Graph-based Models of Internetworking Systems*

Gianluigi Ferrari, Ugo Montanari, and Emilio Tuosto

Dipartimento di Informatica, Università di Pisa
e-mail: {giangi,ugo,etuosto}@di.unipi.it

Abstract. Graphical notations have been widely accepted as an expressive and intuitive working tool for system specification and design. This paper outlines a declarative approach based on (hyper-)graphs and graph synchronization to deal with the modeling of Wide Area Network applications. This paper aims at contributing to the understanding of crucial issues involved in the specification and design of Wide Area Network systems, as a first step toward the development of software engineering techniques and tools for designing and certificating internetworking systems.

1 Introduction

The problem of supporting the development of highly decentralized applications (from requirement and design to implementation and maintenance) is at the edge of research in software engineering. Several commercial systems (e.g. *Napster*, *Gnutella*) have led to an increasing demand of innovative techniques to model, document and certify the development of applications. On the one hand, the traditional software engineering technologies (e.g. client–server architecture) emphasize an interaction model which is rather different from the interaction model of truly distributed applications. For instance, users of traditional distributed applications can invoke a service regardless of whether the service is local, remote or under the control of a different network authority. Instead, in the context of Wide Area Network applications the *awareness* of network information is crucial for choosing the best services that match user's requirements. Indeed, network awareness can be exploited to provide as much information about the network facilities as possible to designers, aiming at specifying and implementing robust modules. On the other hand, in the next few years evolutionary *middlewares* based on SOAP-XML-UDDI-WSDL will become the standard in software industry. It is interesting to note, however, that some innovative applications (e.g. peer-to-peer) are developed largely "ad hoc", exploiting the traditional client-server interaction model.

* Research supported by European IST FET-GC project AGILE, by MSR Cambridge project NAPI, and by Italian MIUR projects COMETA and NAPOLI.

Hence, independently from the underlying technology, we argue that requirement engineering technologies must support the shift from the client-server interaction model to other interaction models which better accommodate the constraints posed by the new applications. The present paper intends to address this issue.

The Unified Modeling Language (UML) [6, 40] has been widely accepted throughout the software industries and has become the *de facto* standard for specifying the development of software systems. In fact, UML provides a graphical notation to describe both structural and behavioral aspects of systems. In particular class and state diagrams are the fundamental units which allow the designer to specify the behaviour of object-based systems. However, class and state diagrams provide the abstraction to understand method invocation independently from the location of the object. However, as pointed out by Waldo et al. [47], method invocation in a truly distributed application is inherently different from method invocation in a traditional distributed application. A specification technique which ignores such a difference will not support at the right level of abstraction software design pointing out the possible architectural choices in the system under development.

Previous work on the formalization of UML has produced a semantic framework based on *graph transformations* (see [20, 29, 35] and the references therein). The evolution of a UML specification may be understood as a graph transformation. This paper describes a variation of graph transformation semantics which directly supports network awareness. Hence, what is missing in the UML specification can be actually found at the semantic level.

Our approach is based on (hyper-)graphs and local graph synchronization and extends the graphical calculus for mobility introduced in [33]. Hyper-graphs naturally provide the capabilities to describe internetworking systems: edges are used to represent components and nodes model the network environment of components. The sharing of nodes by some edges means that the corresponding components may interact by exploiting network communication infrastructure. Graph synchronization is purely local and it is obtained by the combination of graph rewriting with constraint solving. The intuitive idea is that properties of components are specified as constraints over their local resources. Hence, the local evolutions of components depend on the outcome of a (possibly distributed) constraint satisfaction algorithm.

In other words, graphs and graph synchronization foster a declarative approach by identifying the points where satisfaction of certain properties has a strong impact on behaviours. The key issue of the approach is that components see the network environment as a set of constraints. Then, the declarative specification of service requests to the network yields various kinds of constraints for the graphical calculus. Thus the actual behaviour is the result of a distributed constraint solving algorithm [39, 50].

This paper aims at providing an understanding of some crucial issues of Wide Area Network computing, as a step toward the development of software engineering techniques and tools for the design and certification of such systems. We first discuss some of the difficult issues involved in building Wide Area Network applications, thus delineating the corresponding requirements for software engineering techniques. To present the basic ideas underlying our graphical calculus, we outline an operational framework for the Ambient calculus [12] in terms hypergraphs and hypergraph synchronization. Finally, we delineate a formal methodology that builds over graph synchronization to equip UML with semantical mechanisms to deal with the modelling of Wide Area Network applications.

2 WAN Computing: A Roadmap

Wide-Area Network (WAN) applications have become one of the most important applications in current distributed computing. Indeed, Internet and the World Wide Web are now the primary environment for designing, developing and distributing applications. Network services have now evolved into self-contained components which inter-operate easily with each other by supporting WEB-based access protocols [34]. In addition, network services may adapt themselves to match the particular capabilities of a variety of devices ranging from traditional PCs to Personal Digital Assistants and Mobile Phones having intermittent connectivity to the network. In other words, WAN applications are highly decentralized and dynamically reconfigurable (e.g. network services are assumed to be *linked* to other services to achieve the required functionalities). This section outlines our perspective on the current status of the research on WAN computing by identifying the fundamental concepts and the proper abstractions which are useful in specifying, designing and implementing WAN applications.

Network awareness

Current software technologies emphasize the notion of WEB SERVICE as a key idiom to control design and development of applications. Conceptually, WEB SERVICEs are stand-alone components that reside over the nodes of the network. Each WEB SERVICE has an interface which is network accessible through standard network protocols and describes the interaction capabilities of the service (e.g., the message format). Wide Area Network applications are developed by combining and integrating together WEB SERVICEs, which do not have pre-existing knowledge of how they will interact with each other.

The exploitation of components in a WAN setting raises a number of issues. First, given the heterogeneity of the network environment, components should be highly portable: they should be usable everywhere, provided that certain services actually behave properly (i.e. services are used to adapt components to a variety of infrastructures). Second, security should be ensured in any environment: since components downloaded from different authorities have different security requirements, they should be executed within different run-time environments. Third, dynamic adaptability should be ensured: WAN applications have highly dynamic requirements and they should be able to reconfigure their structure and their components at run-time to respond to dynamic changes of the network environment.

Summing up, a WAN application does not appear as a single integrated computer facility to its users as it is the case of traditional distributed applications. For instance, users of traditional distributed applications can invoke a service regardless of whether the service is local, remote or under the control of a different network authority. Instead, in the WAN setting the *awareness* of network information is crucial for choosing the best services that match user requirements. For instance, users can react to phenomena like network congestion by binding their network devices to different available resources. Similarly, network awareness is exploited by WAN application designers to control resource usages and resource accesses in order to ensure and maintain certain security levels. Finally, network awareness can be exploited to provide as much information about the sources of network exceptions as possible, in order to allow the designer to specify robust exception handlers.

Network awareness is thus the distinguished and novel issue of WAN applications and refers to the explicit ability of dealing with the unpredictable Quality of Service (QoS) properties of the network environment. Here, QoS is meant as a measure of the *non functional* properties of services along multiple dimensions. For instance, *network bandwidth* is a QoS measure for multimedia services. *Timely response* and *security* are other examples of (higher level) QoS measures. In general, the perceived QoS of computations is no longer given by the performance of the WEB servers but rather by the availability of certain resources, by the security level provided, by the flow of network traffic, and so on.

Current distributed technologies allow applications to control network connectivity and resource accesses. A paradigmatic example is provided by the Java programming language through the SOCKET and the SECURITY APIs. Similarly, the Microsoft .NET architecture supplies a programming technology embodying general facilities for handling heterogeneity. As far as security is concerned, *cryptographic* techniques have been exploited to solve several problems related to security of data communications (authentication, secrecy and integrity). Finally, *firewalls* are barriers that administrative domains build to disable the access to some critical services. In this new scenario both final users and WAN application designers put special emphasis on QoS issues.

