A calculus for a new component model in highly distributed environements

Antoine Beugnard, Ali Hassan

A Calculus for a New Component Model in Highly Distributed Environments

Antoine Beugnard[1]

Computer Science Department
Telecom Bretagne
Brest, France

Ali Hassan[2]

Computer Science Department
Telecom Bretagne
Brest, France

Abstract

The current software systems and their corresponding deployment environments are highly complex and demanding. Multiple and unstable network technologies, resource-restricted devices, and mobility, are few examples of these complexities. In this paper we propose a new component model, called Cloud Component (CC), that copes with the challenges posed by mobile and pervasive environments. Traditional distributed applications are based on distribution transparency, where a middleware layer is expected to handle and hide all remote communication. Cloud component model is the result of a paradigm shift from distribution transparency to localization acknowledgment, where all details of the deployment environment including networks and communication, mobile devices, constrained devices, and sensors, are considered a first class concern. The cloud component model is presented informally and formally with a mathematical notation. The informal notation allows for faster comprehension of the general concepts. While the formal notation opens the door for a wide range of theoretical topics and provides a precise language to describe details. We also propose an assembly model to build large systems using CCs as building blocks. This assembly model is presented formally and fully implemented for the designer to be able to automatically check if his/her design conforms to the CC assembly model.

Keywords: Software Components, Formal Models, Component Assembly, Automatic Design Checker, Mobile and Pervasive Computing

[1] Email: Antoine.Beugnard@telecom-bretagne.eu
[2] Email: ali.hassan@telecom-bretagne.eu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

During the last years new distributed platforms have emerged, often qualified as highly distributed environments (HDE). HDEs still include powerful and robust machines but they are rather composed of resource-constrained and mobile devices such as laptops, personal digital assistants (or PDAs), smartphones, GPS devices, sensors, etc [7]. Moreover, these devices communicate using a variety of dependable and undependable fixed and wireless networks.

This fundamental change in the deployment environment was not accompanied by a theoretical software model that provides deep understanding and systematic solutions to build compatible software systems [1].

As Malek *et al.* [10] have noticed "transparency (i.e. hiding distribution, location, and interaction of distributed objects) is considered a fundamental concept of engineering distributed software systems, as it allows for the management of complexity associated with the development of such systems". This is usually achieved through the utilization of a middleware layer that has as a main function (among others) to make remote calls appear as local calls. That is correct for stable distributed systems, however, this same concept, distribution transparency, has been shown to suffer from major shortcomings when applied extensively in HDEs [10].

That leaves us in the following situation: there is excessive and increasing need to build complex mobile and pervasive systems for entertainment and professional uses. And at the same time, the fundamental engineering techniques available are inherited from stable distributed environments, and suffer from several drawbacks and weaknesses when utilized in these new environments. The only available answer currently is applying ad-hoc techniques to overcome these drawbacks and weaknesses.

This work is a direct response to the above mentioned challenge. First we propose a paradigm shift from *remote communication transparency* to *localization* being the first class concern. In other words, we no more hide or abstract location, on the contrary, we acknowledge all aspects related to location including the specification of devices, the networking paradigms they use, the different network specifications available, security features, and all related characteristics of the deployment environment. We discuss the limitations of current component models, the paradigm shift needed, and selected set of related work in section 2.

To achieve the above mentioned objective, we propose in section 3 a novel component model called *cloud component* (CC). This model includes the expected deployment environment in its definition, i.e. we raise the importance of deployment environment to be equal to the functionality required from the component. The other important feature of this novel model is that it is *fundamentally distributed*. A single CC is usually distributed over many dis-

tant hosts, the specification of these hosts are considered and fundamentally acknowledged during the development process of this CC, and all aspects related to communication, coordination, and quality of service are migrated to be internal to the border of the CC.

A software component can be thought of as unit of assembly [3] . This is true for all component models including cloud component model. In this paper we propose a new approach to assemble CCs using systematic methodology that maintains the properties of CC model. CC assembly is a tool to build large systems using CCs as building blocks. Moreover, we present a technique to automatically check the validity of this assembly. Cloud component assembly and checking are presented in section 4.

The cloud component model and CC assembly are presented informally and formally with a mathematical notation. The informal notation allows for faster comprehension of the general concepts. While the formal notation opens the door for a wide range of theoretical topics including component type inference, subtypes, etc, and provides a precise language to describe details. In addition, formal methods allow the designer to produce machine readable designs where automated tools can verify specific properties at design time, which in turn, increases the level of confidence in the correctness of design. We conclude with a brief summary of our proposals and some future work.

2 Highly Distributed Environments - HDE

The emergence of mobile devices such as portable notebook computers, tablet computers, PDAs, and mobile phones, and the advent of various wireless networking solutions make computation possible anywhere [14,13,12]. In this paper, we define highly distributed environments as a target platform of our work. These networks include distributed systems with laptops and wireless networks, mobile systems, and pervasive systems. These networks violate many familiar assumptions about the behavior of distributed environments, and demand new techniques to build compatible and optimized software [1], especially at the architectural level of the software development process.

The characteristics that the HDE infrastructure imposes include [12,1,4,5]: 1- Mobility of hardware, data, and code. 2- Heterogeneity of software and devices. 3- Volatility of hardware and software components. 4- Small devices, highly constrained resources, dynamic resources. 5- Connectivity failure are not rare; disconnected operations. 6- High bandwidth and low latency are no more available in continuous and dependable manner. 7- Software components communicate using a variety of interaction paradigms (e.g.,

[3] In this article we prefer to use the word *assembly* rather than *composition* since the output of this operation (assembly) is not a software component.

