A framework for specifying and verifying the behaviour of open systems

Andrea Bracciali *, Antonio Brogi, Franco Turini

Computer Science Department, University of Pisa, Italy

Abstract

Coding no longer represents the main issue in developing software applications. It is the design and verification of complex software systems that require to be addressed at the architectural level, following methodologies which permit us to clearly identify and design the components of a system, to understand precisely their interactions, and to formally verify the properties of the systems. Moreover, this process is made even more complicated by the advent of the "network-centric" model of computation, where *open systems* dynamically interact with each other in a highly volatile environment. Many of the techniques traditionally used for closed systems are inadequate in this context.

We illustrate how the problem of modeling and verifying behavioural properties of open system is addressed by different research fields and how their results may contribute to a common solution. Building on this, we propose a methodology for modeling and verifying behavioural aspects of open systems. We introduce the *IP-calculus,* derived from the π-calculus process algebra so as to describe behavioural features of open systems. We define a notion of partial correctness, *acceptability*, in order to deal with the intrinsic indeterminacy of open systems, and we provide an algorithmic procedure for its effective verification.

Key words: Open systems, process algebras, verification.

* Corresponding author. Address: Computer Science Dept., University of Pisa, Via F.Buonarroti 2, 56100 Pisa (Italy). `e-mail`: braccia@di.unipi.it

Email addresses: `braccia@di.unipi.it` (Andrea Bracciali), `brogi@di.unipi.it` (Antonio Brogi), `turini@di.unipi.it` (Franco Turini).

1 Introduction

The increasing difficulty to satisfactorily deal with the complexity of software systems has motivated a consistent effort to establish system development as a *formal architectural process*. Coding is no longer the main activity in building applications, but rather architectural design, that is, the definition of the components which constitute an application and their relationships.

Building systems by integrating components offers a number of well recognised advantages, such as reusing existing code, having a modular architecture and studying recurrent architectural patterns which may be exploited to address similar problems and classify systems. On the other hand, formal architectural software development requires a new set of models, linguistic abstractions, and verification tools, which were neither applicable, nor needed, by code development.

According to principles of modularity and encapsulation, components are represented by abstractions that hide, as much as possible, their internal structure, as long as this is not interesting to the environment where components are going to be assembled into a system. Unfortunately, traditional interfaces have been shown not to be expressive enough to satisfactorily support architectural design. Indeed, beyond traditional functional properties, such as object signatures, also behavioural properties and other *non-functional* properties, such as quality of service and security properties, must be described by abstract interfaces in order to properly support architectural design and verification.

This paper focuses on *behavioural properties*, i.e. the description and the analysis of how components properly interact with each other, possibly concurrently and in a distributed environment. Moreover, the recent advent of a network-centric model for software applications poses further problems to the development process. *Autonomous* applications may no longer be *centrally controlled* and may be composed of *heterogeneous* components, which can be spread and distributed over networks, belong to different domains, depend on components beyond their control, and even *migrate* from one domain to another. The environment itself, hence, may only be partially accessible, as it is populated by other independent components which may dynamically provide and require services, or even try to maliciously exploit the work of the others. Such systems operate with *incomplete information about the environment*, which may not always be available. Systems with the above mentioned features, are generally called in the research community *open systems*. Not surprisingly, as it will be discussed in Section 2, the design and verification of open systems in general, and their behavioural aspects in particular, appear more difficult than usual closed component-based systems, and require the

definition of new models, properties and verification techniques.

Traditionally, *process algebras* have often been proposed as a useful paradigm for the specification and analysis of the above mentioned behavioural properties. In particular, being neither merely mathematical abstractions nor real programming languages, they not only facilitate rigorous system analysis by focusing on the aspects of interest (e.g. security, distribution, mobility), but can also offer the basis for prototypical implementation and for verification tools. For example, there are now many running implementations of languages based on calculi that were originally proposed as benchmarks to experiment and study a few basic primitives and behavioural issues (e.g. name passing in π-calculus [1], process passing in HOπ-calculus [2], behaviour refinement [3], cryptographic messages in spi-calculus [4], distributed dataspaces in KLAIM [5], mobile environments in ambient-calculus [6]).

Nonetheless, we discuss in this paper how the use of process algebras as a model for component interaction within open systems requires the adoption of suitable linguistic constructs (essentially, explicit late binding mechanisms) and the definition of new correctness properties and their verification methodology. We therefore present the IP-calculus, by adapting the π-calculus process algebra in order to suitably model the autonomous behaviour of, possibly heterogeneous, components within open systems. The calculus is intended, as usual, to enrich component interfaces with an abstract description of their interactive behaviour. The choice of a name-based calculus instead of more traditional finite state machines, and its adaptation to open system concerns represent the major novelty of our work. Arguing that dynamics and incomplete accessibility of an open system prevent a complete analysis of global properties, we focus on partial abstractions, namely, component behaviour is partitioned into finite-in-time fragments. We have also adopted explicit runtime composition mechanisms. Accordingly, the properties one may wish to check in an open system must be reconsidered. We have hence introduced a notion of *partial correctness*, called *acceptability*, which, informally speaking, is read as "the system, up to now, does not contain unrecoverable errors, and all the components *might* successfully accomplish their tasks". This property is based on the finite verification of the behaviour that components have declared to perform for the *next transaction* of the system. Note that the indeterminacy of the definition is due to the incomplete information available in an open system, since, for instance, new components dynamically joining the system may either provide the resources needed by other components, or introduce unrecoverable deadlocks. Acceptability, if preserved along the lifetime of an open system, for instance by preventing the access to harmful components, guarantees its proper functioning. On the other hand, as expected, it is not possible to make, at a given instant, statements about the overall life of an open system. Finally, an algorithmic procedure based on finite model checking techniques to effectively verify acceptability is illustrated. Correctness of

the procedure relies on the adopted hypothesis of finiteness for the behaviour model.

SYNOPSIS. Next Section 2 motivates the needs for modeling and verifying behavioural aspects in open systems, and discusses the contributions and limitations of current proposals in the field. The IP-calculus is introduced in Section 3, while acceptability and its algorithmic verification in Section 4. Section 5 draws some concluding remarks.

2 Motivations and state of the art

The effort of defining software development at an architectural level is undertaken by different research fields, which, even with different interpretations, are based on the concepts of component and composition. From different perspectives, all face the problem of how modeling and reasoning about the interactive behaviour of components and their properties. Moreover, the network-centric shift in the development of applications, forces them to take into consideration the typical features of open systems. In particular, for the scope of this paper, the three most relevant research fields are:

- SOFTWARE ARCHITECTURE (SA). The main aim of this field has been to define system development as an architectural process, within a well-understood theory on *"The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time"*, [7], so as to allow for design by patterns, verification of system structures and reuse of previously developed (sub-) systems.
- COORDINATION (CO). Defined as *"the process of building programs by gluing together active pieces"*, [8], it was born to cope with the complexity of concurrent and distributed programming. The key starting point for the development of a coordination theory is that *interaction concerns are orthogonal to computation concerns*, and as such they must be described separately in system models, [8–11]. This is still in contrast to what happens to most of the current technology infrastructures [12], where interaction is often "hard-wired" into the code of applications.
- COMPONENT-BASED SOFTWARE ENGINEERING (CBSE). The main motivation of this research field is *reuse*, according to the principle "write once, run forever". By exploiting existing components, which *"are binary units of independent production, acquisition and deployment that interact to form a functioning system."* [13], the development process can be improved in terms of time and cost. Even if in this context the system specification is generally fully accessible (*"architecture first"*), still interaction of components independently developed is an issue.

In the following we further illustrate the problems relative to architectural design, behavioural concerns and openness of the systems, which the above approaches share.

2.1 Architectural design

Complexity of software systems calls for formal and rigorous design methodologies, and in particular for shifting software design from the usual process of *developing requirements into code,* towards *assembling components into systems*. As it was already widely recognised a decade ago *"As the size of software increases, the algorithms and the data structures of the computation no longer constitute the major design problems. When systems are constructed from many components, the organisation of the overall system [...] presents a new set of design problems"*, [14], and hence standard "code-oriented" methodologies appeared not suitable for sustaining the technology evolution. The benefit of a convergence of Software Engineering and (architectural) Programming Languages has extensively been recognised, [15], motivating the birth of SA as a sub-field of software engineering. The main aim of this research field has been to define system development as an architectural process, within a well-understood theory on the structure of the systems, the components and their interrelationships, so as to allow for design by patterns, verification of system structures and reuse of previously developed (sub-) systems. Moreover, a clear separation of the system structure design from its actual implementation is desirable for systems that are intrinsically *heterogeneous*, made of independent components which possibly do not share a homogeneous programming paradigm, and can be assembled together by adding, upgrading or removing components to a, possibly running, system. Heterogeneity also regards the problem of legacy software, i.e. existing applications that one wants to integrate in the new architectures, but that often have not been designed for this purpose. Clearly, these above mentioned issues are of interest also for CBSE. Finally, both SA and CBSE are concerned with *reuse*, read as modularity, substitutability or refinement, with *maintenance* and with *verification*. Indeed, both architectural descriptions and component architectures are suitable to be equipped with automatic verification tools (typically based on model checking techniques).

After the work about the description and the analysis of recurrent design patterns in object-oriented code development, [16], the idea of exploiting recurrent design structures at an architectural level has also been adopted in SA, [7,17–19], with the objective of identifying basic components and their interconnections and studying the patterns that more often recur in building an architecture out of components and interconnections. Clearly identifying the most suitable patterns in the early steps of design improves the quality of

the design process, enhancing system analysis by techniques based on a well understood set of architectural structures. Some examples of recurrent architectural patterns are, according to [7], client-server models, pipelines, filters, layered structures and blackboard systems. This trend necessarily requires a precise identification of the constitutive elements of a system, i.e. its *components*, from the one hand, and of their *inter-relations*, from the other, as soon as one wishes to integrate autonomous components designed, possibly by independent vendors, according to different paradigms. Notice that standard interfaces, like for instance object-oriented signatures, are not enough expressive for these purposes, since they do not account for the dynamics and dependencies of interactive behaviour.

