POSTFIX . S-lib) . (C1 . S-lib) . O)
 link((C1 . S-lib) . (POSTFIX . S-lib) . M)
 link((CA . S-Amavis) . (POSTFIX . S-Amavis) . O)|))
```

5. THE RELEVANCE OF THE INSTALLATION APPROACH

In this paper, we have extended the formal installation framework based on predicate logic (Belguidoum and Dagnat 2007) by using a rewriting logic formalism to overcome some of its limitations. In fact, the proposed predicate logic based model gives good example of formal specification of component intra-dependencies and its installation, but hereby some of its limitations:

- The specified deployment is *static* because the predicate logic does not support the concurrent change, it does not consider the dynamic and concurrent changes of the environment ;
- *Proof* of the *success* and the *safety* of a simple deployment operation is too complex, since it needs the use of a distinct tool, a simpler and automatic proof method and a distributed component deployment model checker must be provided.
- In the case of nondeterministic application of the installation rules, there is no specific *strategy* for choosing the appropriate rule to be applied as it may be done in Maude with the internal strategy concept.

Table 4 presents the complementarity between predicate logic based approach (Belguidoum and Dagnat 2007) and Maude based approach. We can see that Maude overcomes some limitations such as: genericity, meta-level programming, efficient execution of formal specification and formal proof of concurrent and distributed component deployment using automatic formal proof.

Genericity and meta-programming
Each concept involved in the deployment approach of software components has a precise semantics. Conceived Maude modules, specify not just theories, but also the intended mathematical models, that are algebras or categories.

Furthermore, the proposed approach is general enough since the rewriting theory `CONFIG-INSTALL` presented in Listing 3 on page 6 is a generic model of software components installation; it remains valid for any component or environment examples. Indeed only one equation is included in this module to specify clearly and in a global manner the installation of a given software component in a specific context. These particular structures are introduced by additional constant operators (see Listing 4 on

Main features	Predicate logic based framework	Maude based framework
Genericity	the genericity is not intrinsic to the predicate logic	Maude modules and Maude commands constitute a general solution for any component deployment
Meta-level	UML meta-model for deployment and for component (Belguidoum and Dagnat 2009)	Maude implements Rewriting logic which is intrinsically reflective, it can use meta-programming, internal strategies, parametrized modules (Full Maude)
Automatic formal proof	the safety and the success of deployment have to be proven thanks to another extra tool needing a translating process	using LTL and ITP tools of Maude
Formal execution tools	a framework developed with Ocaml	Core Maude and Full Maude
Concurrent and distributed deployment	concurrency and distribution are not intrinsic to the predicate logic	to be able to execute in parallel several rules at the same time. The distributed character of the environment structure is taken into account by this specification

Table 4: *The relevance of the Maude based framework*

page 8). This represents a typical instance of the generic model.

Automatic formal proof
Our model may actually exhibit analysis and formal checking techniques using formal execution tools of Maude especially the LTL model checker for our case study. The benefit of The Maude tool is to use it for both specification and verification of component deployment. Using the given Maude specification and a temporal logic formula as an input of the LTL tool specified by the predefined Maude module `MODEL-CHECKER.maude`. For example, we have specified the installation *success* (see Listing 6) and the installation *safety* (see Listing 7) in Maude and verify them using temporal logic formulae and Maude LTL tool. Note that the installation *success* verifies certain conditions only on the considered component and the installation *safety* verifies conditions throughout target system.

In this paper the property of installation success verifies for an installed component that it should never belong to the set of forbidden component of the context. The property `IsSuccessfullyInstalled` is defined in the system Maude module `SUCCESS-INSTALL-CHECKER` as an operation which takes a component as an argument and returns a proposition. The evaluation of this property is defined by Maude equations, it checks if the considered component which belongs to the set of installed component (`C BelongsTo InstC(I)`) in the context `ctx(E,I,G)` belongs to the set of forbidden component (`C BelongsTo ForbiddenC(I)`). Indeed, the installed component is *successfully installed* if it does not belong to the set `ForbiddenC(I)`.

Listing 6: A Maude module for analysis component installation success

```
mod SUCCESS-INSTALL-CHECKER is
protecting TEST-INSTALL .
including MODEL-CHECKER .
including SATISFACTION .
subsort Configuration < State .
vars C : Component .
var P : Prop .
var cnf :  Configuration .
var E : EnvVars .
var I : InstalledCs .
var G : Graph .
op IsSuccessfullyInstalled : Component -> Prop .
ops initial-conf : -> Configuration .
eq initial-conf = my-conf .
eq ctx(E,I,G) cnf |= IsSuccessfullyInstalled(C) =
       not(C BelongsTo InstC(I)
       and C BelongsTo ForbiddenC(I)) .
eq cnf |= P = false [owise] .
endm
```

The success property is expressed by the following LTL formula:
`[] IsSuccessfullyInstalled(POSTFIX) .`
It means that `POSTFIX` should be *always* successfully installed. Hence, we can check that the system Maude module `SUCCESS-INSTALL-CHECKER` satisfies this property, or obtain a useful counterexample showing that this property is violated. In our example, the property is checked using the following command:

```
Maude> load model-checker.maude .
Maude> reduce in SUCCESS-INSTALL-CHECKER :
    modelCheck(initial-conf,
            []IsSuccessfullyInstalled(POSTFIX)) .
rewrites: 97 in -2100562889694ms cpu (0ms real)
(~ rewrites/second)
result Bool: true
```

It is satisfied and returns `true` as boolean result.

The property of installation *safety* verifies that all installed component in the target system should never belong to its set of forbidden component. The property `safety` is defined in the system Maude module `SAFETY-INSTALL-CHECKER` (see Listing 7) as an operation which is defined by a set of Maude equations. The property is not verified (equals to `false`) in two cases: if the considered component `C` belongs to the set of its forbidden component $< \boldsymbol{C}, PS = P, FS = F, FC = \boldsymbol{CCs} >$ (see the first equation in Listing 7) or if the component `C` belongs to the set of forbidden component of another component (see the second equation in Listing 7). In the other cases, the safety property is verified (equals to `true`).

Listing 7: A Maude module for analysis installation safety

```
mod SAFETY-INSTALL-CHECKER is
protecting TEST-INSTALL .
including MODEL-CHECKER .
including SATISFACTION .
 subsort Configuration < State .
 vars C CC : Component .
 var S : Service .
 vars P P1 : Services .
 vars F F1 : Services .
 vars Cs Css : Components .
 var I : InstalledCs .
 var PP : Prop .
 var cnf :  Configuration .
 var E : EnvVars .
 var G : Graph .
 op safety : -> Prop .
 eq ctx( E , < C , P , F , C Cs > I , G ) cnf
                          |= safety = false .
 eq ctx(E , < C , P  , F , Cs > < CC , P1
, F1 , C Css > I , G) cnf
                          |= safety = false .
 eq cnf |= PP = true [owise] .
 endm
```

The property is expressed by the LTL formula: `[] safety`, it is satisfied (returns `true`) in the system Maude module `SAFETY-INSTALL-CHECKER` using the following command:

```
Maude> reduce in SAFETY-INSTALL-CHECKER :
     modelCheck(initial-conf, []safety) .
rewrites: 73 in 171853974326ms cpu (0ms real)
(0 rewrites/second)
result Bool: true
```

Concurrent and distributed deployment
This model defines a concurrent semantics of the dynamic deployment process. For instance, distributed configurations extending the previous one, shown in our illustrative example is described in Listing 8.

This complex configuration represents a distributed structure of existing components and their predisposed context. For example, `POSTFIX` in `ctx` and `WindowsMail` in `ctx'`, etc. (see Listing 8). This initial complex configuration is similarly rewritten with parallel application of essentially local rule `install`

Listing 8: An example of distributed configurations