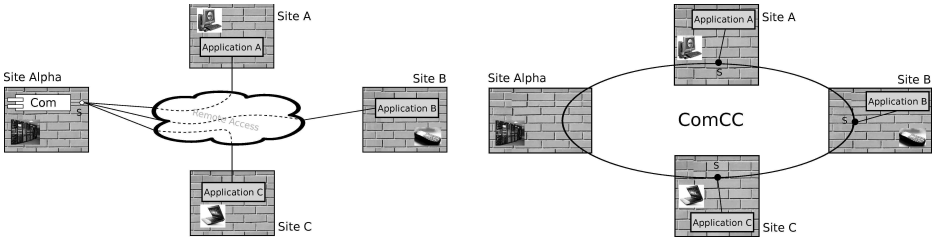

Fig. 1. Left: Distributed component model (CCM, EJB, .Net, etc) - remote access; no control over the underneath infrastructure. Right: Cloud component model - local access; the component is responsible for remote communication.

SOAP messaging, media streaming).

In spite of the above challenges that permeate the entire traditional software development life cycle, software in these systems are expected to obey the following constraints [4,14,5,10]: 1- Customized implementation: the implemented software need to be efficient, customized, and can be deployed on resource constrained devices. 2- Correctly respond to runtime changes in the environment. 3- Preserve dependability and quality of service in this highly dynamic environment.

2.1 Current component models limitations

The concept of software components has been widely adopted because of its attractive and powerful encapsulation attributes [3,9]. Lau *et al.* noted: "Encapsulation has the potential to counter *complexity*" [9].

After analyzing several component technologies such as CCM and EJB for industry and Fractal and SOFA for academia, we found that they follow a common paradigm. These component models rely on strong assumptions, and they emulate local call on top of distributed networks, and finally they consider any deviation from their implicit or explicit assumption as exceptions. All of these points are considered limitations with regards to HDEs. For more detailed discussion on these limitations, please refer to appendix A.

2.2 Paradigm shift with cloud components

To overcome these limitations, we propose a new component model called *Cloud Component* (CC) which is a novel extension to the 'Medium' concept proposed by Beugnard et al [2,11]. CC encompasses all features provided by currently existing component models and, moreover, is especially designed to be used in the above mentioned complex environments. CC model provides the capability of the instantiation of its interface(s) on each host that potentially needs to access the service provided by this CC [2]. This will make the service access explicitly local. In other words, if we want our component to be accessed

at some host, we need to deploy an interface instance at that host. It is evident that this instance will have some sort of remote communication with other entities inside the component, but this is internal with respect to the component border as explained in figure 1.

Migrating the communication to be internal inside the CC border has significant contribution to the overall architecture of distributed applications. In figure 1 (left) the server provides a service S which is accessible in sites A, B, and C. If the resources at site A are not enough, or the connectivity at site C is not adequate, or simply the configuration of site B is not compatible, the service S of component Com is not accessible, i.e. useless.

With CC, figure 1 (right), this is not the case. Cloud component comCC has its interface S instantiated on sites A, B, and C. Using it is simply a local access of a locally available service. S at the three sites provides the same (or similar) functionality, however, it is possible (and highly probable in this case) to be implemented using completely different approaches. For example, in site A special arrangements should be carried out to handle the extremely limited resources and the mobile networking. In site C, constraints of site A are relaxed, as result, different implementation technologies are utilized. The same argument holds for site B, where there are stable fixed networking and power supply, and rather advanced resources. At this site, there is no need for the implementation to be prepared to handle complexities that arise in sites such as A or C.

In other words, we allow several implementations of the same functionality to exist side by side. The variation of implementation is driven by the variation of deployment environment, i.e. the characteristics of the hosts where interfaces will be instantiated, and the characteristics of connectivity available for each host. For the interface S to be instantiated on site A, there need to be implementation variant that is compatible with the characteristics of site A. This compatibility is checked statically before the instantiation of the system and each of its roles but is out of the scope of this article.

2.3 Related Work

Didier Hoarau *et al.* deal with the challenges of HDEs [5,6,7]. However, their solution has different scope from ours. First, they expand the already existing component model Fractal. Second, they only model and handle the disconnection of network connection among all characteristics of HDE.

Marija Mikic-Rakic provides a sophisticated response to one challenge proposed by HDEs, which is the discontinuity of services where the system needs to continue functioning in the near absence of the network [12]. This work proposes a redeployment solution as part of a middleware called Prism-MW.

Finally, and the most related work, Sam Malek *et. al.* propose a frame-

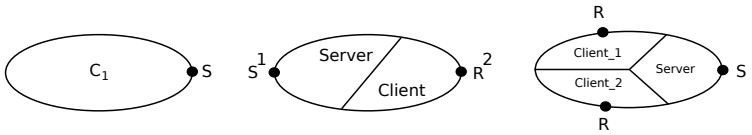

Fig. 2. Left: CC style with a single interface S. Middle: CC with two roles, cardinality, and location. Right: CC *com* with two roles and three hosts.

work and tools to support the complete software engineering life-cycle for the development of HDE applications [10]. While their tools help overcome the challenges posed by HDEs, these challenges become natural details in our novel component model, where they can be handled systematically.