After about ten years, significant progress has been made. Mainly,

- the needs for an engineering approach has been fully recognised, both as a research topic and in the industrial practice,
- a set of formal notation and tools for describing and analysing architectural designs, like *Architecture Description Languages* (ADL) within SA, or *Interface Description Languages* (IDL) within CBSE, have been defined.

However, at present, there is not yet a complete agreement on which are the constitutive elements of a system that must, or must not, be supported by an ADL, like, e.g., domain specific architectural features, supported architectural styles, event-based or distributed message passing communication protocols, support for the system analysis and simulation, architectural refinement, support for different kinds of connectors, etc. Some ADLs promote an explicit distinction between components and connectors, e.g. [18], arguing that such a distinction is more natural for architecture design, facilitating the practical usage of description formalisms, [20], while other proposals, like the Leda ADL, [21,22], support the view of connectors as computational entities, hence components, in name of the minimality of the set of abstractions needed by an ADL to describe the system. Being a detailed comparison of existing ADL out of scope for this paper, we refer the interested reader to general surveys, like [23,24].

2.2 Behavioural issues

Autonomous components require to coordinate their tasks and the interaction with the environment. The wide connectivity made available by the development of network infrastructures and the availability of large integration capabilities, both at the hardware and the software level, permits applications to refer to the global (inter-)net as a seamless offer of resources, so that, often their activity can not be centrally controlled, either because global control is

impossible or impractical, or because the entities are created or controlled by different owners. The need for modeling *behavioural* properties is hence fully recognised [25–27]. However, even if it has been clearly observed within the CO field that *interaction concerns are orthogonal to computation concerns*, and as such they must be described separately in system models, [8–11], a major limitation of both many ADLs and available component-oriented platforms is that they do not provide suitable means for describing and reasoning about the interactive behaviour of concurrent components. Moreover, for most of the current technology infrastructures [12] and IDLs, interaction is often "hard-wired" into the code of applications. For instance, component based architectures like CORBA [28], or JavaBeans [29], rely on low-level mechanisms for managing concurrency, like *lock* capabilities over shared resources and *synchronised* methods, which must be coded within the "functional" code of a components. Several proposals have been put forward in order to enhance IDLs with a description of component concurrent behaviour, in order to prove correctness properties, such as absence of deadlocks, as well as compatibility relations, such as "the components can properly interact with one another" or "this component can be substituted with that one" (see, for instance, [27]). Behavioural descriptions are typically provided by means of concurrency models, like, Finite State Machines, process algebras (CSP [30], CCS [31]) and name based calculi (π-calculus, [1]). A limitation of some of these approaches is the computational cost of verifying such properties when they are interpreted as global properties of the whole system. Following this general trend, also some ADL features behavioral descriptions, as, for example, in the case of *Wright,* [32,20], an ADL which supports behavioural descriptions for components and connectors by means of a finite state process algebra (based on CSP, [30]). Architectures are verified against compatibility of ports, roles and glue code. A relation of refinement over behaviours is defined. If a behaviour is replaced by one of its refinements, correctness of the composition is preserved. The choice of CSP as an underlying model does not support the dynamic reconfiguration of the communication network. Other approaches adopt different models, like the one in [33], which exploits the CHAM (Chemical Abstract Machine) interaction model in order to statically check the compatibility of the behaviour expected by a component with the one exhibited by its environment.

However, these approaches, oriented to the description and verification of a fully controlled application, can hardly be applied to the case of open systems.

2.3 Open architectures

Technology development is shifting architectural design towards a network-centric, *open* model of computation, where applications can be spread and distributed over networks, may belong to different domains or depend on

components beyond their control. The environment itself, hence, may be only partially accessible, in its being populated by other independent components which may dynamically provide and require services, or even try to maliciously exploit the work of the others.

Even if architectural design has been motivated within a context of *closed* systems, the research done in SA and CBSE has already addressed significant issues for open systems, like, for instance, component interfaces description and, sometimes limited, techniques for assembling, reconfiguring and verifying the behavioural composition of components. At any rate, such models are oriented to define system development as an engineering process for assembling fully specified applications, which are totally contained within a controlled domain, limiting the autonomy of components and practically not concerned with security or mobility issues. Moreover, systems are verified against classical notions of correctness.

In this respect, the ability to deal with run-time behaviour reconfiguration, traditionally addressed by CO, appears to be a valuable direction for improving the existing architectural research, as well as the support infrastructures for CBSE, which, as mentioned, do not offer enough abstract control mechanisms. For instance, it could be interesting to export an event-based run-time reconfiguration mechanism (possibly inspired by a coordination language like *Manifold* [34]), to an ADL like Rapide [35], which, even if based on an event model and provided with temporal logic based analysis, cannot naturally express dynamically evolving system specifications.

Another profitable approach comes from the interpretation of components as *services*: the design process, instead of assembling a system by composing components, is oriented to *orchestrating* services, which are available on the network, by *dynamically linking* them to applications, which use them over an open network, not subject to a common domain, and only partially accessible (see, for instance, [36]). A first proposal in this sense, is constituted by Web Services, [37], an architecture based on the concept of available service, where services are independently produced and published, together with a suitable description of their functioning (based on the XML format). Applications interact with services by locating them, possibly via a UDDI (Universal Description, Discovery, and Integration specification) repository, and by dynamically binding and using them, according to the paradigm *publish, find, bind*. The Web Service architecture is based on a standardised layered architecture, which is in charge of supporting the various phases of the process.

Summing up, the open system paradigm imposes an upgrade of the goals and techniques that must be developed for a mature architectural design, [38]. More precisely,

- extra information must be provided by the component abstractions, like the assumptions on which they are based, and conditions under which they can be, possibly dynamically, integrated into a system. This is sometimes called *metainformation* or *self-typing* of a component,
- different notions of correctness, which are incremental, progressive and approximate, are needed. Verification of a system should be oriented at guaranteeing that "the system is *sufficiently* good for the tasks *at hand*,... and the analysis [is done] at a reasonable cost",
- the incomplete accessibility of the system must be taken into consideration by suitable dynamic binding (also known as *late binding*) mechanisms as well as failure recovery strategies,
- security and mobility issues are central to system design.

This view, also shared by others, like for instance [36] in the CO context, suggests to integrate the illustrated achievements of the three mentioned research fields in order to devise a model where coordination is clearly separated from computation, dynamic integration is properly supported by coordination interfaces and late binding techniques, autonomy and heterogeneity of services can be dealt with, possibly by means of appropriate metadata abstractions, application design is engineered at an architectural level, and functional and non functional properties, can be properly described and verified by means of dynamically (and efficiently) operated verification techniques.

3 Interaction Pattern calculus (IP-calculus)

This section introduces the *IP-calculus* that we propose to design and verify behavioural properties of open system architectures. First the assumptions on which our approach is based are explained, and then the calculus is defined.

3.1 Abstracting component behaviour

The IP-calculus presented in this section, and firstly introduced in [39], is designed to support behaviour concerns within component interfaces. According to the approach explained in Section 1, the behaviour of a component is projected into a set of temporally finite fragments represented as *interaction patterns*, characterising what can be observed from the environment which hosts the component: *references* (i.e. communication channels) offered to the environment and the *behaviour*, in terms of an expression of communication actions, that the component is ready to execute. Composition consists in appropriately connecting references, so that the behaviour that components exhibit through connected references properly matches. Such a definition is

sufficiently abstract to be applied, for instance, to systems that are partially specified in an early (statical) design phase, as well as to dynamically evolving applications based on mobile code. This kind of "local in time" analysis is motivated by two principal intents:

(1) Dealing with the partial specification of open systems, which makes the verification of global properties, i.e. properties regarding the whole life of a system, often unfeasible, and
(2) reducing computational complexity, which reverts to the exploration of the *finite* structure of the computation exhibited by a finite set of interaction patterns (and that can be approached with well-settled finite-state model checking techniques).

References that an interaction pattern offers to the environment are intended in a broad sense, e.g. as communication channels, method calls, etc., as naturally interpreted by name-based calculi (access as knowledge of the name of the resource), and in particular by π-calculus in a dynamic context. The usefulness of (dialects of) π-calculus to naturally express open systems (e.g. network reconfiguration, ensuring deadlocks avoidance, proper sharing of resources (names), substitutability of components), has been illustrated for component models like COM [40] and CORBA [41], and ADLs like Darwin [42] and Leda [22]. Semantics of communication requires that both parties of a communication share the same reference and agree (e.g. with respect to a type system) on the type of exchanged data.

A component participates in a *session,* i.e. a collection of interconnected interaction patterns which interact with each other, by proposing an interaction pattern and a given way of connecting the references of the pattern to those in the sessions. Sessions are *open* if they offer references to incoming patterns and *closed* otherwise, and consist of the finite amount of information represented by their interaction patterns. It is worth pointing out that:

- Independent patterns of the same component do not share a state since they are not meant to model global properties. Indeed, intuitively speaking, verification of interaction patterns corresponds to "locally" checking the compatibility of an interaction pattern with the "finite" session it joins. (A *subject reduction* theorem guarantees the invariance of this compatibility through possible evolutions, while access to potentially harmful interaction patterns will be denied).
- The IP-calculus is not provided with linguistic constructs for pattern dynamics, like mobility primitives to join a session, or to choose/negotiate one of the available interaction pattern, since different application architectures may require different constructs, which are however not essential to model and verify interaction pattern composition.
- The problem of determining a suitable connection among heterogeneous,

$$E ::= 0 \mid \alpha.E \mid \tau.E \mid E|||E \mid E + E \qquad C ::= Ch \mid X$$
$$\alpha ::= in(C, D) \mid out(C, D) \qquad D ::= B \mid C \mid X \mid (D, D)$$

Fig. 1. Behavioural expression syntax

possibly separately developed components, is a hard problem that may substantially vary, e.g., from engineer's decisions to automatic negotiation, depending on the specific architecture. In general, our proposal does not aim at studying how such connections can be determined, but rather at validating them, once they have been given.