```
...
eq distributed-conf = ctx( ( FDS = 500000.0 )
        ( OS = Linux ) ( RAM = 256 )  ,
< C1 , S-lib , nils , nilc >
< CA , S-Amavis , nils , nilc > ,
| ( C1 . S-lib ) ( CA . S-Amavis ) , nile | )
POSTFIX :
[ FDS >= 1380.0 ] /\ notC C-SM /\ S-lib => Smta
 andD  ? S-Amavis => Sav ,
ctx'( ( FDS = 600000.0 ) ( OS = windows )
( RAM = 1000 )  ,
< C2 , S-lib , nils , nilc >
< Kaspersky , S-antiVirus , nils , nilc > ,
| ( C2 . S-lib ) ( Kaspersky.S-antiVirus ) , nile |)
WindowsMail :
[ FDS >= 2670.0 ] /\  S-lib => Smta
 andD  ? Kaspersky.S-antiVirus => Sav .
```

and others defined equations as well as deduction rules of this logic.

Maude system is then used to implement these theories. The models described here have precise mathematic semantics and are directly executable via the Maude system.

6. CONCLUSION

In this paper, we focus on the concurrent and dynamic component installation. To achieve this purpose, we have presented a complementary approach to the predicate logic based one (Belguidoum and Dagnat 2007) using the rewriting logic model and its Maude implementation which is a high performance logical framework. Indeed, the component deployment formalization based on predicate logic has been naturally encoded in rewriting logic. The result specification is then executed and may be formally analyzed. We have shown how rewriting logic overcomes the limitations of the predicate logic based framework. In fact, using rewriting logic we can specify in Maude both the dynamic and concurrent changes of the environment and the component deployment. We have specified software components via their dependencies description, the target system capable to shelter the component and the installation rule in Maude. We have shown the relevance of the Maude based deployment approach relating to its intrinsic interesting features such as: genericity, concurrency, distribution, Maude formal tools.

Future work will involve the development of a complete framework for the entire deployment phases of component-based software. Hence, we plan to exploit the following Maude aspects:

- *Object oriented* modules of Maude, extending all the module operations available in Core Maude with a more convenient syntax to

support object-oriented concepts (objects, messages, classes, inheritance, etc.).

- *Internal strategies* of Maude module execution that guide the possibly nondeterministic application of rewriting rules. These strategies may be defined by rewrite rules in a *metalevel* module.
- *Parameterization* of Maude modules, to specify intra-dependencies as a parameterized contract or parameterized deployment rules depending on the user context.

7. REFERENCES

Aiguier, M., Bahrami, D. and Longuet, D. (2006). An abstract way to define rewriting logic, *Electron. Notes Theor. Comput. Sci.* **159**: 205–226.

Belguidoum, M. and Dagnat, F. (2006). Analysis of deployment dependencies in software components, *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, ACM Press, New York, NY, USA, pp. 735–736.

Belguidoum, M. and Dagnat, F. (2007). Dependency management in software component deployment, *Electron. Notes Theor. Comput. Sci.* **182**: 17–32.

Belguidoum, M. and Dagnat, F. (2008). Formalization of Component Substitutability, *Electron. Notes Theor. Comput. Sci.* **215**: 75–92.

Belguidoum, M. and Dagnat, F. (2009). Vers un déploiement sûr et flexible des composants logiciels, *NOTERE 2009 : neuvième conférence internationale sur les nouvelles technologies de la répartition*, Montreal, Canada.

Borovanský, P., Kirchner, C., Kirchner, H. and Moreau, P.-E. (2002). ELAN from a rewriting logic point of view, *Theor. Comput. Sci.* **285**: 155–185.

Carzaniga, A., Fuggetta, A., Hall, R., Hoek, A., Heimbigner, D. and Wolf, A. (1998). A characterization framework for software deployment technologies, *Technical Report CU-CS-857-98*, Department of Computer Science, University of Colorado.

Clavel, M., Durn, F., Eker, S., Lincoln, P., Martoliet, N., Meseguer, J. and Talcott, C. (2011). Maude manual (version 2.6). `http://maude.cs.uiuc.edu/`.

Diaconescu, R. and Futatsugi, K. (1998). *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, Vol. 6 of *AMAST Series in Computing*, World Scientific.

Eker, S., Meseguer, J. and Sridharanarayanan, A. (2002). The Maude LTL model checker, *in* F. Gadducci and U. Montanari (eds), *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*, Vol. 71 of *Electronic Notes in Theoretical Computer Science*, Elsevier.

Grunske, L., Reussner, R. and Plasil, F. (eds) (2010). *Component-Based Software Engineering, 13th International Symposium, CBSE 2010, Prague, Czech Republic, June 23-25, 2010. Proceedings*, Vol. 6092 of *Lecture Notes in Computer Science*, Springer.

Heydarnoori, A. (2008). *Deploying Component–Based Applications: Tools and Techniques*, Vol. 253 of *Studies in Computational Intelligence*, Springer-Verlag, Prague, Czech Republic, pp. 29–42.

Liu, Y. D. and Smith, S. F. (2006). A formal framework for component deployment, *In Proceedings of OOPSLA'2006*, Portland, Oregon, pp. 325–344.

Martí-Oliet, N. and Meseguer, J. (2000). Rewriting logic as a logical and semantic framework, *in* J. Meseguer (ed.), *Electronic Notes in Theoretical Computer Science*, Vol. 4, Elsevier Science Publishers.

Martí-Oliet, N. and Meseguer, J. (2002). Rewriting logic: roadmap and bibliography, *Theor. Comput. Sci.* **285**: 121–154.

Meseguer, J. (2004). Rewriting logic semantics: From language specifications to formal analysis tools, *In Proceedings of the IJCAR 2004. LNCS*, Springer LNAI, Cork, Ireland, pp. 1–44.

Meseguer, J. and Roşu, G. (2007). The rewriting logic semantics project, *Theoretical Computer Science* **373**(3): 213–237.

Mouakher, I., Lanoix, A., Souquires, J., Cnrs, L. and Nancy, U. (2006). Component adaptation: Specification and verification, *Proceedings of the 11th International Workshop on Component Oriented Programming (WCOP06)*, Nantes, France, pp. 23–30.

Parrish, A. S., Dixon, B. and Cordes, D. (2001). A conceptual foundation for component-based software deployment., *Journal of Systems and Software* **57**(3): 193–200.

Postfix (2009). The Postfix Home Page. `http://www.postfix.org/`.

Reussner, R. H. (2001). The Use of Parameterised Contracts for Architecting Systems with Software Components, *in* W. Weck, J. Bosch and C. Szyperski (eds), *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*, Budapest, Hungary.

Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley.

Vieira, M. and Richardson, D. (2002). The role of dependencies in component-based systems evolution, *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, ACM Press, New York, NY, USA, pp. 62–65.