A distributed and mobile component system based on the ambient calculus
Mylonakis, N. and Orejas, F.
Research Report LSI-04-35-R
A distributed and mobile component system
based on the ambient calculus

Nikos Mylonakis and Fernando Orejas

ISI Department
Universitat Politecnica de Catalunya

Abstract. In this paper we present a new component concept equivalent to the one of [2] which is more appropriate for distributed applications. After that, we present the notion of component system and define a set of operations of component systems which some of them are used to define an ambient calculus [1] with component systems. Finally we present an example.

1 Introduction

The final aim of this work is the study of the modelling and development of component-based distributed applications with mobile processes in the internet. In particular, we believe that the concept of component can play an important role in the development of such applications where components can be defined as independent units with a specific task or functionality. These systems are in general heterogeneous which means that they can be described or implemented using different formalisms. To develop these systems, we use a uniform notion of component which is, to a certain extent, independent of the formalism which is used. The framework which we define is based on a generic notion of transformation or refinement which is used to define the semantics of components and their interconnection. In particular, in [2], the conditions which must satisfy these transformations in order to instantiate the given framework to a concrete formalism are presented. Additionally, a simple notion of composition of components is given. In our work, we define a new semantics of components which is a variation of the one defined in [2] which make possible the definition of more complex forms of composition.

In this work we also introduce the notion of component system. A component system is a set of components where every component can have several import interfaces and several export interfaces. Additionally, we can connect an import interface of a component with an export interface of another component. A component system is in general a graph. We define also different operations on component systems. The most relevant is a non-deterministic operation to make some or all the possible connections between two component systems. In order to develop distributed and mobile applications in the internet component systems are not enough. A possible extension of component systems to achieve this goal is to incorporate component systems in a well-established calculus for distribution and mobility like the ambient calculus [1]. In order to extend component
systems with the ambient calculus, it is necessary to define a forest of hierarchies of component systems. Thus, an ambient expression will denote a forest of hierarchies of component systems and after making a move, a sub-hierarchy of a hierarchy will be associated to another hierarchy in the forest. Every component system in the hierarchy can have associated a forest of sub-hierarchies. The main modification in the calculus is the possibility to have a component system as a process and the obligation to have a component system associated to the name of an ambient.

Finally, we present an example of a server with a firewall together with two clients trying to obtain a software component to finish an application. To present the example, we instantiate our component system with a formalism based on algebra transformation systems.

The structure of this paper is the following: in the first section we present our component concept reviewing the notion of the transformation framework in [2]. In the next section we present our component system and interconnected forest of hierarchies of component systems. After that we present the extension of the ambient calculus with component system and finally we instantiate our component concept with a formalism based on algebra transformation system and we present an example.

2 The component concept

As we mentioned in the introduction, components can be seen as independent units with a specific task or functionality which can be used in as many environments and applications as possible [3]. To achieve this, in a component there must be a clear distinction between its body, where the functionality of the component is described (or implemented) in detail, and its interfaces, describing how the component is related to the outside world.

In [2], a generic component framework is presented. This framework is generic not only concerning the underlying specification formalism used inside the components, but also concerning the concept of transformations in order to model abstraction and refinement between interfaces and body of one component, or between import and export interfaces between different components. The only requirement for a specification formalism to be used in connection to our component framework is to show that the formalism is a transformation framework. More precisely:

**Definition 1.** A transformation framework $T$ over a class of specifications consists of a class of transformations and a subclass of embeddings, which include identical transformations and identical embeddings, both are closed under composition and satisfy the following properties:

- For each transformation $t_1 : SPEC_1 \Rightarrow SPEC_3$, and each embedding $i_1 : SPEC_1 \hookrightarrow SPEC_2$ which can be represented as in Figure 1, there is a selected transformation $t_2 : SPEC_3 \Rightarrow SPEC_4$, with embedding $i_2 : SPEC_2 \hookrightarrow SPEC_4$, called the extension of $t_1$ with respect to $i_1$. This
\[
\text{SPEC}_1 \xrightarrow{i_1} \text{SPEC}_2 \quad \text{SPEC}_3 \xrightarrow{i_2} \text{SPEC}_4
\]

**Fig. 1. Extension Diagram**

...extension property can be represented with diagram 2. \( t_2 \) is also denoted as \( E_{\text{int}}(t_1) \).

\[
\text{SPEC}_1 \xrightarrow{i_1} \text{SPEC}_2 \quad \text{SPEC}_3 \xrightarrow{i_2} \text{SPEC}_4
\]

**Fig. 2. Extension Diagram**

- Horizontal and vertical composition of extension diagrams are required in the usual way.