3 Cloud Component Model

We chose the name Cloud Component since our component model encompasses physical borders and hence hides the technologies, implementation variants, and architecture choices used to conform to the physical topology of the underneath infrastructure. Our approach in presenting cloud component model is based on two different notations: the informal notation and the mathematical/formal notation. The informal notation is easier to understand and is highly dependent on figures, while the formal notation is more compact, more precise, and less ambiguous. The formal notation allows us to communicate precise details easier, and allows us to easily present statements and proofs.

3.1 The definition of cloud component

Definitions 1 to 5 collectively form the definition of cloud component.

3.1.1 Definition 1: Roles

Let C_1 be a cloud component with single interface S as illustrated in figure 2 (left). S is defined through an Interface Definition Language - IDL. We assume S defines the signature of provided and required functions of C_1. The contract of this interface could be more sophisticated, but we restrict its definition for the sake of simplicity.

A cloud component can have several interfaces : P, Q, R, etc. We call these interfaces 'roles' because their identification (set of functions they gather) is guided by the way the component can be used through this interface. The cloud component in figure 2 (middle) has two roles: S and R.

3.1.2 Definition 2: Cardinality

Each cloud component can have several roles. In addition, the role is allowed to have several instances, i.e., several carbon copies of the same IDL. The total number of instances of a role in a running version of the component is called: the cardinality of the role. In figure 2 (middle) the role S has cardinality one and the role R has cardinality two. Combined with location property (explained later), this approach will encapsulate the communication and all its details and semantics inside the component.

3.1.3 Definition 3: Connection

Once the component border is defined, the connection rules can be defined. In order to suppress ambiguity of 1-to-many or many-to-many connections identified in [11] we allow a role to connect to only one role of another cloud component in a one-to-one connection.

This rule applies at the instance level, when cloud components are actually implemented. In order to allow a 1-to-many or many-to-many connectivity, we use 'role cardinality'.

3.1.4 Definition 4: Multiplicity

Cardinality is a number $k \in N$. We can allow more complex structure by not specifying k (at some point of the design). Instead, we put constraints on k called multiplicity. For example a role R can have multiplicity $[1..5]$, $[1..*]$, or simply $*$.
At this level of definition, we are not bounded by decidability features but only consider constraints definition.

3.1.5 Definition 5: Location

Each role is assigned a location to run on. The location in the most basic form is a computing host/device. In figure 2 (middle) a cloud component has two different roles, S and R. Role S has one instance that is located at the host $Server$. The role R has two instances that are located at the host $Client$. Figure 2 (right) presents a cloud component that has two different roles, S and R. Role S has one instance that is located at the host $Server$. The role R has two instances one of them is located at the host $Client\_1$ and the other is located at host $Client\_2$.

One should not mix our definition of location with the *geographic location*. Our model does not define or recognize geographic location, rather, we acknowledge location as a computing/electronic device that might be mobile or not. It is fundamental to assert that location is integral part to the CC definition, in other words, without location specification the cloud component

definition will not be complete. Finally, and at design and implementation stages, the collection of all locations are called *'the expected deployment environment.'*

3.2 Formal definition of cloud component

A single cloud component is defined using the following four-tuple:

(i) A finite set of roles $\overline{\Lambda}$.
(ii) A finite set of multiplicities for these roles $\overline{\mu}$.
(iii) A set of possible deployment environments $\overline{L}$. Each L is either a finite set of hosts $\overline{H}$, or a finite set of host types $\overline{T}$.
(iv) A function Z that maps roles to location types or hosts.

$$\Omega \equiv (\overline{\Lambda},\ \overline{\mu},\ \overline{L},\ Z)$$

The following formally defines the cloud component *com* in figure 2 (right):

$\Omega com \equiv (\overline{\Lambda},\ \overline{\mu},\ \overline{L},\ Z)$ where:

$\overline{\Lambda} = \{\Lambda S,\ \Lambda R\}$
$\overline{\mu} = \{(\Lambda S,\ 1),\ (\Lambda R,\ 2)\}$
$\overline{L} = \{\{TServer,\ TClient\_1,\ TClient\_2\}\}$
$Z:\ \ \Lambda S \downarrow TServer,\ \ \Lambda R \downarrow TClient\_1,\ \ \Lambda R \downarrow TClient\_2$

The formal definition is read as follows: the CC *com* is defined using its four-tuple. The set of roles contains two roles: role type S and role type R. Role type S has multiplicity 1, and role type R has multiplicity 2. The set of expected deployment environments has only one set, which contains three host types: host type $Server$, host type $Client_1$, and host type $Client_2$. Finally Z can be read as: The role S is localized at host of type $Server$. Role R has two instances, one is localized at host of type $Client_1$ and the other is localized at a host of type $Client_2$. Symbols used to construct the formal notation are summarized in table B.1 in appendix B.

3.3 Formal definition of cloud component based system

Generally, a software component can be thought of as 'unit of composition'. This is true for all component models including cloud component model. In CC model, roles are the only access points of the component. A role can serve as a connection port where component C_1 connects to other component C_2 as in figure 3 (left). We choose to assembly components in specific architecture to achieve our desired system specifications. *As result we have σ the set of assembly rules that includes the dependency rules between CCs, and all role connections.* Cloud component assembly will be discussed in detail in section 4.