3.2 Interaction Patterns

According to the hypotheses illustrated in the previous section, we have adapted π-calculus into a suitable process algebra, called *IP-calculus.* Interaction patterns are *processes* (or *agents*) of this calculus.

3.2.1 Syntax

Interaction patterns interact with each other by means of synchronous communications through shared channels. A channel is shared by two or more interaction patterns knowing the channel name. The communications that an interaction pattern can perform are represented by its *behavioural expression* E, whose syntax is presented in Figure 1. The behavioural expression 0 corresponds to (successful) termination, while $\alpha.E$ ($\tau.E$) represents a behaviour executing the communication action α (the *silent* action τ) and then the behaviour E (prefix operator). The silent action denotes an internal computational step that a component can perform independently of its environment, i.e. without synchronising with a communication action of any other component, and it is not distinguishable by the environment. The *parallel* composition of two behavioural expressions ($E_1 ||| E_2$) can execute, in any order, the actions that E_1 or E_2 is ready to execute, and the *nondeterministic choice* ($E_1 + E_2$) can either behave as $E1$ or as E_2, exclusively.

Communication actions α send or receive data through a channel, identified by its name: $in(C, D)$ is an input action receiving data D through channel (of name) C, while $out(C, D)$ is an output action sending data D through channel C. A channel C may explicitly consist of a channel name (Ch), or of a variable (X). Finally, a datum D may be a generic "basic" datum B (depending on the specific architecture), a channel (which can hence be communicated and then used, modifying the network topology), a variable, or a pair of data (i.e. a structured datum). In this paper, we do not take into consideration application data and type systems, but it is worth remarking that our model

allows them to be lifted at the behavioural interface level, preserving features of interest of the underlying component model. This allows, for instance, for typed communications, or for lifting cryptography as appropriate matching of shared cryptographic keys, as done in [43], where the framework has been applied to security.

As usual, variables in a behavioural expression E are bound by input actions. Scoping rules and α-renaming for variables work as usual as for the names of π-calculus, [1]. For instance in $in(c, X).out(X, d).0|||out(X, d).0$, the input action binds X in the first output action, but it does not bind X in the second one, which is free. With $bv(E)$ and $fv(E)$ we indicate the *bound* and *free* variables in E (if E is a communication action, it is treated as a label in π-calculus, i.e. X is bound in $in(Z, X)$).

The presence of the silent action is motivated to express the autonomy of each interaction pattern. The expression $\alpha + \beta$ can synchronise either with an action $\bar{\alpha}$ or an action $\bar{\beta}$ offered by another component in the environment, in a so-called *global* choice. On the other hand, the expression $\tau.\alpha + \tau.\beta$, via an internal (autonomous) step can evolve either in α or β, hence *autonomously* deciding, by a *local* choice, which action to synchronise on.

Interaction patterns support heterogeneity of components by not making any assumption on implementation. Moreover, all the names of a behavioural expression are assumed (unless otherwise specified) to be *local* (restricted, according to π-calculus jargon), since heterogeneous components, in general, do not share a common set of names. [1] Interacting components can anyway exchange names and consequently modify the communication network either by communication or by composition operations. A clear definition of this mechanisms facilitates the formal analysis of component-based open architectures.

An interaction pattern consists of an explicit declaration of the channel variables it offers to the environment, the *open variables*, and a behavioural expression.

Definition 1 (Interaction patterns) *An interaction pattern is a process of IP-calculus of the form:*

$$(X)[E],$$

where X, the open variables *of the pattern, is a set of variables. Open variables bind their free occurrences in E. The interaction pattern is required to be* closed*, i.e. $fv(E) \subseteq X$.*

[1] Note that the absence of restriction and recursion operators inhibits the creation of an infinite number of new names, and hence the Turing completeness. According to the hypotheses made, and to the approach that considers interaction orthogonal to computation, and hence possibly not Turing complete, the incompleteness of IP-calculus does not limit its expressiveness here.

Variables ranging over channel names are used to explicitly acquire references either by means of communications, or, differently from standard π-calculus, by means of a dynamic composition operation (late-binding), acquiring references from the environment. Indeed, within E a channel may occur as

- a *local* channel, i.e. a channel name which is defined inside the scope of the expression,
- a channel that has to be *received* by a communication, i.e. a channel variable bound by an input action,
- a channel that has to be *connected* to a channel of another component in the environment, i.e. a channel variable bound by an open variables.

Example 2 *Let us consider a WEB-server. It repeatedly waits for a request over a known channel, namely its IP address, and then it either provides the requested page, or it replies with an error message. This pattern of interaction can be represented by the following interaction pattern:*

`(MyAdd) [  in(MyAdd, page(P)).`

`(`τ`.out(MyAdd, a_page).0 + `τ`.out(MyAdd, error).0) ]`

The open variable `MyAdd` *represents the channel offered by the server to its environment, to which a client component can be connected. After that* `MyAdd` *has been connected to a client channel, the server can receive a request for a page, by offering the "service"* `page(P)` *(where* `page(_)` *represents the invocation to the offered service, and* `P` *is the variable to store the received url relative to the desired page). Then, the server either locally decides to send back the requested page* `a_page` *through the same channel, or to issue an error message. Note that in the chosen representation of the behaviour, the actual page and the kind of error sent are not relevant to the interaction description and treated as constants.*

3.2.2 Semantics

Semantics of IP-calculus is given as a Labeled Transition System (LTS), according to the seminal idea of Structured Operational Semantics [44], whose *states* represent processes and whose rules model how a state can evolve to another one. Transitions are *labeled* with information about the (observable) behaviour the processes exhibit, i.e. communication or silent actions. The semantics of IP-calculus is given by a pair of LTS. The first one, $\rightarrow$, defined in Figure 2, models the intensional behaviour of the interaction patterns in isolation: executed actions are observable as labels. The second one, $\mapsto$ of Figure 3, lifts $\rightarrow$ to the semantics of interaction patterns within a session.

The LTS $\rightarrow$ is defined up to structural congruence $\equiv$ (`+` and `|||` are associative monoidal operators with 0 as neutral element and $\alpha-$renamed terms are equivalent, [1]) as standard for π-calculus. A behavioural expression consisting

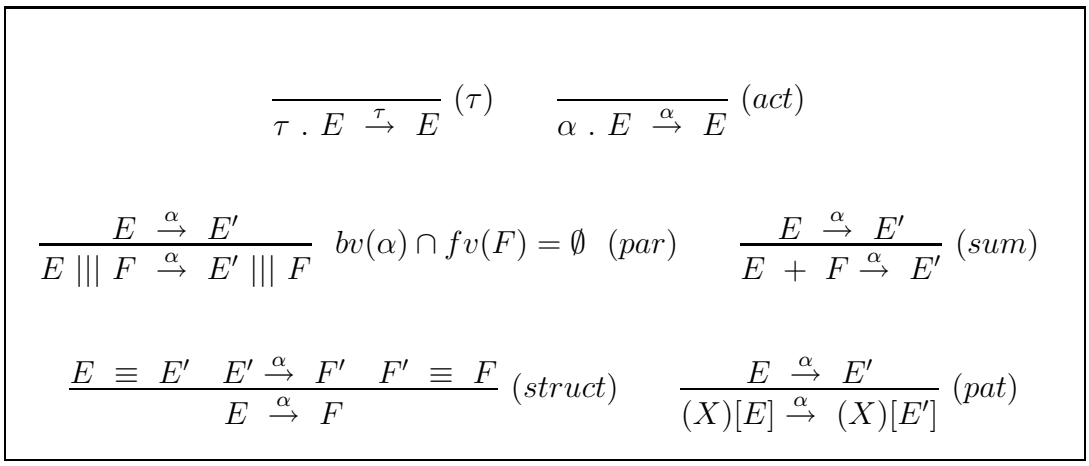

Fig. 2. Interaction pattern semantics.

of a prefix can execute the prefix action (rules (τ) and (act)), the parallel composition of two expressions can evolve according to both the expressions (rule (par)), with the standard condition to avoid free variable capture [2], while the sum of two expressions can non-deterministically evolve according to one of the two expressions (rule (sum)). Rule ($struct$) deals with structural equivalence, and the last rule (pat) lifts actions as label from behavioural expressions to interaction patterns. As expressed by the rules, an interaction pattern alone does not modify its open variables. As usual, $\rightarrow^+$ indicates the transitive closure of $\rightarrow$, and $\rightarrow^*$ its reflexive and transitive closure.

Three main differences between IP-calculus and π-calculus must be noticed. First, *absence of intra-pattern communication:* dual actions of the same behavioural expression do not synchronise with each other and communication does not happen, since we are not interested in modeling intra-component interaction. The unusual $|||$ parallel operator reminds this difference. Second, *absence of recursion:* according to the premises explained in Section 3.1, components coordinate finite patterns of interactions, and hence, infinite, recursive behaviours are not considered. Third, *explicit (syntactical) treatment of open composition*, by means of open variables. This way of proceeding resembles *abstraction* of polyadic π-calculus [45], which uniformly binds both local (restricted) and input-bound names. In IP-calculus, instead, communication and connection binder are not represented uniformly, and can not be structurally nested into processes (since sub-processes do not interact with each other). Hence the architectural model is simpler. Finally, while abstractions instantiate variables all at once, open variables can be subsequently instantiated according to the dynamic nature of late bindings.