In this paper, we consider that components may have several import and export interfaces allowing us to connect a given component with several other components. In particular, export interfaces describe services that a component offers to the outside world, while import interfaces specify services used inside a component that are assumed to be defined by other components. Export interfaces are assumed to be different abstractions of the body and each of them is related to the body by a transformation modelling a refinement. Import interfaces are assumed to be related to the body by independent embeddings. More precisely:

**Definition 2.** A component consists of a body specification with a list of independent import specifications together with the corresponding embeddings and a list of export specifications together with the corresponding transformations into the body specifications. Thus, a component will have this general form

\[
(BOD, <IMP_1, i_1, \ldots, IMP_n, i_n>, <EXP_1, e_1, \ldots, EXP_m, e_m>)
\]

A possible graphical representation is in figure 3.

As said above, we assume that the family of embeddings \( i_j : I_j \to BOD \) for each connector is independent. This means intuitively that the import interfaces \( IMP_j \) of \( B \) are pairwise disjoint. Formally, independence of embeddings can be expressed by the following definition:
Definition 3. A family of embeddings $i_j: SPEC_j \rightarrow SPEC,(j = 1..n)$ are independent if the following properties are satisfied:

- For every family of transformations $t_j \in \text{Trafo}(SPEC_j),(j = 1..n)$, there exists a selected transformation $t \in \text{Trafo}(SPEC)$ and selected independent embeddings $i_j', i_k'$ such that the diagram Fig. 4 commutes.

$t$ is called the parallel extension of $\{t_j\}_{j=1..n}$ with respect to $\{i_j\}_{j=1..n}$ and will be denoted as $\text{PE}_{\{i_j\}_{j=1..n}}(\{t_j\}_{j=1..n})$

- For any $SPEC_j, 1 \leq j \leq n$, given the extension diagram of figure 5 and any $SPEC_{k}, 1 \leq k \leq n$, we have that the diagram in figure 6 is a parallel extension diagram, where $i''_k$ is the composition of $i_k$ and $t''$. Note that, in this case, we are asking that the composition of the embedding $i_k$ and the transformation $t''$ should be an embedding.

- Parallel extension diagrams can be composed vertically.

Now, in this paper we provide a new semantics to components which can be seen as a variation on the semantics defined in [2]. In particular, in this paper
we consider that the semantics of a component is defined as a class of tuples of transformations (refinements) of the export interfaces that have to satisfy some constraints. On one hand, these transformations must be defined as the composition of the corresponding export connections $e_k$ and a given transformation $t$ which is a refinement of the body specification. On the other hand, $t$ must be a parallel extension of some family of refinements of the import specifications. More precisely:

Definition 4. The semantics of a component $COMP$:

$$(BOD, <IMP_1, i_1, \ldots, IMP_n, i_n >, <EXP_1, e_1, \ldots, EXP_m, e_m >)$$

is defined as follows:

$\text{TrafoSem}(COMP) = \{ <t \circ e_1, \ldots, t \circ e_m > \mid t \in \text{Trafo}(BOD) \text{ and } \exists t_1 \in \text{Trafo}(IMP_1), \ldots, t_n \in \text{Trafo}(IMP_n) \text{ s.t. } t = PE_{\{i_j\}_{j=1..n}}(\{t_j\}_{j=1..n})\}$$

where $\text{Trafo}(SPEC)$ denotes the class of all transformations defined over $SPEC$.

3 Component systems

A component system is a set of components together with a set of connections. More precisely, every component $j$ can have $m_j$ import interfaces and $p_j$ export interfaces, and we can connect an import interface of a component $j$ with an export interface of a component $k$ via a transformation $e_{jk}$. We will assume that every component has a name. More precisely:

Definition 5. A component system $CS$ consists of a list of components and a list of connections:

$$CS = (L_{Components}, L_{Conns})$$
where each component in \( L_{\text{Components}} \) has the form:

\[
C : (BOD^C, <IMP^C_i, i_1, \ldots, IMP^C_{n_c}, i_{n_c}>, <EXP^C_1, e_1, \ldots, EXP^C_{m_c}, e_{m_c}>)
\]

where \( C \) is the name of the component, and each connection in \( L_{\text{Conns}} \) is a transformation:

\[
\text{conn}_{C,C'} : IMP^C_j \Rightarrow EXP^{C'}_k
\]

where \( IMP^C_j \) is an import interface of the component \( C \) and \( EXP^{C'}_k \) is an export interface of the component \( C' \).