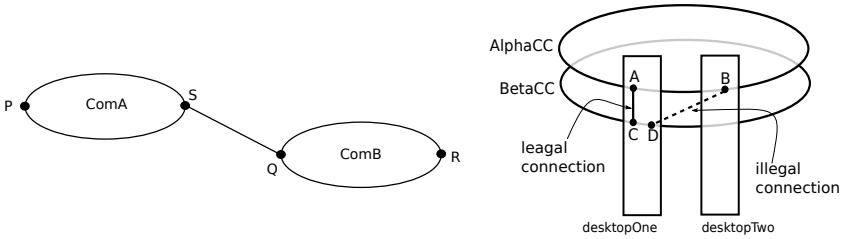

Fig. 3. Left: Two CCs are composed using roles S and Q. Right: Two CCs *AlphaCC* has two role instances A and B, and *BetaCC* has two role instances C and D. A, C, and D are hosted by *desktopOne*, while B is hosted by *desktopTwo*. Therefore, the connection between A and C is legal, whereas the connection between B and D is not permitted.

A system built using cloud components consists of:

(i) A finite set of cloud components $\overline{\Omega}$.
(ii) A finite set of multiplicities for these cloud components $\overline{M}$.
(iii) A set of assembly rules σ.
(iv) A set of possible deployment environments $\overline{L}$.

As result, the system type is fully defined using the "four-tuple" notation:

$$S \equiv (\overline{\Omega},\ \overline{M},\ \sigma,\ \overline{L})$$

Finally we define the system instance $\hat{S}$. Let S be a CC based system that is defined as above. $\hat{S}$ is an instance of that system and is defined using the following five-tuple:

(i) The system type S that we want to instantiate.
(ii) The function τ that takes a cloud component as a parameter and returns the number of instances of it.
(iii) The function K that takes a role as a parameter and returns its cardinality, i.e. number of instances.
(iv) The deployment environment L which is a finite set of hosts $\overline{H}$.
(v) The function Z that maps $\overline{\Gamma}$[4] to L.

$$\hat{S} \equiv (S, \tau, K, L, Z)$$

4 Cloud Component Assembly

4.1 Assembly Constraints

In CC model, roles are the only access points of the component. A role can serve as a connection port where component *ComA* connects to other component *ComB* as in figure 3 (left).

[4] Γ is defined in appendix B.

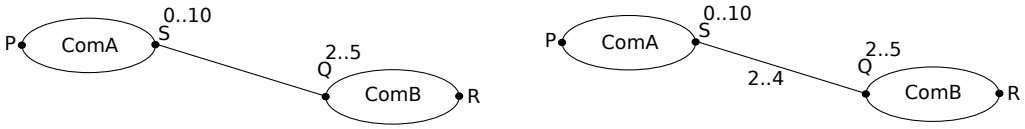

Fig. 4. The importance of the 'connection multiplicity'. Left: No information. Right: The multiplicity of the connection is defined: [2..4].

4.1.1 First constraint - one-to-one

An instance of role S can connect to one instance only of any other role at any time instance. We raised the importance of this constraint from being a recommended design choice to be a fundamental model constraint for several reasons. One of these reasons is to remove ambiguities in the connections. Another and important reason is to control the design precisely, and to be able using this control to ensure the delivery of the expected non-functional properties. As an example, let us take the role S in figure C.1 from the banking example in appendix C. And suppose S is hosted by some regular desktop. If S is expected to have 10 connections, i.e. 10 clients that want to use the video service, is completely different from S is expected to have 10^6 connections at the same time. The difference exists in the design, implementation, and the deployment host (probably a normal desktop will not be able to serve 10^6 connections). This difference should be recognized from the very early stages in the design, and this is done in CC model by setting the multiplicity (or cardinality) constraints over roles.

4.1.2 Second constraint - local connections only

Two instances of two roles can connect to each other only if both of them are instantiated at the same host as in figure 3 (right). If they are instantiated at different hosts they simply can not connect to each other. This is a direct result of the paradigm shift discussed in section 2.2. It is fundamental in our model to migrate all remote communications to be internal to the border of the CC itself. This migration means that these remote communications are designed and implemented using the special software development process [5] of the CC model, and more important, passed all checks necessary to ensure the quality of service expected.

4.1.3 Third constraint - Connection multiplicity

When there is a connection between two roles, that does not mean that all instances of these two roles should connect to each other. Figure 4 (left) is

[5] We propose a novel software development process to build CCs and CC based systems. The description of this process is out of the scope of this article and we will propose it in future publication.

an update of figure 3 (left) by adding multiplicities to roles S and Q. To understand the connection in this figure we need to see the uncertainty that exist at this phase of design. During runtime, there might be one instance of S and five instance of Q, or nine instances of S and two instance of Q. So how many connections we have at runtime between S and Q? To answer this question we need to remember that the final responsibility of the design is held by the designer himself, we only provide an advanced model and accompanied tools and checkers. To facilitate the assembly design we add the *connection multiplicity*, which is a range $[min..max]$, where min is the minimum number of connections that must exist at runtime, and max is the maximum number of connections that might exist at runtime, as in figure 4 (right). Usually these numbers reflect the need of either of the roles, or both. For example if I have a role W that connects an ATM machine (CC ATM) to the bank system (role S of CC *Agent*), I can expect W to need only one connection at runtime, i.e. [1..1]. On the other hand I expect S to allow zero or more connections at runtime, i.e. $[0..*]$. Please see figure C.1 in appendix C.