In general, QoS attributes are special parameters of network services. *Awareness* of these information is crucial for choosing network services to match user requirements. For instance, final users can react to network congestion by binding their network devices to different sites where the requested services are available. Similarly, QoS awareness is exploited by WAN application designers to control resource usage and resource access in order to guarantee and maintain certain security levels and to provide users with differentiated QoS.

The advances in network technologies and the growth of commercial WEB services have prompted questions about suitable mechanisms for providing QoS guarantees. In the last few years, several models have been proposed to meet the demands of QoS. We mention the *Resource Reservation Protocol* (RSVP) [7], *Differentiated Services* [5], *Constraint-based Routing* [45], and we refer to [49] for a detailed discussion of this topic. This stream of research is basically *system-oriented*: it focuses on the lower layers of the Internet protocol stack.

Another significant line of research has dealt with enhancing existing distributed programming middlewares to support QoS features. QoS-aware middlewares allow clients to express their QoS requirements at a higher level of abstraction. In this way the application has good degree of control over QoS without having to deal with low-level details. Examples of QoS-aware middleware are Agilos [36], Mobiware [3], and Globus [25].

At a foundational level, most models exhibit explicit localities to reflect the idea of network awareness, e.g. Ambient calculus [12], KLAIM [18], and Mobile-Unity [38] to cite a few. Roughly speaking, locations fully identify the network environment of a component. The aforementioned approaches have improved the formal understanding of the complex mechanisms underlying network awareness. For instance, the problem of modeling resource access control of highly distributed and autonomous components has been faced by exploiting suitable notions of type [19, 30, 9, 13]. The growing demands on security have led to the development of formal models that allow specification and verification of cryptographic protocols (see [1, 46, 24, 37, 16] to cite a few). Indeed, the real challenge is to formally understand which are the features of an integrated security model for WAN applications. Wide Area Network applications integrate different computing environments having different security requirements. Moreover, the application security policy maker cannot decide with full knowledge of the current state of the application. Any realistic approach will have to identify which portion of the state of the WAN application is potentially relevant and may affect or be affected by security policy decisions. Interestingly, the notion of QoS briefly outlined above may help to investigate the proper trade-off between expressiveness and security concerns (some preliminary results can be found in [17]). However, a foundational model dealing with all these facets of network awareness is still missing.

Mobility

Mobility provides a suitable abstraction to design and implement WAN applications. The main breakthrough is that WAN applications may exchange active units of behavior and not just raw data. The usefulness of mobility emerges when developing both applications for nomadic devices with intermittent access to the network (*physical mobility*), and network services having different access policies (*logical mobility*).

Mobility has produced new design patterns [27] other than the traditional client-server paradigm:

- *Remote Evaluation*: the code is sent for execution to a remote host;
- *Code On-Demand*: the code is downloaded from a remote host to be executed locally;

– *Mobile Agents*: processes can suspend their execution and migrate to new hosts, where they can resume it.

Among these design paradigms, Code On-Demand is probably the most widely used (e.g. Java Applets). The mobile agent paradigm is, instead, the most challenging since:

– an agent, in order to run, needs an *execution environment*, i.e. a server that supplies resources for execution;
– an agent is *autonomous*: it executes independently of the user who created it (*goal driven*);
– an agent is able to detect changes in its operational environment and to act accordingly (*reactivity* and *adaptivity*).

Another interesting feature of mobile agents is the possibility of executing *disconnected operations* [42]: an agent may be remotely executed even if the user (its owner) is not connected; if this is the case, the agent may decide to "sleep" and then periodically try to reestablish the connection with its owner. Conversely, the user, when reconnected, may try to *retract* the agent back home (i.e. instruct the remote agent to return its home site).

In addition to this scenario, *ad hoc networks* [15] allow connection of nomadic devices without a fixed network structure. Finally, the shift from client-server to peer-to-peer architectures (e.g. Napster and Gnutella) has introduced a new pattern for internet interaction where information is shared among distributed components and changes dynamically.

Clearly, a formal characterization of the key concepts involved in the development of mobile applications (e.g. QoS, adaptability, resource discovery) is a major concern from a software engineering perspective.

Programming languages and systems provide basic facilities for mobility. A well known example is the Java programming language. Another interesting example is provided by Oracle [41], which supports access to a database from a mobile device by exploiting a mobile agent paradigm. However, current technologies provide only limited solutions to the general treatment of mobility.

At a foundational level, several process calculi have been developed to gain a more precise understanding of distribution and mobility. We mention the Distributed Join-calculus [26], Klaim [18], the Distributed π-calculus [31], the Ambient calculus [12], the Seal calculus [14], and Nomadic Pict [48]. Other foundational models adopt a logical style toward the analysis of mobility. *MobileUnity* [38] and *MobAdtl* [22] are program logics specifically designed to specify and reason about mobile systems exploiting a Unity-like proof system. Spatial logic [11, 10] allows one to specify properties on both the spatial dimension and the temporal dimension of WAN applications.

Coordination

Wide Area Network applications are highly decentralized and dynamically reconfigurable. Hence, they should be easily scalable in order to manage addition/removal of services, subnetworks and users without requiring to be reconfigured. Coordination is a key concept for modeling and designing WAN applications. Coordination principles separate the computational components from composition modules called *coordinators* which glue together components. Coordinators are therefore the basic mechanism to adapt components to network environment changes, to discover resources, to synchronize activities, and so on. For instance, coordinators are in charge of supporting and monitoring the execution of dynamically loaded modules. Moreover, coordinators are able to observe evolutions, and therefore they may react to an action by modifying themselves. Finally, coordination policies must be programmable to meet the evolving composition demands and to accommodate the design and the implementation of open systems. Two recent examples of coordination middlewares for WAN programming are represented by Jini and .NET Orchestration, proposed by Sun and Microsoft, respectively. The distinction between computation and coordination is also at the basis of the research on software architectures [44].

Many approaches to coordination are based on the Linda model [28] which proposes the structure of *tuple space* as the mechanism to represent the environment of applications. Experimental programming languages and middlewares have been designed following this metaphor [4, 43]. Some preliminary results on defining a discipline for orchestrating WEB SERVICEs are outlined in [2]. The approach is based on the idea of separating WEB SERVICE providers from *contract* mechanisms (also known under the name of *connectors*), which regulate WEB SERVICE coordination. Coordination laws which characterize *transactional* mechanisms in the context of distributed middleware have been presented in [8].

Research about coordination languages and models has improved the formal understanding of dynamically adaptable mobile components. However, the definition of the right level of abstraction coordination and the choice of suitable constructs to program crucial policies such as adaptation, loading and security require further research.

3 Hypergraph Synchronization

This section briefly reviews the notion of *hypergraph* as presented in [33]. First the definition of hypergraph is given, then *hypergraph rewriting systems*, *transitions* and *productions* are introduced. Finally, we give an informal description of how productions are applied to hypergraphs in order to rewrite them.

We assume that $\mathcal{N}$ is an ordered set of nodes. A *hypergraph* has a set of nodes and a set of *hyperedges* connected to the nodes.

In a traditional graph, an edge connects two nodes; instead, a *hyperedge* connects a set of nodes. Intuitively, an edge can be thought of as representing a binary relation between two nodes, while a hyperedge represents a relationships among many nodes. We write $L(x_1, ..., x_n)$ to indicate an edge labeled L connecting nodes x_1,..., x_n. The *rank* of an hyperedge is the number of nodes that it connects, hence we say that the L above has rank n (written as $L : n$) and that L has a tentacle for each x_i; nodes $x_1, ..., x_n$ are the *attachment nodes* (or *attachment points*) of L.

Hypergraphs are described as *syntactic judgments* of the form $\Gamma \vdash G$. In a syntactic judgment, $\Gamma \subseteq \mathcal{N}$ is a set of nodes representing the external interface of the graph, namely the attachment nodes toward the environment. We shall call *external nodes of the graph* the nodes in Γ. Term G is generated by the following grammar

$$G ::= nil \;\Big|\; L(\boldsymbol{x}) \;\Big|\; G|G \;\Big|\; \nu\, y.G.$$

The above productions permits generating the empty graph (represented by *nil*), single edges (using $L(\boldsymbol{x})$) composing terms in parallel (via $G \mid G$) and hiding nodes (through $\nu\ y.G$). The nodes in G which are in the scope of ν operator are called *bound* nodes; let $bn(G)$ and $fn(G)$ respectively denote the set of the bound and *free* nodes of G (the nodes of G which are not bound). A judgment $\Gamma \vdash G$ is *legal* if $fn(G) \subseteq \Gamma$.