As in the case of single components, we define the semantics of a component system as a set of tuples of transformations of the export specifications in the system that satisfy a given set of constraints. In particular, given a refinement for each import specification of the given system, we require that the transformation associated to a given export specification should be the composition of the corresponding export connection and the parallel extension of the import refinements. This means that these transformations must belong to the semantics of the components of the system. On the other hand, each connection, \( \text{conn}_{C,C'} : IMP^C_j \Rightarrow EXP^{C'}_k \), induces the additional constraint that the transformation associated to \( EXP^{C'}_k \) composed with \( \text{conn}_{C,C'} \) coincides with the given refinement of \( IMP^C_j \). More precisely:

**Definition 6.** Given a component system \( CS = (L_{\text{Components}}, L_{\text{Conns}}) \), where \( L_{\text{Components}} \) consists of components:

\[
C_j : (BOD^{C_j}, <IMP^{C_j,i_1}_{n_{c,j}}, \ldots, IMP^{C_j}_{n_{c,j}}, t^{C_j}_{n_{c,j}}>, <EXP^{C_j}_1, e^{C_j}_1, \ldots, EXP^{C_j}_{m_{c,j}}, e^{C_j}_{m_{c,j}}>)
\]

with \( 1 \leq j \leq p \) and \( L_{\text{Conns}} \) consists of connections:

\[
\text{conn}_{C_{j_1},C_{j_2}} : IMP^{C_{j_1}}_{k_1} \Rightarrow EXP^{C_{j_2}}_{k_2}
\]

where \( C_{j_1} \) and \( C_{j_2} \) are components in \( L_{\text{Components}} \), the semantics of \( CS \) is defined as:

\[
\text{TrafoSem}(CS) = \{ t^{C_j_1} \circ e^{C_j_1} \circ \ldots \circ t^{C_j_{n_{c,j_1}}} \circ e^{C_j_{n_{c,j_1}}} \circ \ldots \circ t^{C_j_{n_{c,j}}} \circ e^{C_j_{n_{c,j}}} \mid
\forall j (1 \leq j \leq p) \forall k (1 \leq k \leq n_{c,j}) \exists \{^{C_j}_{k} \in \text{Trafo}(IMP^C_j) \text{ such that}
\forall j (1 \leq j \leq p) \forall k (1 \leq k \leq n_{c,j}) \exists \{^{C_j}_{k} \in \text{Trafo}(IMP^C_j) \text{ such that}
\}
\]

\[
\forall \text{conn}_{C_{j_1},C_{j_2}} \in L_{\text{Conns}}(t^{C_{j_2}} \circ e^{C_{j_2}} \circ \text{conn}_{C_{j_1},C_{j_2}} = t^{C_{j_1}})
\]

In the rest of this section, we define some operations on component systems. We present an operation to connect two component systems by a single connection, an operation to add a connection to a component system and an operation to make non-deterministically some or all of the connections between two component systems.

To define these operations, we need the following definition:
Definition 7. Given a component system $CS$, we denote by $\text{Comp}(CS)$ the set of components of $CS$ and $\text{Conn}(CS)$, the set of connections.

Now we define the semantics of the operations on component systems.

Definition 8. If $C_1$ and $C_2$ are components in $CS_1$ and $CS_2$, respectively, then the operation to connect $CS_1$ and $CS_2$:

$$CS_1 \circ_{\text{con}} IMPC_1 \Rightarrow EXPc_2 CS_2$$

defines the following component system:

$$(\text{Comp}(CS_1) \cup \text{Comp}(CS_2), \text{Conn}(CS_1) \cup \text{Conn}(CS_2) \cup \{\text{con}\})$$

Definition 9. If $C_1$ and $C_2$ are components in $CS$, then the operation to add a connection to $CS$

$$\text{add}_\text{con}(\text{con} : IMPC_1 \Rightarrow EXPc_2, CS)$$

defines the following component system:

$$(\text{Comp}(CS), \text{Conn}(CS) \cup \{\text{con}\})$$

The next definition is required for the definition of the next operation:

Definition 10. Given two specifications $SP_1, SP_2$, $\text{Trans}(SP_1, SP_2)$ holds if and only if there exist a transformation $t : SP_1 \Rightarrow SP_2$.