[2] According to semantics of Figure 3, values received are recorded by means of substitutions. For instance, the input of a value v in $in(c, X).out(X, d).0|||out(X, d).0$ would cause the substitution $[v/X]$ to be applied to the whole expression, erroneously *capturing* the free occurrence of X. The side condition prevents this requiring the α-renaming of the expression in $in(c, Y).out(Y, d).0|||out(X, d).0$.

3.3 Sessions

The problem of the verification of open architectures, also represented as *coordinators* (processes with variables ranging over processes), has not yet been satisfactory solved in the field of process algebras (see, for instance, [46]). According to the framework we propose, we do not aim at dealing with general coordinators, but rather with those which have the quite simple, *flat*, structure of parallel interaction patterns. Differently from the approach "process as variables", which requires global names, our approach instead explicitly models component connections and their sharing. To remark the differences with the standard notion of coordinator and to remind its finitary temporal dimension, the concept of (open) environment is formalised by the notion of *session*. Simply, a session is a set of interaction patterns. The union of the open variables of the patterns in a session are connections offered to new interaction patterns to join the session. The formal definitions of *open* and *closed* session follow.

Definition 3 (Session) *A session* $\mathcal{S} = \{(X_1)[E_1], ..., (X_n)[E_n]\}$ *is a finite, possibly empty, multi-set of interaction patterns.*

Definition 4 (Open and closed session) *A session*

$$\mathcal{S} = \{(X_1)[E_1], ..., (X_n)[E_n]\}$$

is closed *if* $\bigcup_{i=1}^{n} X_i = \emptyset$. *It is* open, *otherwise.*

A session can evolve either because of the joining of a new interaction pattern or because of the communications among interaction patterns already in it.

3.3.1 Accessing a session

The access of an interaction pattern to a session is done by assigning some of its open variables and some of those in the session to the same (channel) names, thus establishing connections. Connections are hence created according to a given *mapping*. Acceptance of the incoming interaction pattern may be subject to the satisfiability of correctness properties by the session which would result from the insertion of the new pattern. In particular, we have addressed the case in which acceptance prevents incoming patterns from introducing unrecoverable deadlocks. When interaction proceeds inside a session, further channel names can be shared by means of communication, as typically happens in π-calculus. Open sessions can then be seen as partially specified sessions, whose future history also depends on the components that will eventually join them and are at the moment unknown. Note that this partial specification can model the dynamism of a system, but also its incremental architectural

development.

Example 5 (Open session) *Considering the simple case of the WEB-server of Example 2, and the following WEB-browser that requires and downloads a page and also forwards it to another component:*

`(Www, One) [out(Www, url). in(Www, Data). out(One,Data).0]`

In practice, the needed connection is easily defined since both components adhere to the same protocol, agreeing on the meaning of an IP address. In general, mapping definition is a hard problem, beyond the scope of this paper. The two interaction patterns can be connected by mapping their open variables `MyAdd` *and* `Www` *into the same channel,* `ipc` *say, obtaining the open session:*

`{ () [in(ipc, page(P)). (`τ`.out(ipc, a_page).0 + `τ`.out(ipc, error).0)],`

`  (One) [out(ipc, url). in(ipc, Data). out(One, Data).0 ] }`

Given a session and a mapping, when an interaction pattern accesses the session the mapping is applied to all the open variables (substitution works as expected). Instantiated variables are not open anymore.

Definition 6 (Join) *Let* $\mathcal{S} = \{(X_1)[E_1], ..., (X_n)[E_n]\}$ *be a session, let* $(Y)[F]$ *be an interaction pattern (variable-disjoint with the session), and let* Ch^* *be a (finite) set of channel names disjoint with the ones already occurring in* $\mathcal{S}$*. Let*

$$\gamma : \bigcup_{i=1}^{n} X_i \cup Y \ \rightarrow \ Ch^*$$

be a partial mapping. Then:

$$join((Y)[F], \gamma, \mathcal{S}) = \bigcup_{i=1}^{n} \{(X_i \setminus dom(\gamma))[E_i\gamma]\} \ \cup \ \{(Y \setminus dom(\gamma))[F\gamma]\}.$$

The freshness restriction on Ch is to avoid name clashes, and the finiteness one is to avoid infinite branching, considering that only a finite number of channel sharing combination is possible, being the open variables finite. Simultaneous joining of more patterns can be reduced to this simple "unary" joining operator by sequentially joining a pattern at a time. Open variables corresponding to channels shared by more interaction patterns will be assigned only when all the relative patterns have joined the session.

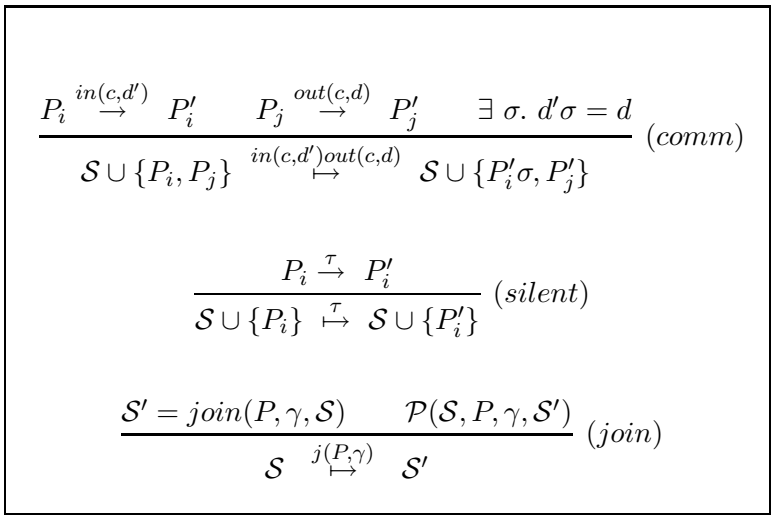

Fig. 3. Session semantics.

3.3.2 Session semantics

Semantics of sessions is given by the LTS $\mapsto$ in Figure 3.[3] Interaction inside sessions is modeled by synchronous communication actions occurring over a channel (rule *comm*): input (structured) data and output data must match. Trivially, the communication channel must be specified, i.e. communication does not occur through channel variables. Matching of data ($d'\sigma = d$) might also encompass type checking or other features relevant for the system under analysis, and determines a substitution σ, which is applied to the receiving interaction pattern. Rule *silent* models silent actions, while the not-yet-completely specified rule *join* models the joining of a new interaction pattern. We indicate with $\mapsto^+$ and $\mapsto^*$ the transitive closure and the reflexive and transitive closure of $\mapsto$ respectively, while $\not\mapsto$ stands for the absence of a rule to apply: $\mathcal{R} : \mathcal{S} \mapsto^* \mathcal{R}$ is a session reachable from $\mathcal{S}$, while $\mathcal{R} \not\mapsto$ is a deadlocked session. Labels of the $\mapsto$ relation may be omitted, when it is not necessary to distinguish the actions which permit the transitions to be fired. A sequence of sessions $\Delta = \mathcal{S}_0 \ldots \mathcal{S}_i \ldots$ such that $\forall i.\ \mathcal{S}_i \mapsto \mathcal{S}_{i+1}$ is a *trace* of the session $\mathcal{S}_0$. Sometimes, to make clear that we are referring to traces in which (*join*) rule is not applied, i.e. evolutions of a session without the contribution of new joining interaction patterns, we will use the symbol $\mapsto_{\not j}$.

Example 7 *The open session of Example 5, unless a new pattern joins it, can perform a communication step and evolve in the session*

```
{  () [τ.out(ipc, a_page).0 + τ.out(ipc, error).0],
   (One) [in(ipc, Data). out(One, Data).0 ] }
```

Intuitively speaking, this open session appears "properly-assembled" since, if

[3] As a by-product of this two level semantics, we obtain that the session model, its semantics and also its verification, is orthogonal to the IP-calculus.

an interaction patterns joins it, and accepts some `Data` *through the shared channel* `One`*, then all the components* at present *in the session can terminate, in every trace, achieving the tasks declared by their interaction patterns.*

Some remarks are worth to be made here:

(1) Session access is not a primitive action in the IP-calculus, but it appears as a (constrained) transition rule of session semantics (*join*). Keeping mapping mechanisms apart from the calculus, permits us to abstract from the problem of determining how components need to be connected, focusing instead on whether a given composition is admissible with respect to certain (correctness) properties. The calculus can be extended, for specific contexts in which mapping definition can easily be solved, by adding the necessary operator (whose semantics will be a variation of (*join*)).

(2) The join operator permits a session to evolve through infinitely many steps by the contribution of an infinite number of interaction patterns joining it. This respects our hypothesis of studying finite structures since:
   - we will reason about properties regarding the structure of a session at a given instant without considering possibly joining interaction patterns,
   - we will show how to preserve correctness properties along the (infinite) life of a session by finitely verifying the compatibility of the joining interaction patterns with those already in the session.

(3) The unspecified pre-condition $\mathcal{P}(\mathcal{S}, P, \gamma, \mathcal{S}')$ of rule *join* constrains the application of the rule to a generic (correctness) property of the session and the joining interaction pattern. In Section 4, it is shown how *acceptability* can be used as pre-condition for denying the access to those interaction patterns that may introduce unrecoverable errors in a session.

    More general properties, depending on the architecture, can be enforced, possibly by means of ad-hoc defined logics. Verification is, as in general, based on *finite model checking*, exploiting the benefits of analysing finite structures (see [47]). An initial attempt to enforce properties by means of only checking the kind of connections made, i.e. the mapping γ, has been studied in [43] with the aim of reducing the model checking of properties over computations to the more efficient verification of constraints over mappings.