4.2 Formal definition of cloud component assembly

CC assembly is based on the connection operator $\otimes$, which is a binary operator that takes two CC roles and returns true if the designer explicitly listed those two roles to be connected (this is done in σ as described later), otherwise it returns false. The set of assembly rules is called σ and is defined using the following context free grammar:
$E \rightarrow \{\ I\ \}$
$I \rightarrow IJ,\ \mid\ IJ\ \mid\ \epsilon$
$J \rightarrow (\Omega var.\Lambda var \otimes \Omega var.\Lambda var,\ \ int,\ \ int)$
Where *var* and *int* are terminals such that: *var* represents any string of characters and *int* represents a positive integer. This grammar will recognize the following syntax:
$\sigma = \{(\Omega name.\Lambda name \otimes \Omega name.\Lambda name, min, max),\ \ (\Omega name.\Lambda name \otimes \Omega name.\Lambda name, min, max), ..., (\Omega name.\Lambda name \otimes \Omega name.\Lambda name, min, max)\}$.
The following shows the assembly in figure 4 (right) using formal notation:
$\sigma = \{(\Omega ComA.\Lambda S \otimes \Omega ComB.\Lambda Q,\ \ 2,\ \ 4)\}$
In general we can write:
$\sigma = \{(\Omega ComA.\Lambda S \otimes \Omega ComB.\Lambda Q,\ \ m,\ \ n)\}$
This connection has the following semantics: at least m instance of S connect to m instance of Q, and at most n instance of S connect to n instance of Q. *This is correct for one and only one instance of each cloud component.*

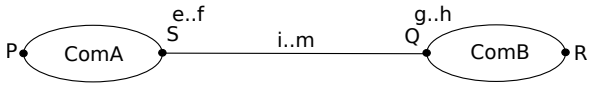

Fig. 5. CC assembly normal form A. Ranges are always consistent (i.e. $min \leqslant max$).

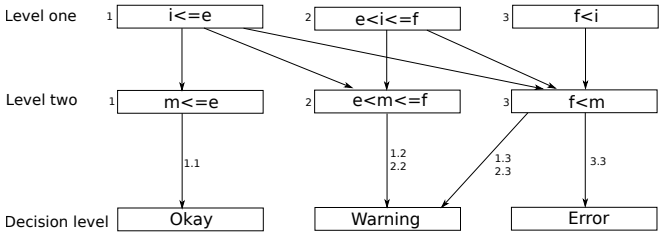

Fig. 6. The relation between the two ranges $[e..f]$ and $[i..m]$ in figure 5. We start with level one, and depending on the value of i we move to level two where we inspect the value of m. The label(s) on the arrows leading to the decision level indicate the decisions made on the upper two levels.

4.3 Remark

The connection operator $\otimes$ is symmetric:
$\Omega ComA.\Lambda S \otimes \Omega ComB.\Lambda Q \Longleftrightarrow \Omega ComB.\Lambda Q \otimes \Omega ComA.\Lambda S$
The above statement is read as follows: role S is connected to role Q if and only if role Q is connected to role S.

4.4 Assembly checking algorithm

Figure 5 presents the general case of assembly, which is defined as normal form A assuming there is a single instance of the CC. The connection here has the following semantics (as mentioned in the definition): at least i instance of $S(Q)$ connect to i instance of $Q(S)$, and at most m instance of $S(Q)$ connect to m instance of $Q(S)$. This is correct for one and only one instance of each cloud component $ComA$ and $ComB$.
The two ranges $[e..f]$ and $[g..h]$ are not related in any way since cloud components may have been designed independently. On the other hand, the two ranges $[e..f]$ and $[i..m]$ are related as in figure 6. Cases presented in figure 6 can be reduced to the following four cases:

(i) $i \leq e \ \ \& \ \ m \leq e \Rightarrow$ Valid

(ii) $i \leq e \ \ \& \ \ e \leq m \Rightarrow$ Warning

(iii) $e < i \leq f \Rightarrow$ Warning

(iv) $f < i \Rightarrow$ Error

The same argument holds for the two ranges $[g..h]$ and $[i..m]$ in figure 5. Depending on the numbers, we have three cases:

(i) Valid: in this case we do not have a chance of connection problems at runtime if the instantiation of roles respected the design.

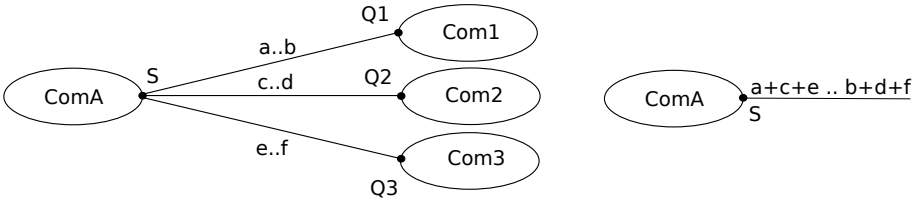

Fig. 7. Left: CC assembly normal form B. Multiple connections - role S is connected to three roles $Q1$, $Q2$, and $Q3$. Right: Role S after assembly reduction - phase one.

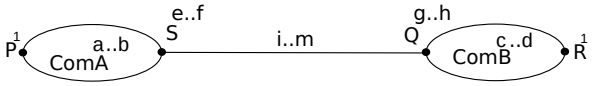

Fig. 8. CC assembly normal form C. Other CCs can connect to Q, S, etc. Omitted for space.