Definition 11. The non-deterministic operation to make some or all the connections between two component systems

$$\text{make}_\text{con}(CS_1, CS_2)$$

with arbitrary set $CON$ such that

$$\text{con} : IMPC_1 \Rightarrow EXPc_2 \in CON \iff C_1 \in \text{Comp}(CS_1) \land C_2 \in \text{Comp}(CS_2) \land \text{Trans}(IMPC_1, \text{EXPc}_2))$$

defines the following component system:

$$(\text{Comp}(CS_1) \cup \text{Comp}(CS_2), \text{Conn}(CS_1) \cup \text{Conn}(CS_2) \cup \text{CON})$$

In order to extend the component system with the ambient calculus, it is necessary to define an interconnected forest of hierarchies of component systems. Thus, an ambient expression will denote an interconnected forest of hierarchies of component systems. Every component system in the hierarchy can have associated a forest of sub-hierarchies. This forest of sub-hierarchies can be infinite. The interconnections among the component systems of the forest can be made
autonomously by the component systems. The interconnections among component systems of the forest which we allow are from a component system to its immediate ancestor component system in the hierarchy. For this, we have to use a similar operation to make_con but now for two separate component systems in a hierarchy of component systems. This operation will be called make_con_uh. Additionally, we allow interconnections among component systems of the same hierarchy. For this, we will use the operation make_con_wh.

The operations which we will use in the definition of the forest associated to an expression of the ambient calculus are the following (here we just provide an intuitive description of these operations, although it would not have been difficult to define them formally):

- **empty_fo**ret: Operation which returns the empty interconnected forest.
- **add_fo**rest: Operation which given a component system and an interconnected forest of hierarchies returns a forest with just one hierarchy with root the given component system and the forest of hierarchies as associated sub-hierarchies.
- **union_fo**rest: Operations which given two interconnected forest of hierarchies returns the union of the two interconnected forests.

Now we describe the operations make_con_uh and make_con_wh which we have mentioned above to make interconnections among the component systems of a forest:

- **make_con_uh**: Operation which given two hierarchies and an interconnected forest such that the second hierarchy is a sub-hierarchy of the first hierarchy in the given interconnected forest, performs non-deterministically a set of connections from the component system of the root of the sub-hierarchy to the component system of the root of the hierarchy similar to the operation make_con.

- **make_con_wh**: Operation which given two hierarchies and a interconnected forest such that the two hierarchies are sub-hierarchies with the same root in the given interconnected forest, performs non-deterministically a set of connections from the component system of the root of the first sub-hierarchy to the component system of the root of the second sub-hierarchy similar to the operation make_con.

These two last operations are used in the following operation which is used to define the interconnected forest associated to an ambient calculus expression:

- **make_interconnections**: Operation which apply to all the possible pairs of component systems which can interconnect the operation make_con_uh or make_con_wh adequately.

4 Extension of the component system with the ambient calculus

In this section, we extend the component system with operators of the ambient calculus [1]. We have chosen the ambient calculus because it is a well-established
calculus to describe mobility of processes in a hierarchical physical space of computing sites represented by ambients. This hierarchical space can change over time, having the possibility to move an ambient inside another ambient, to move an ambient out of another ambient and to open an ambient. All these moves change the hierarchical space of computing sites.

The main idea of the extension is to define an ambient calculus where component systems may be joint to the name of an ambient. Thus, in our case, the hierarchical space of computing sites is a forest of component systems.

The calculus of components with ambients is defined as follows:

\[
P := P \parallel Q \mid (m,CS) \mid (m,CS)[P] \mid 1P \\
M.P \mid m.P \\
M := \text{in } n \mid \text{out } n \mid \text{open } n \mid M.M'
\]

As in the definition of the ambient calculus, \( n \) and \( m \) ranges over names, \( P \) and \( Q \) over processes and \( M \) and \( M' \) over capabilities, which are actions to make a move. \( CS \) ranges over component systems.