Hereafter, hypergraphs, hyperedges or hyperarcs will be simply called graphs, edges, respectively.

A *graph rewriting system*, $\mathcal{G} = \langle \Gamma_0 \vdash G_0, \mathcal{P} \rangle$, consists of a graph and special *transitions* which we call *productions*. A transition is a logical sequent

$$\Gamma_1 \vdash G_1 \xrightarrow{\Lambda,\pi} \Gamma_2 \vdash G_2. \tag{1}$$

where $\Lambda \subseteq \Gamma_1 \times Act \times \mathcal{N}^*$ is a *set of constraints* and $\pi : \Gamma_1 \to \Gamma_1$ is a *fusion substitution*[1]; both Λ and π are detailed below. The set of actions Act is used to model synchronized rewriting. We associate actions to (some of) the nodes of $\Gamma_1 \vdash G_1$, via Λ. In this way, each rewrite of an edge must synchronize its actions with one or more of its adjacent edges; thus, in general, more that one participants will move: the number depends on the synchronization policy.

Transition (1) above rewrites $\Gamma_1 \vdash G_1$ into $\Gamma_2 \vdash G_2$ whenever the set of constraints Λ is satisfied and the fusion substitution π is applied.
If $(x, a, \boldsymbol{y}) \in \Lambda$ then all edges in G_1 that have a tentacle connected to x participate to the synchronization. They must satisfy condition a that will depend on the chosen synchronization algebra. The nodes in $\boldsymbol{y}$ are the nodes of the constraint; we let $n(\Lambda)$ denote the union of all such nodes in Λ.
Let us now consider the structure of the right hand side of sequent (1). $\Gamma_2 = \pi(\Gamma_1) \cup n(\Lambda)$; it consists of the free nodes of G_1 as transformed by π and the new nodes used in the synchronization. In general, G_2 may be any graph whose free nodes are in Γ_2.

We impose two further conditions on transitions, namely we require:

1. $\forall x, y \in \Gamma_1 . \pi(x) = y \Rightarrow \pi(y) = y$, so that π induce a partition on Γ_1, where all nodes in an equivalence class are mapped to a representative element of the class.
2. $n(\Lambda) \cap \Gamma_1 \subseteq \pi(\Gamma_1)$, i.e. the nodes in $fn(G_1)$ used in the synchronization have to be representative elements induced by π.

[1] We often omit the fusion substitution in transitions when it is the identity.

A *production* is a transition of the form

$$set(\boldsymbol{x}) \vdash L(\boldsymbol{x}) \xrightarrow{\Lambda,\pi} \Gamma \vdash G$$

where L is an edge label of rank n and $\boldsymbol{x}$ is a n-tuple of nodes with $set(\boldsymbol{x})$ the set of nodes appearing in $\boldsymbol{x}$.

Synchronized edge replacement is obtained using graph rewriting combined with constraint solving. More specifically, we use *context-free* productions labeled with actions for coordinating the simultaneous application of two or more productions. Coordinated rewriting allows the propagation of synchronization all over the graph where productions are applied. Determining the productions to be synchronized at a given stage corresponds to solving a distributed constraint satisfaction problem [39]. In [32, 33, 21] synchronized graph rewriting has been employed to model mobility. There constraint satisfaction amounts to unification. In the present paper (see Section 4.5) an example is given where the constraints represent shortest path requirements for the routers.

A production rewrites a single edge into an arbitrary graph. A production $p = (L \to R)$ can be applied to a graph G yielding H if there is an occurrence of an edge labeled by L in G. Graph H is obtained from G by removing the previously matched edge and by embedding a fresh copy of R in G by coalescing its external nodes with the corresponding attachment nodes of the replaced edge.

A derivation is obtained by starting from the initial graph and by executing a sequence of transitions, each obtained by synchronizing possibly several productions. The synchronization of a rewriting rule requires matching of the actions and unification of the third components of the constraints Λ. After productions are applied, the unification function is used to obtain the final graph by merging the corresponding nodes.

Given a graph rewriting system $\langle \Gamma_0 \vdash G_0, \mathcal{P} \rangle$, the set $T(\mathcal{P})$ of possible transitions is obtained from the productions $\mathcal{P}$ using four inference rules. We refer to [21] for a complete presentation of our graph rewriting system.

4 The Ambient Calculus

This section outlines the application of the graph synchronization framework to the Ambient calculus. We will not consider the whole calculus, but only a simple fragment since we aim at presenting the key ideas of the approach. The interested reader if referred to [21] for a detailed presentation.

4.1 The calculus

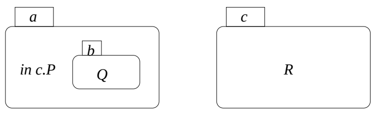
The Ambient calculus relies on the notion of *ambient* that can be thought of as a bounded environment where processes interact. The syntax of Ambient is[2]

$$\begin{aligned} P &::= a[P] \mid P \mid Q \mid M.P \\ M &::= \mathit{in}\ a \mid \mathit{out}\ a \mid \mathit{open}\ a \mid M.M \end{aligned}$$

[2] We consider a fragment of Ambient without communication and restriction.

An *ambient process* is written as $a[P]$ and represents a "place", called a, containing a process P. P is made of the parallel composition of processes and subambients or it is prefixed with a sequence of capabilities M. An example of ambient together with an intuitive graphical representation are given below

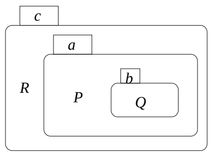
$$a[in\ c.P \mid b[Q]] \mid c[R]$$



Processes exploit *capabilities* to control ambient interactions.

For instance, the in-capability in the previous example, allows the (pilot) process to drive a inside c in accordance with the following reduction

$$a[in\ c.P \mid b[Q]] \mid c[R] \to c[R \mid a[P \mid b[Q]]]$$



Dually, the out-capability makes a process to drive its surrounding ambient a outside the ambient containing a:

$$b[a[out\ b.P \mid Q] \mid R] \to b[R] \mid a[P \mid Q].$$

The semantics of the *open* prefix is defined by the following reduction:

$$open\ a.P \mid a[Q] \to P \mid Q.$$

4.2 Graph representation of Ambient calculus

The Ambient calculus can be casted in our graphical framework preserving the semantics of processes. We do not entirely report the translation which is detailed in [21] and limit our attention to the following translation:

$$\begin{array}{rll}
[\![\ \mathbf{0}\ ]\!]_x & = x \vdash nil & \\
[\![\ a[P]\ ]\!]_x & = x \vdash \nu\ y.(G \mid n(y,x)), & \text{if } \ y \neq x \wedge [\![\ P\ ]\!]_y = y \vdash G \\
[\![\ in\ a.P\ ]\!]_x & = x \vdash L_{in\ a.P}(x) & \\
[\![\ P_1|P_2\ ]\!]_x & = x \vdash G_1 \mid G_2, & \text{if } \ [\![\ P_i\ ]\!]_x = x \vdash G_i, \text{where } i = 1,2.
\end{array}$$

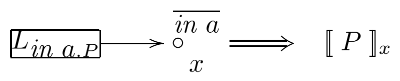
(We have ignored translation of out- and open-capabilities.) The above equations introduce the mapping $[\![\ P\ ]\!]_x$ that returns a graph whose (only) free node x corresponds to the root of the ambient process P.