(ii) Warning: in this case the designer need to be careful because even if the instantiation respected the minimum requirements, we might face invalid situations. For example, if $e < i \leq f$, and at runtime we have only e instances of S (legal situation), and we need i connections to S. This situation will produce runtime error. As result, before asking for i connections to S, the application must instantiate at least i instances of S (possible because $i \leq f$).

(iii) Error: here we do not have enough instances of the role to satisfy the minimum connections need.

A role (specifically, role type) is not limited to be connected to only one other role, rather, this number is unlimited. At runtime, this role is expected to have several instances, where each instance is connected to one other role. This is assembly normal form B and presented in figure 7 (left). To check this assembly we need to get it back to normal form A in figure 5. We call this conversion from the form normal form B to normal form A: assembly reduction - phase one. For role S in figure 7 this is accomplished as in figure 7 (right). Formally: Let $\sigma = \{(S \otimes Q_1, a_1, b_1), (S \otimes Q_2, a_2, b_2), \cdots, (S \otimes Q_n, a_n, b_2)\}$. After assembly reduction - phase one, we get: $\sigma = \{(S, Q, a, b)\}$ such that: $a = \sum_{i=1}^{n} a_i$, $b = \sum_{i=1}^{n} b_i$, and Q is virtual role for checking only. This is for role S only and must be done for all other roles that have connections to more than one role.

Figure 8 presents CC assembly normal form C. The connection here has the following semantics: at least i instance of S(resp.Q) connect to i instance of Q(resp.S), and at most m instance of S(resp.Q) connect to m instance of Q(resp.S). This is correct for one and only one instance of each cloud component. More over, CCs $ComA$ and $ComB$ have multiplicities $[a..b]$ and $[c..d]$ respectively.

Because of the multiplicities of the CCs, we are unable to use the checking procedure used for normal form A directly on normal form C. To be able to check this assembly, we will follow several assembly reductions starting

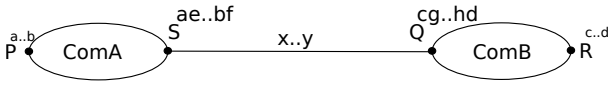

Fig. 9. The result after reduction phase two and three on figure 8- CC multiplicities are completely removed.

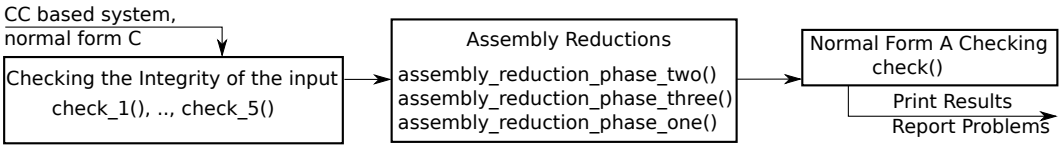

Fig. 10. Inclusive checking algorithm. The integrity checks, namely, check1() through check5(), insure that the input is not corrupted with respect to normal form C.

from this general model. Assembly reduction phase two reduces the multiplicity of the CC to be incorporated (inserted) into the multiplicities of its roles. Formally, let: $\Omega ComA \equiv (\{\Lambda P, \Lambda S\}, \{(\Lambda P, 1), (\Lambda S, e, f)\}, \overline{L}, Z)$ and $\Omega ComB \equiv (\{\Lambda Q, \Lambda R\}, \{(\Lambda Q, g, h), (\Lambda R, 1)\}, \overline{L}, Z)$. $\sigma = \{(\Omega ComA.\Lambda S \otimes \Omega ComB.\Lambda Q, i, m)\}$. Now let $S \equiv (\{\Omega ComA, \Omega ComB\}, \{(\Omega ComA, a, b), (\Omega ComB, c, d)\}, \sigma, \overline{L})$. Assembly reduction phase two produces the new multiplicities for all roles: $\overline{\mu_{ComA}} = \{(\Lambda P, a, b), (\Lambda S, ae, bf)\}$ and $\overline{\mu_{ComB}} = \{(\Lambda Q, cg, hd), (\Lambda R, c, d)\}$.

Assembly reduction phase three is trickier. The multiplicity of the connection $[i..m]$ is affected by both CC's multiplicities, namely $[a..b]$ and $[c..d]$. The objective of this phase is to end up with the connection multiplicity $[x..y]$ using the rule: 'for x we choose the max of the mins, and for y we choose the max of the maxs'. Formally: $x = max\{ia,\ ic\}$, and $y = max\{mb,\ md\}$. By the end of this phase we will get back to normal form A that can be checked directly as in figure 9.

The algorithm in figure 10 is fully implemented using C programming language, and used to check the banking system example in appendix C.

5 Conclusion and Future Work

Highly distributed environments pose a number of challenges for software development process. In this paper we propose a novel component model, the cloud component, that converts the above mentioned challenges into regular and systematic software development details and tasks. Moreover, we proposed a formal notation to describes our component model. This formal notation is more compact, more precise, and less ambiguous. The formal notation allows us to communicate precise details easier, and allows us to easily present statements and proofs. Finally, we developed a formal model to build large systems using CCs as building blocks, and developed an algorithm to check the validity of the assembly, and implemented an automatic assembly checker based on that algorithm.