4 Verification of open sessions

In this section we introduce a notion of (weak) correctness for open sessions, called *acceptability,* and show how it can be verified and enforced throughout the life of an open system. For the sake of presentation, we refer the reader to [48] for technical details and proofs of theorems. The practical verification of acceptability relies on the finiteness of the state space to be checked, stated

by the following proposition. The set of states that can be reached by the computations of a given session, without the joining of new interaction patterns, is finite.

Proposition 8 *Let $\mathcal{S}$ be a session. The set $\{\mathcal{S}' | \mathcal{S} \mapsto_{\mathcal{J}}^{*} \mathcal{S}'\}$ is finite.*[4]

4.1 Partial correctness for open sessions

The definition of acceptability for open architectures relies on a notion of *total correctness* for closed architectures, which requires *successful termination* (i.e. all the interaction patterns have reduced — up to structural equivalence — to the empty behaviour 0) for all the traces of a session.

Definition 9 (Totally correct session) *Given a closed session $\mathcal{S}$, it is* totally correct *if and only if:*

$$\forall\, \mathcal{R} :\ \mathcal{S} \mapsto_{\mathcal{J}}^{*} \mathcal{R}\ \wedge \mathcal{R} \not\mapsto_{\mathcal{J}}\ \Rightarrow\quad \mathcal{R} \text{ is successful}$$

The successful requirement can be relaxed in order to study other properties than termination. For example a state could be considered successful when some check-points have been reached or when only the components that are critical for the correct functioning of the system have terminated. This can be used to introduce a structure over the flat notion of session by means of a hierarchy of components, when required by the system under analysis. The requirement of totality of successful traces can be relaxed, too, for example in order to check whether a state satisfying a given property can be reached according to a (dual) notion of existential correctness (like done in [49], about the safety of security protocols). Total or existential correctness can be verified by running the session and analysing all its finite traces.

Open architecture analysis is more difficult because of their partially "unpredictable" future behaviour, which depends on the behaviour of the components that will participate in the system and that are at the moment unknown. Necessarily, as discussed in Section 2, only weaker assessments about correctness can be stated. According to these premises, acceptability is read as the "potential" correctness of the up to now constructed system, that we express as: "an open session is acceptable if an interaction pattern exists that can join the session making it closed and totally correct".

[4] Notice that the absence of recursion in the patterns plays a fundamental role here: the set of possible traces has a finite dimension. Obviously, Proposition 8 holds up to structural equivalence, in particular as far as α-renaming is concerned.

Definition 10 (Acceptable session) *A session $\mathcal{S}$, where the set $X_{\mathcal{S}}$ is the union of the open variables of its patterns, is* acceptable *if and only if an interaction pattern $(Y)[E]$, disjoint from $\mathcal{S}$, and a mapping γ from $(X_{\mathcal{S}} \cup Y)$ to a set of fresh names exist, such that:*

$$\mathcal{S}' = join((Y)[E], \gamma, \mathcal{S})$$

is a totally correct *session. The pattern $(Y)[E]$ is called a* completion *for $\mathcal{S}$.*

The above definition reduces the acceptability of open sessions to the correctness of closed ones, indeed the condition $dom(\gamma) = (X_{\mathcal{S}} \cup Y)$ implies that $\mathcal{S}'$ is a closed session. The completion $(Y)[E]$ represents, informally speaking, all the interaction necessary to the patterns already in the session for completing their coordinated tasks.

Example 11 (Acceptable and not-acceptable sessions) *The open session of Example 7, if joined by the interaction pattern* `(I) [in(I, Data).0 ]` *with the obvious matching* `[c/I, c/One]`*, becomes closed and totally correct, since its traces lead to successful termination:*

```
{  () [τ.out(ipc, a_page).0 + τ.out(ipc, error).0],
   () [in(ipc, Data). out(c, Data).0 ]
   () [in(c, Data).0 ] }
```

Let us now consider a server which can either receive a request or autonomously decide to suspend, after signaling a time-out:

```
(S)  [τ.in(S,Request).0 + τ.out(S, time_out).0 ]
```

and a proxy component which receives and forwards a request to the server, but is not able to react to a time out:

```
(W,V)  [in(W, A_request).out(V, A_request).0 ]
```

If we try to build a system out of the two, by means of the intended mapping `[c/S, c/V]`*, we get a not-acceptable session, where no forthcoming interaction pattern can react to the server time-out. Note that the server may autonomously send the time-out through the* `c` *channel, which is no longer accessible in the session:*

```
{ ()  [τ.in(c,Request).0 + τ.out(c, time_out).0 ]
  (W)  [in(W, A_request).out(c, A_request).0 ] }
```

On the other hand, the system built by the same server and a proxy component which is able to react to the server time-out, can be easily shown to be acceptable:

```
{ ()  [τ.in(c,Request).0 + τ.out(c, time_out).0 ]
  (W)  [in(W, A_request).(out(c, A_request).0 + in(c, Msg).0) ] }
```

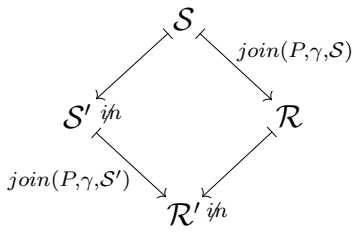

Fig. 4. Acceptability is preserved by join-free transitions.

4.2 Preserving acceptability

It is now possible to instantiate the rule $(join)$ according to the introduced notion of acceptability, so as to enforce compatibility of an interaction pattern with a running session: an interaction pattern is allowed to access the session only if this preserves the "potential" correctness of the session.

Definition 12 (*(in)* rule) *Let $\mathcal{S}$ be an acceptable session with $X_{\mathcal{S}}$ the union of the open variables of the interaction patterns in the session. Let $(Y)[E]$ be an interaction pattern, and let γ be a mapping from $V \subseteq (X_{\mathcal{S}} \cup Y_E)$ to a set of fresh names. The following rule (in) completes the semantics of sessions by instantiating rule $(join)$:*

$$\frac{\mathcal{S}' \;=\; join((Y)[E], \gamma, \mathcal{S}) \quad \mathcal{S}' \text{ is acceptable}}{\mathcal{S} \mapsto \mathcal{S}'} \; (in)$$

Intuitively speaking, acceptability regards the possible traces of an "isolated" session, to which rule $(join)$ can not be applied. Clearly, while $(join)$ can in principle spoil acceptability of a session, conversely rules $(comm)$, $(silent)$ and (in) preserve it. By using (in) in place of $(join)$, acceptability becomes an invariant property that holds throughout the potentially infinite life of a session continuously joined by "good" interaction patterns. The correctness of closed systems can be investigated by means of a finite amount of information, while about open systems, by means of a finite process, we can only check their acceptability at a given instant.

Proposition 13 (Subject reduction) *Let $\mathcal{S}$ be an acceptable session. If $\mathcal{S}' : \mathcal{S} \mapsto \mathcal{S}'$ by rule $(comm)$, $(silent)$ and (in) then $\mathcal{S}'$ is acceptable.*

Figure 4 informally shows that if P is a completion for $\mathcal{S}$ and $\mathcal{S}$ evolves to $\mathcal{S}'$ by means of $(comm)$ or $(silent)$, then, by definition of acceptability, P is a completion for $\mathcal{S}'$, too. Indeed P can join both $\mathcal{S}$ and $\mathcal{S}'$ with the same mapping, generating $\mathcal{R}$ and $\mathcal{R}'$. The totally correct $\mathcal{R}$ can evolve to $\mathcal{R}'$ (applying the same rule to the same interaction patterns applied in the transition $\mathcal{S} \to \mathcal{S}'$), which must also be totally correct.

Finally, our methodology for the design and verification of behavioural aspects in open systems is provided with a, proved correct, algorithmic procedure for acceptability checking, based on the synthesis of a possible completion. Such a completion can be incrementally constructed by nondeterministically exploring the finite space of the possible session evolutions. We present an abstract specification of the algorithm that, being inherently nondeterministic, can be naturally expressed in Prolog (see [48] for the full specification of the algorithm and the proof of the correctness theorem below stated). The algorithm at work is illustrated by some examples in Section 4.3.1.

Given a session `S`, the algorithm incrementally builds a completion `C` by trying to eliminate progressively all the deadlocks that may occur in the evolution of `S` $\oplus$ `C`. With `S` $\oplus$ `C` we indicate the algorithm internal representation of the session joined by the under-construction completion `C`, which has access to all the open channels in the session. Informally, while the derivation tree of the session `S` $\oplus$ `C` contains a deadlock, the algorithm extends `C` with an action that will trigger one of the deadlock states. Due to its specific function, a completion can be constructed by using only sequential composition and nondeterministic choice of actions. For the sake of readability, we focus on the main skeleton of the algorithm, and skip some obvious predicate definitions, in favor of a shorter informal explanation. The top-level predicate is:

```
find_completion(Session,Completion,LastAction,NewCompletion)/4
```

which, given a session (`Session`), a partially constructed completion (`Completion`), and the last action added to the completion (`LastAction`) returns a completion (`NewCompletion`) for the given session, if there exists one, otherwise it fails. The predicate is invoked by the query:

```
? find_completion(ctxt, [], [], Comp).
```

where the first argument is initially bound to a term (`ctxt`) representing the session `S`, the second parameter, `[]`, represents a null completion, and the third one states that no action has been added yet to the completion. `Comp`, in case of `S` being acceptable, will be instantiated with (the behavioural expression of) a completion for `S`. The definition of the predicate follows.

```
find_completion(Session,Comp,_,Comp) :-
    deadlocks(Session ⊕ Comp,[]).

find_completion(Session,Comp,Last,NewComp) :-
    deadlocks(Session ⊕ Comp,[_|_]),
```

```
        evols_after_last(Session ⊕ Comp,Last,Es),
        deadlocks(Es,[D|Ds]),
        successes(Es,[]),
        triggers([D|Ds],A),
        insert(A,Last,Comp,Comp1),
        find_completion(Session,Comp1,A,NewComp).

find_completion(Session,Comp,Last,NewComp) :-
        deadlocks(Session ⊕ Comp,[_|_]),
        evols_after_last(Session ⊕ Comp,Last,Es),
        deadlocks(Es,[]),
        prefix(Last,Comp,NewLast),
        find_completion(Session,Comp,NewLast,NewComp).
```

The first rule specifies that if the session together with the completion (`Session` $\oplus$ `Comp`) is deadlock-free, then `Comp` itself is a completion, and the construction successfully ends as there are no more deadlocks to solve. Deadlock freedom is checked by predicate `deadlocks/2` whose first argument is a session, and which returns the possible deadlock states, as one may expect.