The first equation defines the translation of the deadlocked process $\mathbf{0}$; its corresponding graph has an isolated node. The graph of $a[P]$ with free node x is obtained by constructing the graph of P on node y, attaching it to the edge $a(y, x)$ and restricting y; note that the ambient name a is interpreted as an edge from y to x labeled a. The capability *in* $a.P$ is directly represented by edges labeled by *in* $a.P$. The parallel composition $P_1 \mid P_2$ is obtained by making the graph of P_1 and P_2 to share their root node x.

4.3 Graph semantics of Ambient calculus

There are two kinds of productions: *activity productions*, and *coordination productions*. The activity productions describe the evolution of sequential processes of the form $M.P$, which, in our approach, become edge labels: when an action is performed, an edge labeled by $M.P$ is rewritten as the graph corresponding to P. For each production, we give both the sequent and its graphical representation. When $(x, \mu, \langle \boldsymbol{y} \rangle) \in \Lambda$, node x in the right member is labeled by $\mu, \langle \boldsymbol{y} \rangle$.

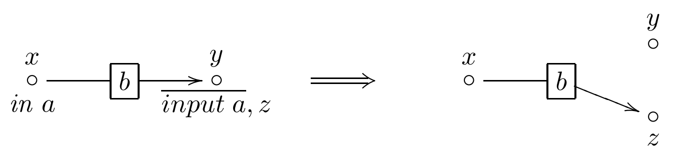
The activity production of an *in* capability has the form



$$x \vdash L_{in\ a.P}(x) \xrightarrow{\{(x,\overline{in\ a},\langle\rangle)\}} [\![P]\!]_x.$$

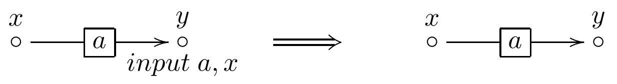
Coordination productions describe ambient interactions. In particular, coordination productions define which are the complementary actions that ambients must perform in order to fire the required synchronization. For instance, the coordination productions for the *in* capability are given as follows.

(*input1*)



$$x, y \vdash b(x, y) \xrightarrow{\{(x,in\ a,\langle\rangle),(y,\overline{input\ a},\langle z\rangle)\}} x, y, z \vdash b(x, z)$$

(*input2*)



$$x, y \vdash a(x, y) \xrightarrow{\{(y,input\ a,\langle x\rangle)\}} x, y \vdash a(x, y)$$

Production (*input1*) asserts that when a process inside b wants to drive b in an ambient a, then the destination of b will become the new node z. Production (*input2*) controls the entrance of an external process inside ambient a: this production simply passes the source x of a to the entering process.

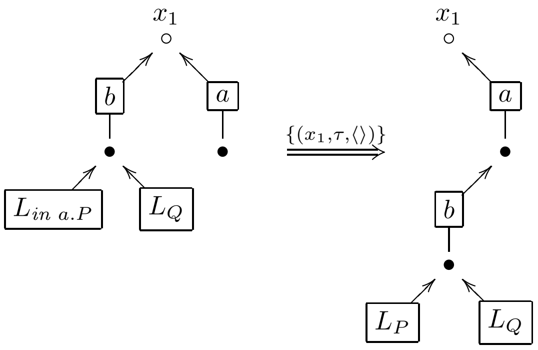
4.4 Example

We now show the correspondence between reductions in the Ambient calculus and the corresponding graph transitions. Let us consider the ambient reduction

$$b[in\ a.P \mid Q] \mid a[\mathbf{0}] \rightarrow a[b[P \mid Q]]$$

where P and Q are sequential processes. Intuitively, a system evolution should be of the form (we represent the restricted nodes with • and the free nodes with

◦)



(2)

The picture on the left is the graphical representation of $[\![\ b[in\ a.P \mid Q] \mid a[\mathbf{0}]\ ]\!]_{x_1}$, while the rightmost picture is $[\![\ a[b[P \mid Q]]\ ]\!]_{x_1}$.

Transition 2 will indeed be the result of a two-stages procedure:

1. the initial graph is decomposed into its constituent edges; for each edge, the productions of the edge are considered,
2. external nodes of each component are fused together in order to obtain the initial graph again; however, in this phase the productions determined in the previous step are synchronized.

First we decompose the graph in its elementary edges and determine the productions that correspond to the elementary components of the transition.

$$x_1, y_1 \vdash b(y_1, x_1) \xrightarrow{\left\{\begin{array}{l}(x_1, \overline{input\ a}, \langle z_1\rangle),\\ (y_1, in\ a, \langle\rangle)\end{array}\right\},\mathrm{id}} x_1, y_1, z_1 \vdash b(y_1, z_1) \tag{3}$$

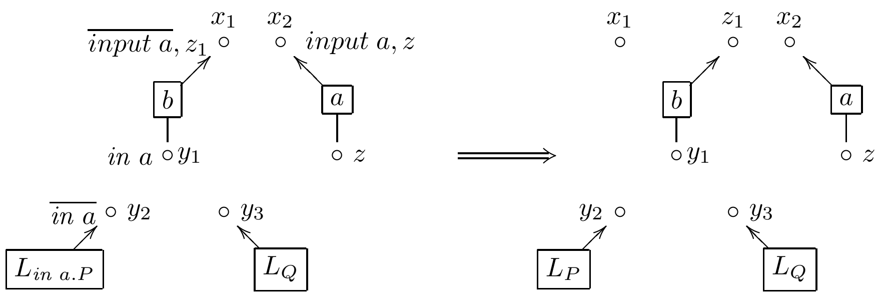
$$y_2 \vdash L_{in\ a.P}(y_2) \xrightarrow{\{(y_2,\overline{in\ a},\langle\rangle)\},\mathrm{id}} y_2 \vdash L_P(y_2) \tag{4}$$

$$x_2, z \vdash a(z, x_2) \xrightarrow{\{(x_2,input\ a,\langle z\rangle)\},\mathrm{id}} x_2, z \vdash a(z, x_2) \tag{5}$$

$$y_3 \vdash L_Q(y_3) \xrightarrow{\emptyset,\mathrm{id}} y_3 \vdash L_Q(y_3) \tag{6}$$

Transitions (3) and (5) are instances of the coordination productions (*input1*) and (*input2*), respectively; transition (4) is the activity production of *in* $a.P$ and transition (6) is the identity transition that leaves L_Q idle.

Graphically, we have:



The previous graph represents the transition obtained by collecting the productions (3), (4), (5) and (6). Let

$$\begin{array}{l} G_1 = b(y_1, x_1) \mid a(z, x_2) \mid L_{in\ a.P}(y_2) \mid L_Q(y_3) \\ G_2 = b(y_1, z_1) \mid a(z, x_2) \mid L_P(y_2) \mid L_Q(y_3) \\ \Gamma \ \ = \{x_1, x_2, y_1, y_2, y_3, z\} \end{array}$$

then, in terms of sequents we have:

$$\Gamma \vdash G_1 \xrightarrow{\left\{\begin{array}{l}(x_1, \overline{input\ a}, \langle z_1 \rangle), \\ (x_2, input\ a, \langle z \rangle) \\ (y_1, in\ a, \langle\rangle) \\ (y_2, \overline{in\ a}, \langle\rangle)\end{array}\right\}, \mathrm{id}} \Gamma, z_1 \vdash G_2 \tag{7}$$