As we can observe, the rest of the operators are very similar to those of the ambient calculus. We have changed the syntax of the parallel operator \( \parallel \) where in [1] is \( \cdot \). See also [1] for an informal explanation of the rest of the operators, and the formal definition of the function free names which can be extended to our calculus very easily.

As we mentioned in the previous section, an expression of our calculus denotes an interconnected forest of hierarchies of component systems. We define this semantics in two steps. First, a function called \( \text{Forest} \) defines the forest associated to each (sub)expression. However if the expression involves two subsystems, for example in the case of \( P \parallel Q \) the result will be the union of the forests associated to \( P \) and to \( Q \), but it will not include any connection between components from \( P \) and \( Q \). The second step permits the definition of connections between components belonging to different subsystems. In particular, we assume that our components are autonomous agents that can establish connections with other components in a given scope. In this sense, a function called \( \text{make_interconnections} \) is assumed to define, nondeterministically, the interconnections that the component systems can perform autonomously as we described in the previous section.

**Definition 12.** Given an expression of the ambient calculus (\( \text{acxpr} \)), the interconnected forest of hierarchies of component systems associated to this expression is defined as follows:

\[
\text{Intconforest}(\text{acxpr}) = \text{make_interconnections}(\text{Forest}(\text{acxpr}))
\]

where the function \( \text{Forest} \) is inductively defined as follows:
\[
\begin{align*}
\text{Forest}(P \parallel Q) &= \text{union}\_\text{forest}(\text{Forest}(P), \text{Forest}(Q)) \\
\text{Forest}((n,\text{CS})) &= \text{add}\_\text{forest}((n,\text{CS}), \text{empty}\_\text{forest}()) \\
\text{Forest}((n,\text{CS})[Q]) &= \text{add}\_\text{forest}((n,\text{CS}), \text{Forest}(Q)) \\
\text{Forest}(M.P) &= \text{Forest}(P) \\
\text{Forest}(\nu n.P) &= \text{Forest}(P) \\
\text{Forest}(\lambda P) &= \text{union}\_\text{forest}(P, \text{Forest}(\lambda P))
\end{align*}
\]

In the following we explain some of the cases of this definition:

- The forest associated to an expression of the form \( P \parallel Q \) is the union of the forests of \( P \) and \( Q \), and the forest associated to \( \lambda P \) is the infinite forest built as the union of infinite occurrences of the forest associated to \( P \).
- To build the forest associated to the restriction operator \( \nu n.P \) we should assign a fresh name to any occurrence of \( n \) in \( P \). We do not do so because we assume that our ambient expression has no name conflicts because it has been previously renamed appropriately. Otherwise the function \( \text{Forest} \) would be incorrect.
- The forest associated to the expression \( M.P \) is the forest of \( P \) because this function denotes the forest associated to \( P \) before making the actions \( M \). After making the actions \( M \) we will have a different forest.

The definition of the structural congruence between expressions of the new calculus and its operational semantics can be defined in a very similar way as in the original presentation in [1]. We just present the operational semantics which is used in the example of next section. As we can see, this semantics coincides essentially with the original semantics of the ambient calculus:

\[
\begin{align*}
(n,\text{CS})[\text{in } m.P \parallel Q] \parallel (m,\text{CS}')[R] &\rightarrow (m,\text{CS'})[(n,\text{CS})[P] \parallel Q] \parallel R \\
(n,\text{CS})[(n,\text{CS}')[\text{out } m.P \parallel Q] \parallel R] &\rightarrow (n,\text{CS'})[P \parallel Q] \parallel (m,\text{CS})[R] \\
\text{open } m.P \parallel (m,\text{CS})[Q] &\rightarrow P \parallel Q \\
P \rightarrow Q \Rightarrow (\nu n)P \rightarrow (\nu n)Q \\
P \rightarrow Q \Rightarrow (n,\text{CS})[P] \rightarrow (n,\text{CS})[Q] \\
P \rightarrow Q \Rightarrow P \parallel R \rightarrow Q \parallel R \\
p' \equiv P, P \rightarrow Q, Q \equiv Q \Rightarrow P' \rightarrow Q'
\end{align*}
\]

5 Example

The example that we present is about a server with a firewall together with two clients trying to obtain a software component. This example is based on an example of [1].