Instead, if the session together with the completion may deadlock, then either the second or the third rule is invoked, depending on whether the introduction of the last action `Last` in `Comp` still produces deadlocks, or all the states reachable via `Last` lead to success (i.e. they are deadlock-free). More precisely, in presence of deadlocks in the whole derivation tree, all the states reachable via `Last` (i.e., through a trace in which `Last` occurs) are collected in the set `Es` by the predicate `evols_after_last/3`. If `Last` is the empty list `[]`, like at the beginning of the construction, then the states of the whole derivation tree are collected in `Es`. The set `Es` is then checked for the presence of deadlocks.

This way of proceeding is motivated by the strategy adopted in implementing the incremental construction of `Comp`: All the new transitions introduced by expanding `Comp` with `Last` are immediately checked against deadlocks (and possibly `Comp` is in-depth recursively expanded). The goal is to discover bad choices, which will eventually lead to deadlocks, as soon as possible.

If there are deadlocks in the set `Es` (`deadlocks(Es,[D|Ds])`), then the second rule for `find_completion/4` applies, provided that the set of successful states in `Es` is empty (predicate `success/2`), otherwise, in presence of both deadlocks and successes, the current branch of resolution fails. Indeed, any attempt to add to `Comp` an action "after" `Last` in order to solve a deadlock state, unavoidably spoils the already successful states, reached doing action `Last`. If `Es` does not contain successful states, an action `A` capable of triggering one of the deadlock states in `Es` is nondeterministically chosen (predicate

`triggers([D|Ds],A)`), if it exists, and used to expand the completion (predicate `insert(A, Last, Comp, Comp1)`) as one of the possible actions following `Last`. More precisely, the trigger is added as a term of the sum of the actions following `last`, like in, e.g., `last. (...+ A +...)`. If there are no triggers the current instance of the construction process fails, otherwise it continues with a tail recursion, being now `A` in place of `Last`.

The third rule for `find_completion/4` applies when there are still deadlocks in the derivation tree of `(Session` $\oplus$ `Comp)`, but not in `Es`. In such case, there is no point in trying to expand further the completion "below" `Last`. Hence, the process tries to continue by considering the action that prefixes `Last` in the completion. The construction backtracks one step behind (predicate `prefix/3`), and the action preceding `Last` is now considered the last one. Recursively, it is checked whether the action under consideration needs further expansions, or all the traces in which it occurs lead to successful states.

Backtrack ends when the action `[]` (i.e. the root of the completion `Comp` term) is reached and all the states reachable via it (i.e. all the states of the derivation tree) are checked. If no deadlock occurs, `Comp` is returned, otherwise the construction fails when all the possible triggers have been tried without success.

The completion construction algorithm has been proved correct according to the property: "the algorithm always terminates and it returns a completion if and only if the session is acceptable," [48]. Formally:

Theorem 14 (Correctness) *Given a session* $\mathcal{S}$ *and its representation* `ctxt`*,*

(1) `? find_completion(ctxt, [], [], Comp).` *always terminates.*

(2) If `? find_completion(ctxt, [], [], Comp).` *returns* `Comp = P` *as a computed answer substitution, then* $\mathcal{S}$ *is acceptable, and* `P` *is a completion.*

(3) If $\mathcal{S}$ *is acceptable then* `? find_completion(ctxt, [], [], Comp).` *does not fail (it does not answer* `no.`*).*

4.3.1 Constructing completions: examples

A few examples in this section illustrate how the algorithm works, giving an intuition about the problems which may arise in constructing completions. Examples refer to communication actions as $\alpha, \bar{\alpha}, \beta, ...$, where $\bar{\alpha}$ is the dual action of α (remember that the completion has access to all the channels corresponding to open variables in the session), and the top element of the completion term is referred to as *root*. The computation carried on by the algorithm is graphically represented by a figure, where each iteration corresponds to a boxed area. Each box contains:

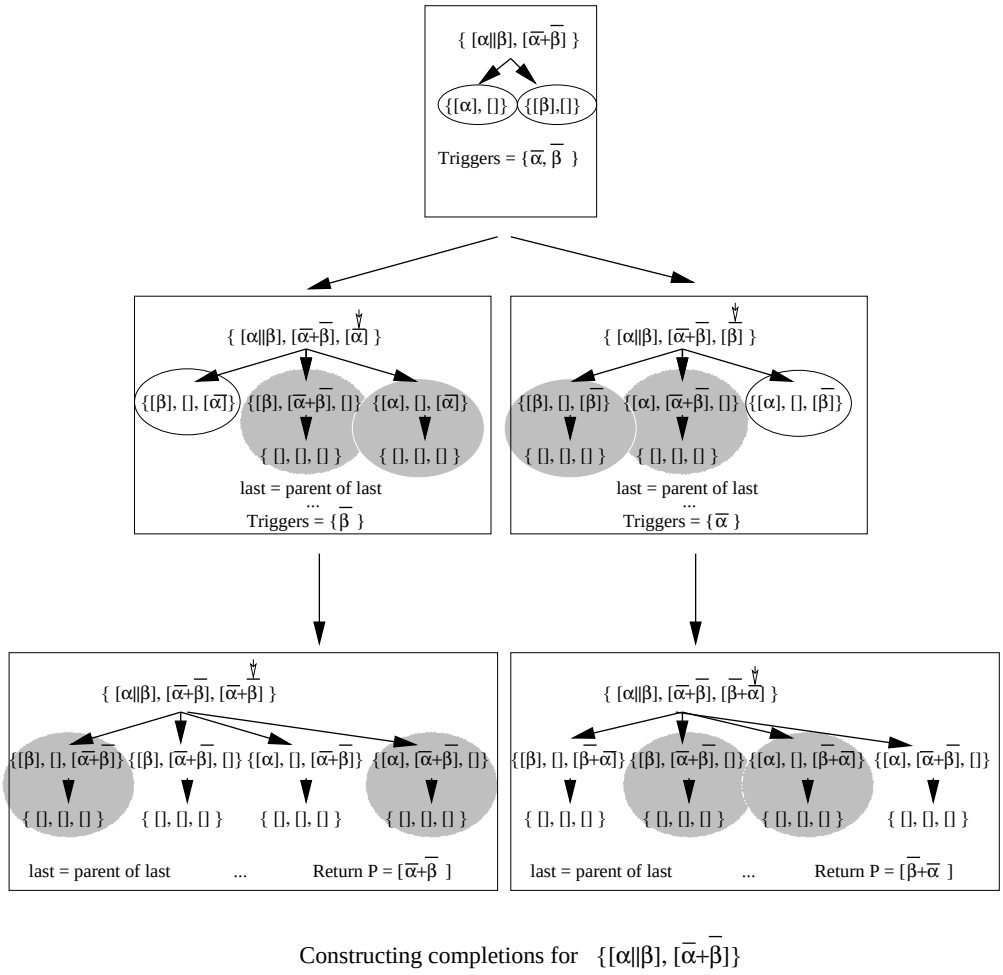

Fig. 5. Completion construction - I.

(1) The session and the so far expanded completion. The `last` action of the completion is marked by an arrow.
(2) The traces of such a session in the form of a derivation tree, where deadlock states are circled.
(3) The conditions that justify the next step (like FAIL, or RETURN), or the set of triggers. Each nondeterministically chosen trigger generates a child box (the Prolog proof procedure exhaustively explores the state space).

Checking for the acceptability of the session formed by the interaction patterns $[\alpha|||\beta]$, and $[\bar{\alpha} + \bar{\beta}]$, Figure 5, works as follows. The session evolves into two deadlocked sessions, offering the triggers $\bar{\alpha}$ and $\bar{\beta}$, respectively (top box). The second rule of `find_completion/4` applies, and one of the two triggers is nondeterministically chosen. Let us follow the case for the trigger $\bar{\alpha}$ (the leftmost branch, the other case is symmetric). Having added $\bar{\alpha}$ as *last* element to the completion constructed so-far, five sessions are now reachable. The evolutions after $\bar{\alpha}$, shaded in the figure, do not contain deadlocks, so *last* is backtracked until *root*, where the only deadlocked session offers the trigger $\bar{\beta}$. After having