Several remaining challenges form the scope of our future work. The most important challenge is related to the deployment environment modeling, and designing an effective algorithm to be the basis of an automatic deployment checker that checks the compatibility between cloud components and the actual environment where we are trying to deploy. We have investigated techniques to accomplish this task, these techniques include Ontology and F-Logic (Object Logic) for modeling. Moreover, we investigated several algorithms that depend on reasoning based queries for automatic deployment checker.

References

[1] Cardelli, L., *Abstractions for mobile computation*, Secure Internet Programming, Security Issues for Mobile and Distributed Objects - Lecture Notes in Computer Science **1603** (1999).

[2] Cariou, E., A. Beugnard and J.-M. Jézéquel, *An architecture and a process for implementing distributed collaborations*, in: *6th International Enterprise Distributed Object Computing Conference EDOC 2002, Lausanne, Switzerland, Proceedings* (2002), pp. 132–143.

[3] Crnkovic, I., A. Vulgarakis and M. Chaudron, *A classification framework for software component models*, Software Engineering, IEEE Transactions on (2010).

[4] France, R. and B. Rumpe, *Model-driven development of complex software: A research roadmap*, in: *International Conference on Software Engineering, ISCE, Workshop on the Future of Software Engineering, FOSE, Minneapolis, USA*, IEEE Computer Society, 2007, pp. 37–54.

[5] Hoareau, D., "Composants ubiquitaires pour reseaux dynamiques," Ph.D. thesis, Universite de Bretagne Sud (2007).

[6] Hoareau, D. and Y. Mahéo, *Distribution of a hierarchical component in a non-connected environment*, in: *31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA), Porto, Portugal*, 2005.

[7] Hoareau, D. and Y. Mahéo, *Middleware support for the deployment of ubiquitous software components*, Personal and Ubiquitous Computing **12** (2008), pp. 167–178.

[8] Hourdin, V., J. Tigli, S. Lavirotte, G. Rey and M. Riveill, *SLCA, composite services for ubiquitous computing*, in: *Proceedings of the International Conference on Mobile Technology, Applications, and Systems* (2008).

[9] Lau, K. and Z. Wang;, *Software component models*, Software Engineering, IEEE Transactions on **33** (2007), pp. 709–724.

[10] Malek, S., G. Edwards, Y. Brun, H. Tajalli, J. Garcia, I. Krka, N. Medvidovic, M. Mikic-Rakic and G. Sukhatme, *An architecture-driven software mobility framework*, Journal of Systems and Software **83** (2010), pp. 972–989.

[11] Matougui, S. and A. Beugnard, *Two ways of implementing software connections among distributed components*, in: *On the Move to Meaningful Internet Systems, OTM Confederated International Conferences, Agia Napa, Cyprus, Proceedings, Part II*, 2005.

[12] Mikic-Rakic, M., "Software architectural support for disconnected operation in distributed environments," Phd dissertation, University of Southern California (2004).

[13] Mikic-Rakic, M. and N. Medvidovic, *Architecture-level support for software component deployment in resource constrained environments*, Component Deployment (2002), pp. 493–502.

[14] Mikic-Rakic, M. and N. Medvidovic, *Software architectural support for disconnected operation in highly distributed environments*, Component-Based Software Engineering (2004), pp. 23–39.

[15] Tibermacine, C., D. Hoareau and R. Kadri, *Enforcing architecture and deployment constraints of distributed component-based software*, Fundamental Approaches to Software Engineering (2007), pp. 140–154.

A Current component models limitations

After analyzing several component technologies, we found that they follow a common paradigm. These component models rely on strong assumptions, and they emulate local call on top of distributed networks, and finally they consider any deviation from their implicit or explicit assumption as exceptions. All of these points are considered limitations with regards to HDEs as we explain in the following:

Rely on strong assumption A common way to distribute a component-based application consists of installing each component instance on a host; the distribution then refers to the fact that a component can make distant invocations to the services implemented by another component [6,11]. This type of architecture usually relies on rather strong assumptions [6]:

(i) The stability of the execution platforms (the component server is highly available - usually with backup recovery system)

(ii) All hosts have sufficient resources which include processing power, memory, and power supply.

(iii) The connectivity is reliable and has good characteristics (low latency, enough bandwidth, stable, no disconnections, etc.).

In general, an application designed using this architecture can not be installed and executed on deployment environments with hosts that are potentially volatile and limited in resources, especially when disconnected network operation and weak consistency of the characteristics of the connectivity are possible or frequent, which is the case in HDEs [6,7].

Emulation The distributed component models mentioned above share a common goal: making aspects related to the distribution transparent to both the application programmer and the users. They hide distribution by making remote call appears to the caller as local call, but to some ad-hoc and limited exceptions (see next point). However, by hiding distribution, these mechanisms do not incorporate aspects related to disconnections, mobility, and all other complexities mentioned in the previous section [8]. In general, distributed applications are designed using the same techniques as a centralized application [5,15].

Exceptions Most common component technologies were not originally designed for HDE. Therefore, they consider any deviation from the strong assumptions mentioned above such as inaccessibility of a machine or the unavailability of certain resources as exceptions. The treatment of the various changes that may occur within the network is usually done by adding code to adapt to these new events. This code will increase the complexity of applications [5,15] with specific and ad-hoc extensions and poor methodological guidelines.