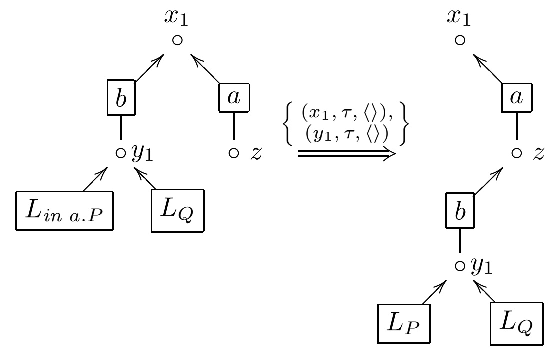
The synchronization of these productions provides the fusion of the nodes in order to obtain a graph of the same shape of the ambient process. Let σ be the function that behaves as the identity on all nodes different from x_2, y_2 and y_3 and

$$\sigma : \begin{cases} x_2 \mapsto x_1 \\ y_2 \mapsto y_1 \\ y_3 \mapsto y_1 \end{cases}$$

that determines $\Lambda' = \{(x_1, \tau, \langle\rangle), (y_1, \tau, \langle\rangle)\}$ and $\rho : z_1 \mapsto z$ and the transition

$$x_1, y_1, z \vdash \sigma(G_1) \xrightarrow{\left\{\begin{array}{l}(x_1, \tau, \langle\rangle), \\ (y_1, \tau, \langle\rangle)\end{array}\right\}, \mathrm{id}} x_1, y_1, z \vdash \rho(\sigma(G_2))$$

that is graphically represented as


x_1
b
a
y_1
z
$L_{in\ a.P}$
L_Q
$\left\{\begin{array}{l}(x_1, \tau, \langle\rangle), \\ (y_1, \tau, \langle\rangle)\end{array}\right\}$
x_1
a
z
b
y_1
L_P
L_Q

We remark that the above transition requires a synchronization involving three edges and two nodes: the edges corresponding to *in* $a.P$ and b (that synchronize on node y_1), and the edges of ambients b and a (that synchronize on node x_1). This makes clear that the *in* capability of ambients requires the synchronization of three components. Finally, transition 2 is obtained by applying a rule which restricts nodes on the left and the right hand side whenever no visible action occurs on them.

We want to emphasize that the steps of the example above (decomposition-synchronization) constitute a standard proof technique of our framework. Indeed, in [21] they have been used to prove a correspondence theorem between the Ambient reduction semantics and the graph semantics.

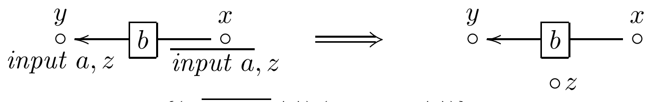
4.5 Remote actions and optimal routing

In this Section we underpin the features of our graph-based model which allows us to describe and reason about non-functional aspects of WAN programming. We specify the productions of an extended *in a* capability that permits an ambient b to enter a *remote* ambient a. Furthermore, we sketch how it is possible to compute and "select" the optimal route to an ambient a by augmenting name unification with the solution of Bellman-Ford shortest path equations. In [17] a similar approach has been adopted for a calculus based on KLAIM [18] using the Floyd-Warshall algorithm [23].

Remote actions Productions for *in a* prefix requires that the moving ambient has a sibling ambient called a. This "neighborhood" condition reflects the reduction rule of the ambient calculus. However, we can relax such requirement and consider the possibility of having "remote" *in a* in the sense that the moving ambient can enter an ambient a that is a sub-ambient of one of its sibling ambients.

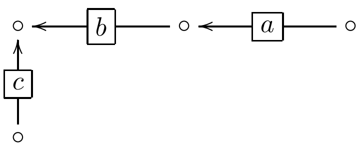
The idea is to add a production (*input3*) that may forward *input* signals to its inner ambients

(*input3*)

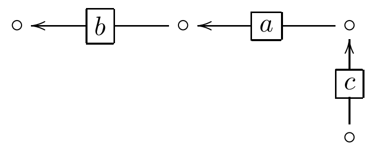


$$x, y \vdash b(x, y) \xrightarrow{\{(x, \overline{input\ a}, \langle z \rangle), (y, input\ a, \langle z \rangle)\}} x, y, z \vdash b(x, y)$$

Let us consider the ambient graph

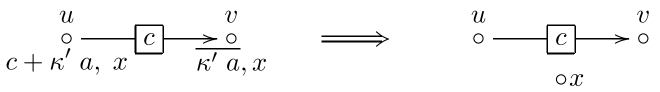


Production (*input*3) and the productions in the previous section allow the evolution whose graph representation is



Note that c enters a non-sibling ambient a. A similar production would allow to forward input signals also to outer ambients. Exploiting both productions, an ambient with an in-capability could enter an ambient labeled a anywhere in the network.

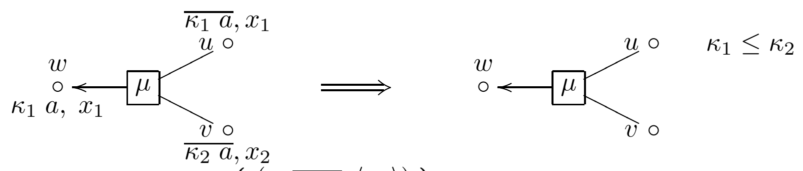
Optimal routing We consider *weighted link edges* and *router edges*; link edges represent the connections between sites. Each link edge is labeled with a cost c. We assume that c is not null. The production schema for a link edge $c(u,v)$ is



$$u, v \vdash c(u,v) \xrightarrow{\{(u, c+\kappa'\ a, x), (v, \overline{\kappa'\ a}, x)\}} u, v, x \vdash c(u,v)$$

When a link edge $c(u,v)$ finds on v the minimal cost κ' of a path to an ambient a, then it "backward" propagates to node u the cost obtained by summing c and κ'.

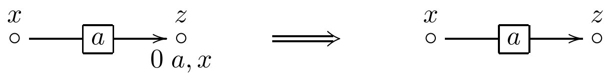
A router edge selects the minimal cost between two paths:



$$u, v, w \vdash \mu(u,v,w) \xrightarrow{\left\{\begin{array}{c}(u, \overline{\kappa_1\ a}, \langle x_1\rangle)\\ (v, \overline{\kappa_2\ a}, \langle x_2\rangle)\\ (w, \kappa_1\ a, \langle x_1\rangle)\end{array}\right\}} u, v, w \vdash \mu(u,v,w), \qquad \kappa_1 \leq \kappa_2$$

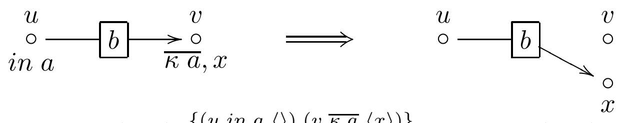
The edge $\mu(u,v,w)$ propagates to w cost κ_1 and node x_1, where κ_1 is the minimal cost between the costs κ_1 and κ_2 of u and v.

Ambient coordination productions are extended with two new productions. The first production allows an ambient to communicate its name to a router edge, at zero cost.



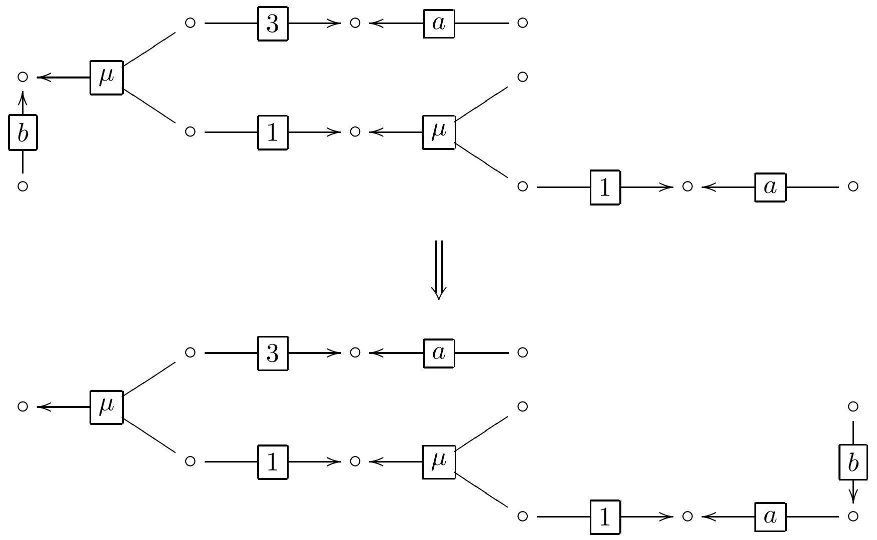
$$x, z \vdash a(x,z) \xrightarrow{\{(z, 0\ a, \langle x\rangle)\}} x, z \vdash a(x,z)$$

The production below states that, when an ambient b is asked to enter an ambient a from one of its internal processes and the router of b communicates the continuation x with the minimal cost κ, then b detaches from v and re-connects itself to x.



$$u, v \vdash b(u,v) \xrightarrow{\{(u, in\ a, \langle\rangle), (v, \overline{\kappa\ a}, \langle x\rangle)\}} u, v, x \vdash b(u,x)$$

It can be proved (similarly to what done in Section 4.4) that the following graph transition can be obtained.