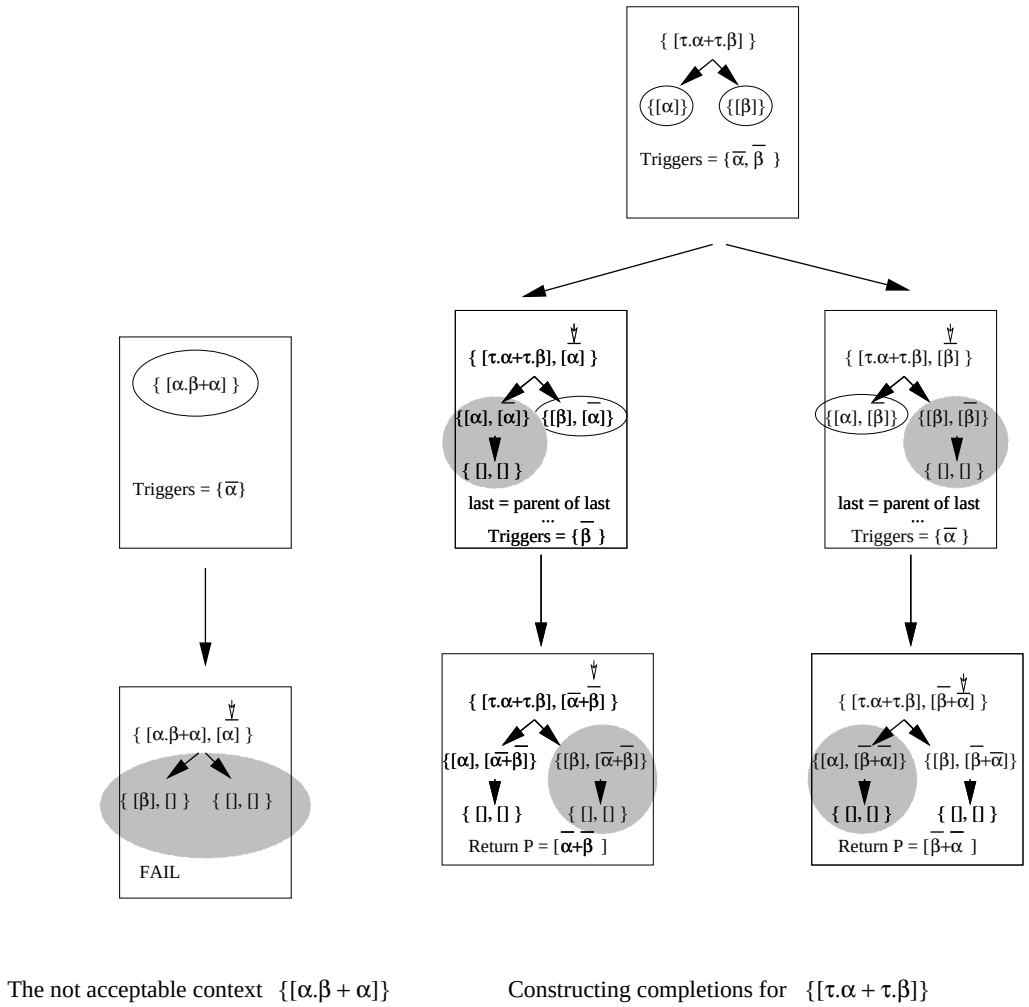

Fig. 6. Completion construction - II.

added $\bar{\beta}$ to the completion as a child of *root*, the evolutions after $\bar{\beta}$ first, and then all the reachable sessions, are checked in two backtracking steps. Since they do not present deadlocks, the completion $[\bar{\alpha} + \bar{\beta}]$ is returned.

The only trigger for the session $\{[\alpha.\beta + \alpha]\}$, $\bar{\alpha}$, generates two traces, one is successful, and the other one leads to a deadlock, Figure 6(a). In this case only the third rule of `find_completion/5` applies. No further action can be added as child of $last = \bar{\alpha}$, since it would spoil the successful trace, neither an action could be added at the same level of $\bar{\alpha}$, since it could not trigger any action of the session. Hence, no completion exists for the given session, and the algorithm fails, returning `no.`, since no rule can be applied.

In Figure 6(b) we can see how local choices are addressed during completion construction. The session $\{[\tau.\alpha + \tau.\beta]\}$ can autonomously decide to evolve into

two deadlocked sessions $\{[\alpha]\}$ and $\{[\beta]\}$. Each of them offers a trigger, nondeterministically generating two different computations. In the leftmost the trigger $\bar{\alpha}$ is chosen. The continuations after $\bar{\alpha}$ consist of one successful session. Hence *last* backtracks until assuming the value *root*. Now, a deadlocked session with trigger $\bar{\beta}$ can be reached as continuation after *root*. Then, $\bar{\beta}$ is added as a child of *root*, and since there are not deadlocks, the completion $[\bar{\alpha} + \bar{\beta}]$ is returned in the left bottom step of this derivation. The other derivation returns the structurally equivalent completion $[\bar{\beta} + \bar{\alpha}]$.

5 Concluding Remarks

The development of complex software applications requires to be addressed at the architectural level, by identifying the components of a system and their interactions. The problem of coordinating the interactive behaviour of autonomous components has motivated a large effort in the research of suitable models and formal verification techniques. We have discussed which are the more significant issues for the definition of such models, and how they are addressed by different research fields, suggesting that all of them may contribute to a general approach to the problem. However, many issues are still open. We have illustrated how the same problems become more difficult when addressed within the context of open systems, which seems to be the mainstream for future software development. In this case, the incomplete accessibility of the environment where components operate makes the traditional analysis techniques inadequate.

Building on the natural capability of process algebra to describe component interactions, and π-calculus in particular for dynamic aspects, we propose a methodology to support the verification of the *behavioural compatibility* of components in open systems. The methodology consists of

(1) The *IP-calculus*, to express component behaviour. The choice to model only temporal finite patterns of interaction is motivated by the intrinsic incompleteness of open systems, which not always can be described in all their possible evolutions, and by reasons of practical effectiveness. This choice supports the definition of our notion of (partial) correctness, and its verification by means of finite state model checking techniques.
(2) An explicit mechanism to describe the dynamic composition of component behaviours into an open system, represented as a *session* where components interact with each other (namely, *open variables* and *join/in* semantical operations).
(3) The notion of *acceptability*, which models the (partial) correctness of the system that can at present be accessed. Acceptability may practically be used to enforce such correctness throughout the, potentially infinite, life

of an open system, by preventing the access to harmful components.

(4) An *algorithmic procedure*, shown to be correct, which verifies acceptability, by synthesizing, if it exists, a *completion* for a given session. It provides the computational basis on which the theoretic model can be developed into an actual software architecture.

Our proposal exports successful techniques used in the context of the analysis and verification of behavioural aspects of software systems to the context of open systems. As recognized by influential researchers in the field, this step requires the reconsiderations of notions and techniques, mainly in order to deal with the inaccessibility of an open system. Accordingly, the original, at the best of our knowledge, notion of acceptability we introduce reconsiders a traditional notion of correctness (absence of deadlock), as a partial correctness property that can be checked against the current partial specification of the system. If, on the one hand, acceptability cannot guarantee the overall correctness of a non-completely specified system, on the other hand, it helps preventing local errors in the so-far specified system and in its current evolution, guaranteeing that *"the system is sufficiently good for the tasks at hand,... and the analysis [is done] at a reasonable cost"* [38]. Coherently, the finiteness assumptions on which the model is based allow for effective verification techniques.

At the moment, the model we devised has been applied to two different scenarios: *component adaptation* and *security protocols analysis*. The first is an open problem in CBSE consisting in adapting the mismatching behaviour of components by means of an in-the-middle component, called *adaptor*, in charge of facilitating their interoperability. Our methodology naturally applies to the problem, being the adaptor a "specialised" completion for the session containing the mismatching components. Once that the needed adaptation has been formalised by means of a high-level language we defined, an adaptor can be automatically derived by a variant of the completion construction algorithm, so separating the abstract design of adaptation, from the actual construction process. Correctness, hence, is no longer obtained by forbidding interactions among mismatching components, but, rather, by facilitating their interaction towards successful termination, [50,51]. The framework also naturally applies to security protocol analysis, consisting of a finite interaction in an untrusted open environment that plays the part of an intruder trying to attack the safety of the protocol. Reading references as cryptographic keys, instead of channel names, protocol participants can be modeled as interaction patterns, while the safety of the protocol is read as the impossibility of the environment to drive the execution towards an error condition (expressed by an ad-hoc logic), [43,49,48].

After summarising our results, we briefly comment on how they are related with other existing approaches in the field. We have adopted the approach of extending component descriptions with behavioural concerns common to SA

and CBSE, following ideas also developed by CO. About the works in SA, closer to our architectural level of abstraction, we have followed the approach that does not distinguish between components and connectors, like [21] and differently, for instance, from [18,32]. We have followed the approaches based on name-based calculi, like [42,21], rather than on finite state algebras, in order to account for some forms of dynamic reconfiguration. The main difference with ADLs, which aim at facilitating the engineering process of building systems, is our interest in the dynamics of systems which are open. In this respect, our approach to verification is, at the best of our knowledge, new with respect to ADLs, even if, as explained, it responds to needs already clearly recognised in the field [38]. Moreover, we can still import useful notions from SA, in particular the notions of inheritance and substitutability reformulated so as to preserve acceptability. About this topic, we can also take inspiration from the huge literature about system equivalence in process calculi. In particular, our approach bears similarities with the theory of Testing Equivalences, [52], where processes are compared according to their behaviour with respect to a set of *observers*, which may or may not drive them to (successful) termination. Notion of equivalences are given for processes that are undistinguishable with respect to the set of all the observers. In this perspective, our notion of completion can be seen as playing the role of an observer of the session, which is able to drive the session to successful termination. This similarity could be exploited to devise a notion of equivalent open systems, as those systems which are equivalent with respect to the same set of completions. Finally, the mix of acceptability and completions has also contributed to improve CBSE, when the model has been applied to adaptor development, see [51].

We envisage several directions for extending our proposal, both theoretically and practically. Each one of the completions returned by the algorithm represents, informally speaking, an amount of interaction expected by the session. It would be interesting to study a hierarchy of them, possibly based on acceptability preserving notions of inheritance and compatibility, e.g. along the line of [21], in order to define one or more representative completions for a session. Preliminary investigations show that the completions non-deterministically returned by the algorithm may represent not easily comparable minimal elements. For instance, the completions for $()[\alpha + \beta]$ include both $[\bar{\alpha}]$ and $[\bar{\beta}]$, which could be both "represented" by $[\bar{\alpha} + \bar{\beta}]$, a sort of least upper bound. Unfortunately, the definition of a "complete" upper bound seems to require the construction of all the "minimal" completions, thus impacting on the efficiency of the non-deterministic algorithm.