The formalism which we use for the specification of components are algebra transformation systems as in [4]. Thus, specifications denote a class of computations where we have states which are represented by \( \Sigma \)-algebras and computation steps are partial functions from \( \Sigma \)-algebras to \( \Sigma \)-algebras.
We assume predefined the component systems CS and CSPROD which respectively denote the component system of the client to be finished and the component system of the server which the client needs to finish an application. Additionally, we will use the component systems CSCL1, CSCL1K, CSCL1K", CSSR, CSSRK and CSCL2.

We will give some details of the definition of the component system of the server CSSR. This component system will consist of two components: one defining the proper server and another defining a firewall. The server component has two import algebra transformation systems: one from a client and another from the firewall. The whole component system is defined as follows:

$$(CSSR, < CSR : (BOD^{est}, < IMP_1^{est}, \epsilon_1^{est}, IMP_2^{est}, \epsilon_2^{est} >, < EXP_1^{est}, \epsilon_1^{est} >), CFW : (BOD^{fw}, <>, < EXP_1^{fw}, \epsilon_1^{fw} >) >, < c_{carf_/fw} : IMP_1^{est} \rightarrow EXP_1^{fw} >)$$

We will denote the component system of the server CSSRG adding a G at the end of the name of the component system. We will proceed in the same way with the other component systems which we will define.

The attributes of the body of the server will include the following:

- **CPU, memory, threads, adm.domains**: integer
- **sockets**: list[queue[cl descr]]
- **queue.sentag**: queue[agentid plus descr]
- **queue.recvag**: queue[agentid plus descr]
- **queue.sentprod**: queue[agentid plus descr]

And the attributes imported by the firewall component will include the following:

- **client list**: list[client address]
- **socket.fw**: list[queue[cl descr]]
- **queue_recv_server**: queue[agentid plus descr]
- **queue_recv_cli**: queue[agentid plus descr]
- **queue_recv_prod**: queue[agentid plus descr]

**Comments:**

- CPU, memory, threads and adm.domains are the resources which the server has to assign to incoming mobile agents from clients.
- Sockets in Application Program Interfaces of protocol software for client-server applications are implemented in a similar way than operating systems implement I/O operations to transfer data to or from a file. In both cases the concept of descriptor is used. Because of lack of space we will not describe the main operations of sockets.
- The abstract data types of the attributes used standard parameterized specifications of lists and queues. The queues of agentid plus descr are used to enqueue agents to be sent or to be received. The lists of queues are lists of...
sockets which can have different clients associated to the socket. The list of client addresses are the client addresses from where the firewall accepts connection.

- We also use the specification of client descriptor (cl descr), client address (client address) and agent identification and descriptor (agent id, plus descr) which we do not give details.

The algebra transformation system of the body of the server will include computation steps to make the following algebra transformations:

- to initialize the values of the server.
- to create a new socket for the server.
- to accept a client to a socket of the server.
- to assign resources to an agent of a client.
- to enqueue an agent in the first queue of the server to send the agent to a client accepted in the socket of the server.
- to dequeue an agent in the first queue of the server and enqueue the agent in the first queue of the firewall.

The two last computation steps can be defined as follows:

- If ag is an agent plus descr such that the descriptor of ag belongs to the attribute sockets of the server in the given state algebra, then this computation step enqueues the agent in the attribute queue sentag of the server of the given state algebra.
- If the attribute queue sentag of the server of the given state algebra is not empty, this computation rule dequeues an agent of the attribute queue sentag of the server, and enqueues the obtained agent in the attribute queue recv client of the firewall.

The component system CSSRK, which is the component system associated to the agent of the server, is defined as follows:

\[(CSSRK, (< CSRK : (BOD^{CSSRK}, < IMP^{CSSRK}_1, i^{sink}_1 >, <>) >, <>))\]

The import algebra transformation system will include an operation to assign resources which is connected to the equivalent operation of the export algebra transformation system of the server component of the server component system which we have already mentioned.