Typical HDE applications are highly distributed, decentralized, and mobile. Therefore, they are *highly dependent on the underlying network* [14,13,12]. We believe that the successful paradigm in stable distributed networks *'remote communication'* or *'distribution transparency'* is no more dependable in highly distributed environments HDE. There is fundamental need to move from *hiding* the underlying network into *acknowledging* all its aspects and details. It is possible to achieve that by introducing the concept *'location'*. We call this a *paradigm shift from 'distribution transparency' to 'localization'.*

By location we mean the physical actual computing device where software runs, and that ranges from simple mobile phone to large super computing machine. Modeling location at early design stages will better reflect communication, mobility, and heterogeneity of devices.

Instead of delaying the distribution of software components over the computing devices until the deployment phase, we propose integrating this concept to the very early stages of software development process, especially architecture. Mapping software components to the deployment environment is called *'localization'*. Several models and techniques are proposed in this work to facilitate this approach. The localization of components is revised and refined during the development process until we reach the final deployment plan of the whole application.

It is clear that when we attach location property to a software component, we – either explicitly or implicitly – attach information related to the resources available in this location, the communication paradigms available, the power supply type, and the security features, etc. This information will help (guide) the design and implementation of the system itself.

Acknowledging the properties of the target infrastructure at the very early stages of software development process will help us develop customized software for that infrastructure. This software will utilize the resources to the maximum or near maximum, and at the same time will tolerate the weaknesses and treat the previously considered exceptions as survivable expected events.

B Symbols

Cloud Components and CC based systems can be described using formal/mathematical notation. In this appendix we present the elements and symbols of this notation/language in table B.1.

C Example - Banking System

In this appendix we presnt a simple banking system to explain the algoritm proposed in section 4. The banking example is presented in figure C.1. The $*$

Concept	**Symbol**	**Comments**
Start Symbol	S	Usually the complete system we want to model
Cloud Component	Ω	ΩA is read 'cloud component A'
Roles	Λ	Type ; ΛR is read 'role R"
	Γ	Instance ; ΓR is read 'instantiated role R'
Cardinality	K	
Role Multiplicity	μ	
CC Multiplicity	M	
Location - Type	T	
Location - Host	H	
Localized at	$\downarrow$	
CC Assembly	σ	The set of assembly rules
Connect Operator	$\otimes$	A binding between two roles
Set of	$\overline{symbol}$	Set of CCs $\overline{\Omega}$, set of roles $\overline{\Lambda}$, etc.
Define	$\equiv$	

Table B.1
Symbols used to construct the formal notation.

symbol can be reduced to $[0..MAXINT]$ for computations. In this example the system is built using three CCs. The *Bank* CC is responsible for all database systems, security, transactions, and accounts. It is basically the backbone of the system. The *Agent* CC is the filter that any access to *Bank* will pass through. In other words, nobody can directly access *Bank* CC. *ATM* CC is installed over all ATM machines to allow customers to access their accounts, and perform bank transactions. Similarly, *Internet* CC is installed on the customers devices to allow them to access their accounts using internet banking.

We encoded this example using the formal language presented in this paper, and used the automatic assembly checker to check the design. The assembly checker generated the output presented in figure C.2.

The checker reports expected warnings and errors. For instance the error reported (figure C.2 - right) on the system described figure C.1 (bottom) is due to the too many instantiations of the ATM and Internet CCs.

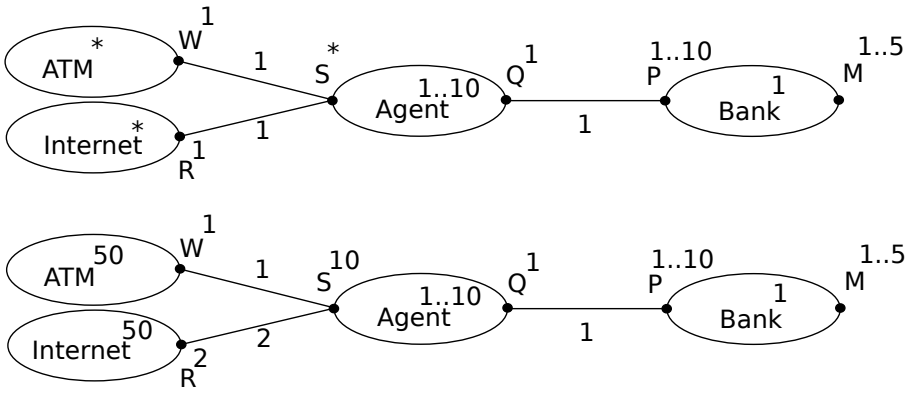

Fig. C.1. The banking system in normal form C. Up: Enterprise Edition. Down: Limited Edition

```
Warning!!!!
Problem type 1.2
Conncetion to P.
This connection is not safe because it is dependent on the max kardinality.
Warning!!!!
Problem type 1.2
Conncetion to Q.
This connection is not safe because it is dependent on the max kardinality.

The design has potential problems. Please read messages.
```

```
Warning!!!!
Problem type 1.2
Conncetion to P.
This connection is not safe because it is dependent on the max kardinality.
Warning!!!!
Problem type 1.2
Conncetion to Q.
This connection is not safe because it is dependent on the max kardinality.
Error!!!! Conncetion to S.
This connection is illegal because it is exceeds the max kardinality.

The design has major errors. Please read messages.
```

Fig. C.2. The output generated by the assembly checker. Up: For Enterprise Edition. Down: For Limited Edition