Another immediate line of research is to investigate if it is worth adding a structure *à la* ambient-calculus [6] to sessions, which are now flat structures, and to study which systems can be modeled with this extension. The practical experimentation on actual component architectures would also represent a valuable test for our proposal, and a possible enhancement to current tech-

nologies in order to systematically afford component interaction.

Acknowledgements

We would like to thank the anonymous referees for their valuable suggestions that have contributed to improve the paper.

References

[1] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I and II, Information and Computation 100 (1) (1992) 1–40,41–77.

[2] D. Sangiorgi, Expressing mobility in process algebras: First-order and higher-order paradigms, Ph.D. thesis, Department of Computer Science, University of Edinburgh (1992).

[3] U. Goltz, R. Gorrieri, A. Rensink, Comparing syntactic and semantic action refinement, Information and Computation 125 (2) (1996) 118–143.

[4] M. Abadi, A. D. Gordon, A calculus for cryptographic protocols: The spi calculus, Information and Computation 148 (1) (1999) 1–70.

[5] R. De Nicola, G. Ferrari, R. Pugliese, KLAIM: A kernel language for agents interaction and mobility, IEEE Transactions on Software Engineering 24 (5) (1998) 315–330, special Issue: Mobility and Network Aware Computing.

[6] L. Cardelli, A. D. Gordon, Mobile ambients, TCS: Theoretical Computer Science 240.

[7] D. Garlan, D. E. Perry, Introduction to the special issue on software architecture, Software Engineering 21 (4) (1995) 269–274.

[8] N. Carriero, D. Gelernter, Coordination languages and their significance, Communications of the ACM 35 (2) (1992) 97–107.

[9] G. Papadopoulos, F. Arbab, Coordination models and languages, Advances in Computers 46.

[10] S. Ahuja, N. Carriero, D. Gelernter, Linda and friends, IEEE Computer 19 (8) (1986) 26–34.

[11] N. Sample, D. Beringer, L. Melloul, G. Widerhold, Clam: Composition language for autonomous megamodules, in: Coordination Languages and Models - COORDINATION'99, Vol. 1594 of LNCS, Springer-Verlag, (1999).

[12] J. Cruz, S. Ducasse, A group based approach for coordinating active objects, in: Coordination Languages and Models - COORDINATION'99, Vol. 1594 of LNCS, Springer-Verlag, (1999).

[13] C. Szyperski, Component Software - Beyond Object-Oriented Programming, Addison-Wesley, (1997).

[14] D. Garlan, M. Shaw, An Introduction to Software Architecture, in: V.Ambriola, G.Tortora (Eds.), Advances in Software Engineering, Volume I, World Scientific Publishing Company, New Jersey, (1993).

[15] C. Gunter, J. Mitchell, Strategic directions in software engineering and programming languages, ACM Computing Surveys 28 (4) (1996) 727–737.

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Massachusetts, (1994).

[17] D. Garlan, Research directions in software architecture, ACM Computing Surveys 27 (2) (1995) 257–261.

[18] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, G. Zelesnik, Abstractions for software architecture and tools to support them, IEEE Transactions on Software Engineering 21 (4) (1995) 314–335.

[19] M. Shaw, D. Garlan, Software Architecture. Perspectives on an emerging discipline, Prentice Hall, Upper Saddle River, NJ 07458, (1996).

[20] R. Allen, D. Garlan, A formal basis for architectural connection, ACM Transactions on Software Engineering and Methodology 6 (3) (1997) 213–249.

[21] C. Canal Velasco, Un lenguaje para la especificación y validación de arquitecturas de software, Ph.D. thesis, Universidad de Malaga (2000).

[22] C. Canal, E. Pimentel, J. M. Troya, Specification and refinement of dynamic software architectures, in: Software Architecture, Kluwer (1999) 107–126.

[23] N. Medvidovic, R. N. Taylor, A framework for classifying and comparing architecture description languages, in: M. Jazayeri, H. Schauer (Eds.), Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), Springer (1997) 60–76.

[24] CMU SEI Software Engineerig Institute, Architecture Description Languages, `http://www.sei.cmu.edu/architecture/adl.html`.

[25] G. Leavens, M. Sitaraman (Eds.), Proceedings of the ESEC/FSE- Workshop on Foundations of Component-Based Systems (FoCBS), Zurich, (1997).

[26] G. T. Leavens, M. Sitaraman (Eds.), Foundations of Component-Based Systems, Cambridge University Press, NY, (2000).

[27] C. Canal, L. Fuentes, J. Troya, A. Vallecillo, Adding semantic information to IDLs. is it really practical?, in: Proceedings of the OOPSLA'99 Workshop on Behavioral Semantics, Denver, Colorado, (1999).

[28] OMG, Concurrency Control CORBAservice, Object Management Group, `http://www.omg.org/techonology/documents/formal/...`
`.../concurrency_service.htm`.

[29] SUN, JavaBeans Documentation, http://java.sun.com/beans/docs/.

[30] C. A. R. Hoare, Communicating sequential processes, Communications of the ACM 21 (8) (1978) 666–677.

[31] R. Milner, Communication and concurrency, Prentice Hall, (1989).

[32] R. Allen, A formal approach to software architecture, Ph.D. thesis, Carnegie Mellon, School of Computer Science, issued as CMU Technical Report CMU-CS-97-144, (1997).

[33] P. Inverardi, A. Wolf, D. Yankelevich, Static checking of system behaviors using derived component assumptions, ACM Transactions on Software Engineering and Methodology 9 (3) (2000) 239–272.

[34] F. Arbab, I. Herman, P. Spilling, An overview of manifold and its implementations, Concurrency: Practice and Experience 5 (1) (1993) 23–70.

[35] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, W. Mann, Specification and analysis of system architecture using Rapide, IEEE Transactions on Software Engineering 21 (4) (1995) 336–355.

[36] L. Andrade, J. Fiadeiro, J. Gouveia, G. Koutsoukos, M. Wermelinger, Coordination for orchestration, in: F. Arbab, C. Talcott (Eds.), Proc. 5th International Conference on Coordination Models and Languages, (COORDINATION 2002), Vol. 2315 of LNCS, Springer, York, (2002) 5–13.

[37] K. Gottschalk, S. Graham, H. Kreger, J. Snell, Introduction to web services architecture, IBM System Journal 41 (2).

[38] M. Shaw, *Architectural Requirements for Computing with Coalitions of Resources*, position paper for First Working IFIP Conference on Software Architecture (1999). Available at http://www.cs.cmu.edu/... ...~Vit/paper_abstracts/ShawCoalitions_paper.html

[39] A. Bracciali, A. Brogi, F. Turini, Coordinating interaction patterns, in: Proceedings of the *ACM Symposium on Applied Computing (SAC'2001)*, Las Vegas, USA, ACM Press, (2001).

[40] L. Feijs, Modelling Microsof COM using π-calculus, in: Formal Methods'99, no. 1709 in LNCS, Springer, (1999) 1343–1363.

[41] M. Gaspari, G. Zavattaro, A process algebraic specification of the new asynchronous CORBA messaging service, in: Proceedings of ECOOP'99, no. 1628 in LNCS, Springer, (1999) 495–518.

[42] J. Magee, S. Eisenbach, J. Kramer, Modeling Darwin in the π-calculus, in: Theory and Practice in Distributed Systems, no. 938 in LNCS, Springer, (1995) 133–152.

[43] A. Bracciali, A. Brogi, G. Ferrari, E. Tuosto, Security and dynamic composition of open systems, in: F. Arbab, R. Menezes (Eds.), Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA02), CSREA Press, Las Vegas, NV, (2002) 1372–1377.

[44] G. Plotkin, A structural approach to operational semantics, Tech. Rep. DAIMI FN-19, Aarhus University, Computer Science Department (1981).

[45] R. Milner, The polyadic pi-calculus: a tutorial, in: F. L. Bauer, W. Brauer, H. Schwichtenberg (Eds.), Logic and Algebra of Specification, Springer-Verlag, (1993) 203–246.

[46] P. Baldan, A. Bracciali, R. Bruni, Bisimulation by unification, in: H. Kirchner, C. Ringeissen (Eds.), Proc. 9th International Conference on Algebraic Methodology And Software Technology (AMAST'2002), Vol. 2422 of LNCS, Springer, Reunion Island, (2002) 254–270.

[47] E. Clarke, J. Wing, et al., Formal methods: State of the art and future directions, ACM Compitung Surveys 28 (4) (1996) 626–643.

[48] A. Bracciali, Behavioural patterns and software composition, Ph.D. thesis, Dipartimento di Informatica, Università di Pisa (2003), available at `http://www.di.unipi.it/phd`

[49] A. Bracciali, A. Brogi, G. Ferrari, E. Tuosto, Security issues in component-based design, in: U. Montanari, V. Sassone (Eds.), Proc. ConCoord, International Worshop on Concurrency and Coordination, Vol. 54, ENTCS (2001).

[50] A. Bracciali, A. Brogi, C. Canal, Adapting components with mismatching behaviours, in: First International IFIP/ACM Working Conference on Component Deployment (CD 2002), Vol. 2370 of LNCS, Springer, Berlin, (2002) 185–199.

[51] A. Bracciali, A. Brogi, C. Canal, A formal approach to component adaptation, Elsevier Journal of Systems and Software. Special Issue on Automated Component-Based Software Engineering. To appear (2004).

[52] R. D. Nicola, M. C. B. Hennessy, Testing equivalences for processes, Theoretical Computer Science 34 (1-2) (1984) 83–133.