Note that the upper-most a is not entered because the path leading to it costs 3, while the bottom-most a has a path cost 2 and has been selected.

5 System Development

In this Section we show how synchronized edge rewriting can be exploited in the earlier phases of software development. In particular, we will consider UML [40] specifications and their graph transformation semantics as given in [35]. We first outline the main ideas of the methodology introduced in [35].

The drive-through example A drive-through can be visited by an ordered set of clients. Each client has a running number which indicates his/her turn. A client may submit an order to the drive-through that later will be served. The service order is established by the running number assigned at visit time.

A UML *class diagram* describing the main relations among the component (i.e. the classes) of the system may be depicted as in Figure 1. Other features of systems are expressed in UML by means of *object diagrams* that may be thought of as diagrams describing the state of the system at a given moment. Figure 2 displays an object diagram of a possible evolution of the system described in Figure 1; a drive-through and three clients have been instantiated. Two of the clients visit the drive-through and one of them has issued an order. The operations listed in the class diagram may affect the relations among the objects in a given state of the system's evolution. This is captured by transformation rules related to class diagrams. These rules transforms object diagrams in object diagrams. In general, a set of graph transformations is associated to each specified

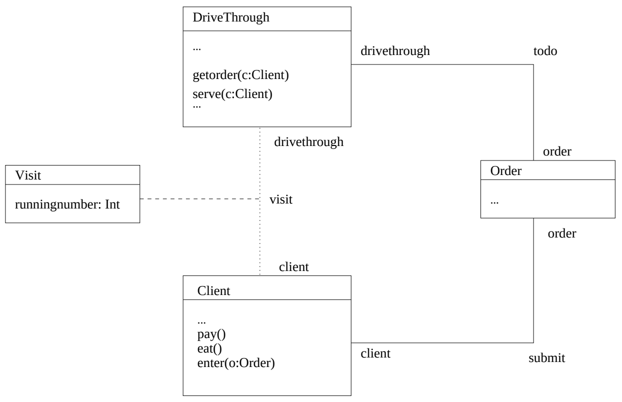


Fig. 1. UML class diagram

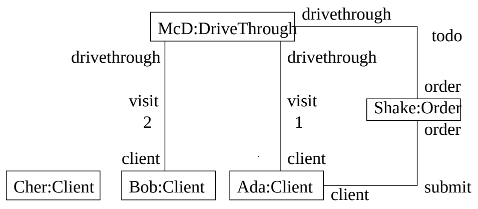


Fig. 2. UML object diagram

operation. Figure 3 illustrates the rule of the serve operation for drive-through objects. The serve rule expresses that the link between the instance of an order

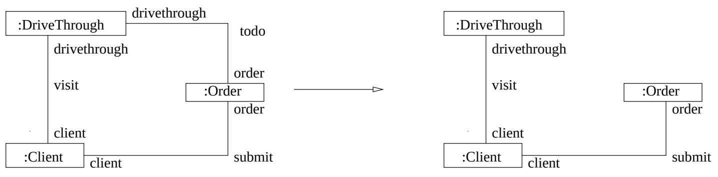


Fig. 3. Serve operation

and the instance of the drive-through that processes it is removed when the serve action is executed.

Dynamic behaviour of the system's components is described in terms of *state diagrams*. State diagrams are associated to classes and describe the state changes of their objects. They are finite state automata whose transitions are labeled with an event, a guard and an action. Labels are written as $e[g]/o'.e'$, where e is the event that triggers the transition, g is a logic formula specified in OCL [40] and represents a pre-condition to the firing of the transition. Finally, $o'.e'$ is the invocation of the method e' of object o'.

Figure 4 describes the state diagrams of classes DriveThrough and Client. The Client diagram details the activity of a client as a cyclic sequence of entering

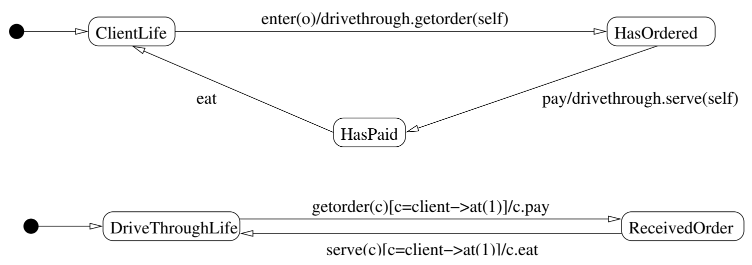

enter(o)/drivethrough.getorder(self)
ClientLife
HasOrdered
eat
pay/drivethrough.serve(self)
HasPaid
getorder(c)[c=client->at(1)]/c.pay
DriveThroughLife
ReceivedOrder
serve(c)[c=client->at(1)]/c.eat

Fig. 4. UML state diagram

an order/asking for the order to be executed, paying/waiting for being served and eating. When a drive-through must process an order, it checks that the order has been issued by the client on the top of the stack. In this case, the client is asked to pay for it and eventually the client is served and can start eating provided that payment has been performed.

Given a state diagram, it is possible to associate a graph transformation to each transition of the diagram. For this purpose, we assume that event stacks are associated to objects. Let us consider a transition $t = s \xrightarrow{e[g]/o'.e'} s'$ of a state diagram of class C. We may interpret t as the evolution of each object o in C whose first event in its event stack is e and the guard $[g]$ is evaluated to true; transition t also dispatches the event e' to the event stack of object o'. This interpretation may naturally be formalized with the graph transformation in Figure 5 while Figure 6 is an instance of the schema detailed above and describes

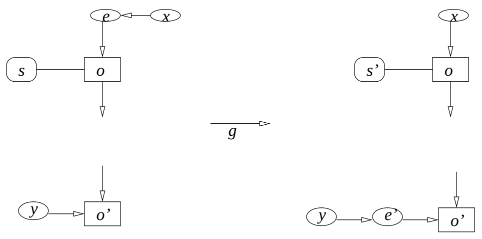


Fig. 5. Graph Transformation of a Transition

the rule corresponding to the serve transition of the drive-through state diagram.

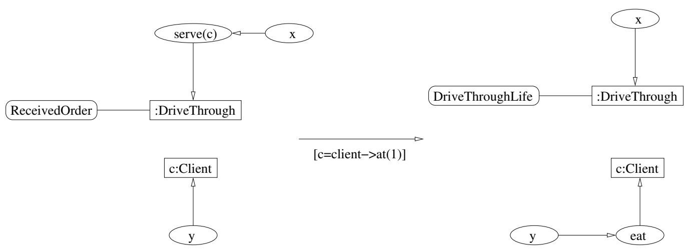


Fig. 6. Transition rule for serve

Roughly, the object transformations represent the global evolution of the system caused by the activity of its components, while transitions of state diagrams represent the local state changes. The graph transformation rules corresponding to those different facets of system evolution must be mixed together in order to obtain the so called *integrated rules*. In the case of the serve rules, we have Figure 7.