The component system CSCL1 is defined as follows:

\[(CSCL1, (< CCL1 : (BOD^{CCL1}, <>,<>
< IMP^{CCL1}_1, i^{sink}_1, EXP^{CCL1}_2, i^{sink}_2 >, <>) >, <>))\]

The export algebra transformation system includes the operations of the import algebra transformation system of the server component plus an operation to assign resources to incoming mobile agents.
The component system CSCL1K' is the component system associated to the agent of the client and it is defined as follows:

\[(CSCL1KL', (\langle BOD_{CSCL1KL'}, IMP_{CSCL1KL'}, i_{CSCL1KL'}^{elk'} \rangle, \\
< EXF_{CSCL1KL'}, i_{CSCL1KL'}^{eslk'} \rangle, <>))\]

The import algebra transformation system will also include an operation to assign resources which is connected to the equivalent operation in the export algebra transformation system of the component of the client component system. The component system CSCL1K'' is defined as follows:

\[(CSCL1KL'', (\langle BOD_{CSCL1KL''}, IMP_{CSCL1KL''}, i_{CSCL1KL''}^{elk''} \rangle, \\
< EXF_{CSCL1KL''}, i_{CSCL1KL''}^{eslk''} \rangle, <>))\]

The component system CSCL2 is defined as follows:

\[(CSCL2, (\langle BOD_{CSCL2}, IMP_{CSCL2}, i_{CSCL2}^{elk2}, IMP_{CSCL2}, i_{CSCL2}^{eslk2}, \\
< EXF_{CSCL2}, i_{CSCL2}^{eslk2} \rangle, <>))\]

The expression of the first client (CLIENT1) is the following:

\[(CSCL1, CSCL1G)[(CSCL1KL', CSCL1KL'G)[
\text{open } CSSRK.(CSCL1KL'', CSCL1KL''G)[\text{open } prod]] \parallel \text{open } CSCL1KL''.CS]\]

The expression of the second client (CLIENT2) is the following:

\[(CSCL2, CSCL2G)[(\text{agent}, (\langle >, < >))][\text{out } CSCL2.in CSSR.R] \parallel S]\]

The expression of the server (SERVER) is the following:

\[(\sigma CSSR)(CSSR.CSSRGG)[
(CSSRK.CSSRKG)[
\text{out } CSSR.in CSCL1.in CSCL1K', out CSCL1.in CSCL1K][\text{open } CSCL1K'.(prod, (\langle >, < >))][
\text{in } CSCL1KL''.out CSSR.in CSCL1.CSOPD]}\]

Initially, we have the two clients and the server in parallel:

\[CLIENT1 \parallel CLIENT2 \parallel SERVER\]

Client2 can not access the server. Client1 can access the server and obtain the software component. In the following, we will see how the protocol works.

In the first sequence of the reductions, a server agent with key CSRK enters client CSCL1. CSCL1 also knows the key CSSR' of the server agent. Making
this move, the component system of the server agent looses his connection with the
global component system of the server, and when entering the client, it
establishes a new connection with the global component of the client. This new
connection will allow the agent of the server to gain resources in the client. The
result expression is the following:

\[
(CSCL1, CSCL1G)([CSSRK, CSSRKG])
\]
\[
in CSCL1' . out CSSR . in CSSR ] ||
\]
\[
(CSCL1K, CSCL1K'G)[
\]
\[
open CSSRK . (CSCL1K", CSCL1K"G) [ open prod ]
\]
\[
open CSCL1K". CS ] || (CSSR)(CSSR, CSSRG)[
\]
\[
open CSCL1K'. (prod, (<>), (<>))
\]
\[
in CSSR . in CSCL1 . CSPROD ] ||
\]
\[
CLIENT2
\]

After making this move, the agent CSSRK enters the ambient with key CSCL1K',
and the resulting expression is the following

\[
(CSCL1, CSCL1G)([CSCL1K', CSCL1K'G])
\]
\[
open CSSRK . (CSCL1K", CSCL1K"G) [ open prod ] ||
\]
\[
(CSCL1K", CSSRKG)[ out CSCL1 . in CSSR ] ||
\]
\[
(CSCL1K", CSSRG)[ open prod ] || (CSSR)(CSSR, CSSRG)[
\]
\[
(CSCL1K'.prod, (<>), (<>))
\]
\[
in CSSR . in CSCL1 . CSPROD ] ||
\]
\[
CLIENT2
\]

In the following sequence of reductions, the agent CSCL1K' of the client
CSCL1 enters the server. Making this move, the global component system of the
agent CSCL1K' looses the connection with the global component system of the
client and when entering the server, it establishes a connection with the global
component system of the server. This connection will allow the agent to gain
resources in the server.

\[
(CSCL1, CSCL1G)[ open CSCL1K", CS ] ||
\]
\[
(CSSR)(CSSR, CSSRG)[(CSCL1K', CSCL1K'G)]
\]
\[
(CSCL1K", CSCL1K"G)[ open prod ] ||
\]
\[
open CSCL1K'.prod, (<>), (<>))
\]
\[
in CSSR . in CSCL1 . CSPROD ] ||
\]
\[
CLIENT2
\]

Finally, the agent CSCL1K' of the client which is now in the server takes the
component and moves it to the client. The final expression is the following:

\[
(CSCL1, CSCL1G)[CS . CSPROD ] ||
\]
\[
(CSSR)(CSSR, CSSRG)] ||
\]
\[
CLIENT2
\]

References

1. Luca Cardelli and Andy Gordon. Mobile ambients. In Foundations of Software


Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Research Reports - 2004

- LSI-04-2-R : *Comparison of Methods to Predict Ozone Concentration*, Orozco, J.
- LSI-04-3-R : *Towards the definition of a taxonomy for the cots product’s market*, Ayala, Claudia P.
- LSI-04-4-R : *Modelling Coalition Formation over Time for Iterative Coalition Games*, Mérkla-Campos, C. and Willmott, S.
- LSI-04-6-R : *An Analysis Pattern for Electronic Marketplaces*, Queralt, A. and Teniente, E.
- LSI-04-7-R : *Exploring Dopamine-Mediated Reward Processing through the Analysis of EEG-Measured Gamma-Band Brain Oscillations*, Vellikko, A. and El-Deredy, W.
- LSI-04-15-R : *Evaluation and symmetrisation of alignments obtained with the Giza++ software*, Lambert, P. and Castell, N.
- LSI-04-16-R : *A note on the use of topology extensions for provoking instability in communication networks*, Blesa, M.J.
- LSI-04-19-R : *Adding Efficient and Reliable Access Paths to the JCF*, Marco, J. and Franch, X.
• LSI-04-20-R: *Exploiting Simple Corporate Memory in Iterative Coalition Games*, Mérida-Campos, C. and Willmott, S.
• LSI-04-22-R: *Complexity issues on bounded restrictive H-coloring*, Díaz, J. and Serna, M. and Thilikos, D.M.
• LSI-04-23-R: *Chromatic number in random scaled sector graphs*, Díaz, J. and Sanwalani, V. and Serna, M. and Spirakis, P.
• LSI-04-24-R: *Bounds on the bisection width for random d-regular graphs*, Díaz, J. and Serna, M. and Wormald, N.C.
• LSI-04-25-R: *Open Source environment to define constraints in route planning for GIS-T*, Pérez, L. and Silveira, A. da M.
• LSI-04-26-R: *A basic repository of operations for the refinement of general ontologies*, de Palol, X.
• LSI-04-27-R: *Tetrahedral mesh subdivision based on underlying volume data*, Rodríguez, L. and Navazo, I. and Vinacua, A.
• LSI-04-28-R: *The Price of Connectedness in Expansions*, Fomin, F.V. and Fraigniaud, P. and Thilikos, D.M.
• LSI-04-29-R: *Smaller kernels for hitting set problems of constant arity*, Nishimura, N. and Ragde, P. and Thilikos, D.M.
• LSI-04-30-R: *Searching Spatial Sense in the Ontological World: Discovering Spatial Objects*, Morocho, V and Pérez, L. and Saltor, F.
• LSI-04-33-R: *Multisided patches*, Pla, N. and Vigo, M. and Cotrina, J.
• LSI-04-34-R: *SVMTool: A general POS tagger generator based on Support Vector Machines*, Giménez, J. and Márquez, LL.
• LSI-04-35-R: *A distributed and mobile component system based on the ambient calculus*, Mylonakis, N. and Orejas, F.

Hardcopies of reports can be ordered from:

Núria Sanchez
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Campus Nord, Mòdul C6
Jordi Girona Salgado, 1-3
03034 Barcelona, Spain
nurias@lsi.upc.es

See also the Departament WWW pages, http://www.lsi.upc.es/