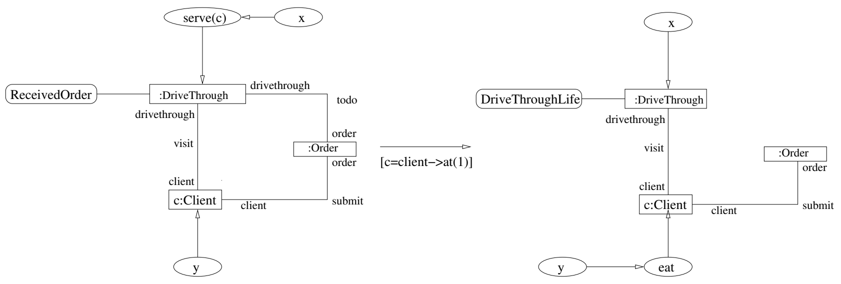


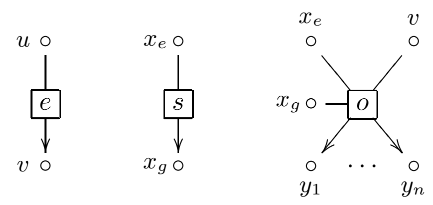
Fig. 7. Integrated rule for serve

Notice that the rule above do not specify some crucial aspects of the specification. For instance, the integrated rule of Figure 7 does not describe howthe eat event is pushed on the event stack of the client. In other words, the interactions between the client and the drive-through remain at the abstract level of method invocation, without being “network aware”.

5.1 Formal specification with edge replacement

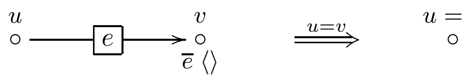
In this section we describe how it is possible to associate productions of our calculus to the graph transformation rules given previously. We aim at showing the use of edge synchronization to formalize the issues that in the above specification have not been considered.

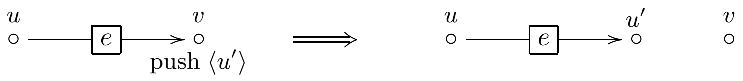
We consider three different forms of edges; events, controls and objects. They are graphically represented as



We assume that an edge label e does exist for each event e, and that a control edge exists for each state in a state diagram. Similarly, an object edge exists for each class in the UML specification. Edge e has two nodes such that a stack may be formed by merging node u of an edge labeled by e with a node v of another event edge. However, v nodes may also be fused with v of object edges. A control edge has two nodes. Node x_e is used to acquire the actual event from the object edge, while node x_g is used for checking guard satisfaction. These nodes are fused with the corresponding nodes of an object edge. An object edge has nodes for synchronizing with its control and event edges but also nodes $y_1, ..., y_n$ for connections with other objects according to the UML class diagram of the system.

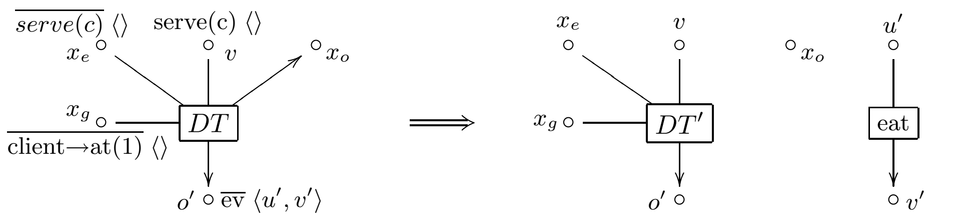
Event edges must be popped when they synchronize with objects, and they must be pushed on the existing stack when they are created. Thus event edges have two productions; the first synchronizes with objects sending to them the event name. After the transition, the edge disappears and reconnects the rest of the stack with the v node of the corresponding object by fusing u and v. The second reacts to a "push" message:



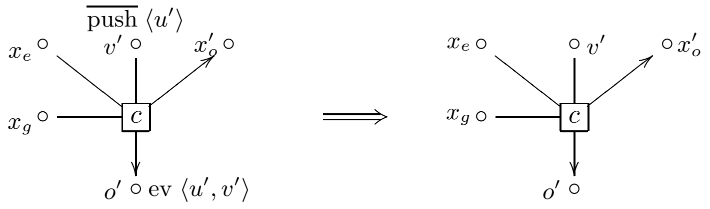


Note that the event that receives a push synchronization shifts back and fuses the v' node with the v node of the relative object edge. We remark that the previous productions are obtained by considering the intended semantics of event stacks in the UML specification. Moreover, productions for control and object edges may be derived from the UML class, object and state diagrams in a uniform way.

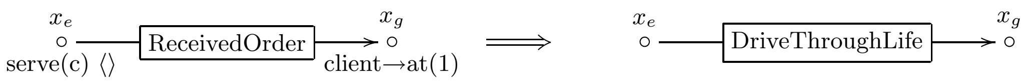
Let us consider the rules for serve in Figure 3 and 6. The following productions describe the evolution of each component of the system in terms of hypergraphs.



$\overline{serve(c)}\ \langle\rangle$
serve(c) $\langle\rangle$
x_e
v
x_o
x_g
DT
$\overline{\text{client}\rightarrow\text{at}(1)}\ \langle\rangle$
o' $\overline{\text{ev}}\ \langle u', v'\rangle$
x_e
v
x_o
u'
x_g
DT'
eat
o'
v'



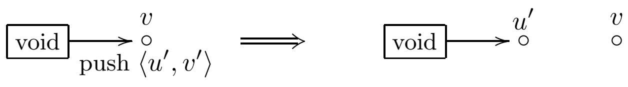
$\overline{\text{push}}\ \langle u'\rangle$
x_e
v'
x'_o
x_g
c
o' ev $\langle u', v'\rangle$
x_e
v'
x'_o
x_g
c
o'



The first production states that an object edge DT that receives an event 'serve(c)' on the node corresponding to the event stack, evaluates the guard 'client→at(1)' and forwards the signal together with the evaluated guard to its control edge. It also sends the new event 'eat' on the node o' connected to the client; this is obtained by passing to the client object the nodes of the 'eat' event. As stated before, guards are expressed as OCL formulas; however, we do not model how they can be mapped into graphs and how they can be evaluated using edge replacement. The second production is the complementary rule of the previous production: when the client object receives the 'eat' event, it pushes the event and its stack. The last production states that the control edge 'ReceivedOrder' changes its label to 'DriveThroughLife' when the 'DriveThrough' object signals the 'serve(c)' event and the verification of the corresponding guard 'client→at(1)'.

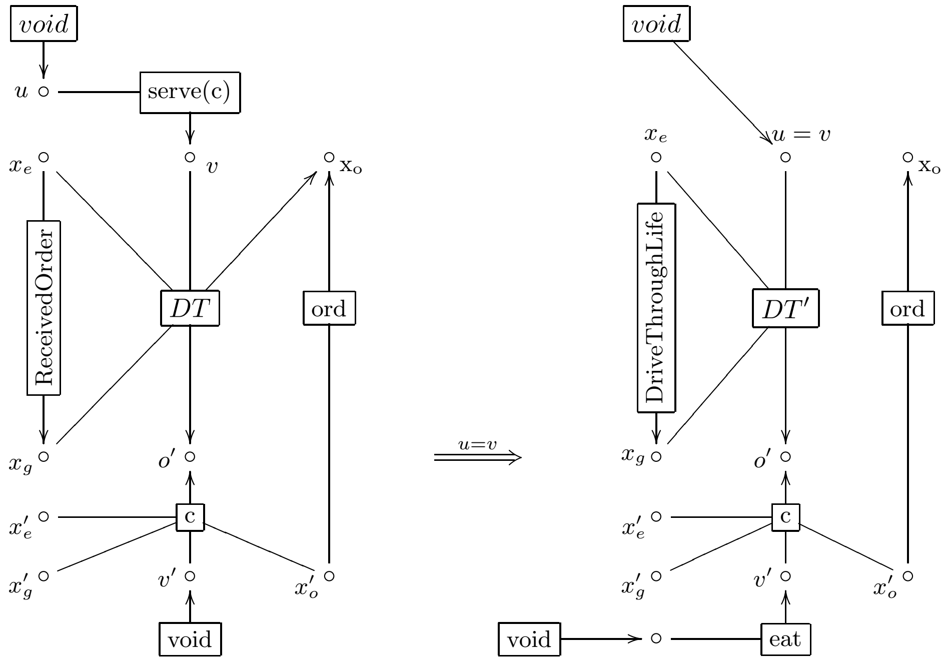
The productions introduced above guarantee that it is possible to obtain a transition which is equivalent to the integrated graph transformation in Figure 7.

In order to make the stack of events properly work, it is necessary to have an edge that manages the empty stack and allows an object edge to synchronize on push action only.



The edge behaves as an event edge that receives a push signal and, after the transition, it is connected to the node u' that is the last node of the stack of events.

The synchronization rules ensure that the following transition can be derived



Note also that the proof technique used to obtain it is as described in Section 4.4.

6 Conclusions

This paper has introduced a formal model for specifying and designing Wide Area Network applications. The novelty of our proposal is given as the combination of the following ingredients:

- the graphical notation is designed to deal with distribution;
- mobility is obtained via local synchronization constraints and their solution using unification;
- the declarative approach based on constraints can be extended to quantitative QoS requirements, e.g. to Bellman-Ford equations for optimal routing.

We showed the applicability of the approach by considering two illustrative case studies. Indeed, we gave an interactive distributed semantics for the Ambient calculus. Moreover, our framework permits extending the Ambient calculus with capabilities for remote interaction in a straightforward manner. Finally, we outlined a technique to refine UML specifications to include explicit synchronization among components which have to reconfigure their connections to accomplish a state change.

References

1. M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.
2. L. Andrade and J.L. Fiadeiro. Coordination for orchestration. In *COORDINATION 2002*. LNCS, 2002.
3. O. Angin, A. Campbell, M. Kounavis, and R. Liao. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking. *IEEE Personal Communications Magazine*, August 1998.
4. L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In *Proc. of the 7th Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 110–115, 1998.
5. S. Blake, D. Black, M. Carlson, E. Davies, Z. Wand, and W. Weiss. An architecture for differentiated services. Technical Report RFC 2475, 1998.
6. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
7. R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (rsvp) - version 1 functional specification.
8. R. Bruni, C. Laneve, and U. Montanari. Orchestrating Transactions in Join Calculus. In L. Brim, P. Jancar, and M. Kretinsky, editors, *Proceedings of CONCUR 2002, 13th International Conference on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, 2002.
9. M. Bugliesi, G. Castagna, and S. Crafa. Reasoning about Security in Mobile Ambients. In *Concur 2001*, number 2154 in Lecture Notes in Computer Science, pages 102–120. Springer, 2001.
10. L. Caires and L. Cardelli. A spatial logic for concurrency (part II). In *CONCUR'02*, volume 2421 of *Lecture Notes in Computer Science*. Springer, 2002.
11. L. Cardelli and A. Gordon. Anytime, anywhere — modal logics for mobile ambients. In *Proceedings of POPL '00*, pages 365–377. ACM, 2000.
12. L. Cardelli and A.D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. An extended abstract appeared in *Proceedings of FoSSaCS '98*, number 1378 of LNCS, pages 140-155, Springer, 1998.
13. G. Castagna, G. Ghelli, and F. Zappa Nardelli. Typing mobility in the seal calculus. In *Concur 2001*, number 2154 in Lecture Notes in Computer Science, pages 82–101. Springer, 2001.
14. G. Castagna and J. Vitek. Seal: A Framework for Secure Mobile Computations. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, number 1686 in Lecture Notes in Computer Science, pages 47–77. Springer, 1999.
15. G. Cinciarone, M. Corson, and J. Macker. Internet-based mobile ad hoc networking. *Internet Computing*, 3(4), 1999.
16. E.M. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.
17. R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A formal basis for reasoning on programmable qos, verification–theory and practice. Proceedings of an International Symposium in Honor of Zohar Manna's 64th Birthday, Springer LNCS, submitted for publication.
18. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998. Special Issue: Mobility and Network Aware Computing.

19. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.
20. G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioural diagrams in UML. In *UML 2000*, number 1939 in Lecture Notes in Computer Science, pages 323–337. Springer, 2000.
21. G. Ferrari, U. Montanari, and E. Tuosto. A lts semantics of ambients via graph synchronization with mobility. In *7th Italian Conference on Theoretical Computer Science – ICTCS'01*, volume 2202 of *Lecture Notes in Computer Science*. Springer, 2001.
22. G. Ferrari, C. Montangero, L. Semini, and S. Semprini. Mark: A reasoning kit for mobility. *Automated Software Engineering*, 9(2):137–150, 2002.
23. Robert, W. Floyd. Algorithm97 (shortestpath). *Communication of the ACM*, 5(6):345, 1962.
24. R. Focardi and R. Gorrieri. The Compositional Security Checker: A tool for the verification of information flow security properties. *IEEE Transaction on Software Engineering*, 23(9):550–571, 1997.
25. I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the International Workshop on Quality of Service*, 1999.
26. C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. of 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.
27. A. Fuggetta, G. Picco, and G. Vigna. Understanging Code Mobility. *IEEE Transactions on Software Engineering*, 24(5), 1998.
28. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
29. M. Gogolla. Graph transformations on the UML Metamodel. In *ICALP Workshop on Graph Transformations and Visual Modeling Techniques*, pages 359–371. Carleton Scientific, 2000.
30. M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. In Uwe Nestmann and Benjamin C. Pierce, editors, *Proc. of HLCL '98: High-Level Concurrent Languages*, volume 16.3 of *ENTCS*, pages 3–17. Elsevier, 1998. To appear in Information and Computation.
31. M. Hennessy and J. Riely. Distributed Processes and Location Failures. *Theoretical Computer Science*, 266, 2001.
32. D. Hirsch, P. Inverardi, and U. Montanari. Reconfiguration of Software Architecture Styles with Name Mobility. In Antonio Porto and Gruia-Catalin Roman, editors, *Coordination 2000*, volume 1906 of *LNCS*, pages 148–163. Springer, 2000.
33. D. Hirsch and U. Montanari. Synchronized hyperedge replacement with name mobility. In *CONCUR 01*, number 2154 in Lecture Notes in Computer Science, pages 121–135. Springer, 2001.
34. IBM Software Group. Web services conceptual architecture. In *IBM White Papers*, 2000.
35. S. Kuske, M. Gogolla, R. Kollmann, and H.J. Kreowski. An Integrated Semantics for UML Class, Object, and State Diagrams based on Graph Transformation. In Michael Butler and Kaisa Sere, editors, *3rd Int. Conf. Integrated Formal Methods (IFM'02)*. Springer, Berlin, LNCS, 2002.

36. B. Li. *Agilos: A Middleware Control Architecture for Application-Aware Quality of Service Adaptations*. PhD thesis, University of Illinois, 2000.
37. G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letter*, 56(3):131–133, 1995.
38. P.J. McCann and G. Catalin-Roman. Compositional programming abstraction for mobile computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, 1998.
39. U. Montanari and F. Rossi. Graph Rewriting and Constraint Solving for Modelling Distributed Systems with Synchronization. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First International Conference COORDINATION '96, Cesena, Italy*, volume 1061 of *LNCS*. Springer, April 1996.
40. OMG. Unified modelling language specification. Available at http://www.omg.org, 2001.
41. Oracle. Oracle 8*i* lite web page. In *http://www.oracle.com/*, 1999.
42. A.S. Park and P. Reichl. Personal Disconnected Operations with Mobile Agents. In *Proc. of 3rd Workshop on Personal Wireless Communications, PWC'98*, 1998.
43. G. Picco, A.L. Murphy, and G.C. Roman. LIME: Linda meets mobility. In *International Conference on Software Engineering*, pages 368–377, 1999.
44. M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice Hall, 1996.
45. J.L. Sobrinho. Algebra and algotithms for qos path computation and hop-by-hop routing in the internet. *IEEE Transactions on Networking*, 10(4):541–550, August 2002.
46. U. Ster and M. Mitchell, J.C.and Mitchell. Automated analysis of cryptographic protocols using murϕ. In *10th IEEE Computer Security Foundations Workshop*, pages 141–151. IEEE Press, 1997.
47. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., November 1994.
48. P.T. Wojciechowski and P. Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, 2000.
49. X. Xiao and L. M. Ni. Internet qos: A big picture. *IEEE Network*, 13(2):8–18, Mar 1999.
50. M. Yokoo and K. Hirayama. Algorithms for Distributed Constraint Satisfaction: A Review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.