

Interaction Patterns for Open Systems

Andrea Bracciali Antonio Brogi Franco Turini
Department of Computer Science
University of Pisa, Italy
{braccia,brogi,turini}@di.unipi.it

Abstract

The ability to describe and verify the concurrent behaviour of interacting components is a key aspect in the development of large component-based software systems. We present a simple interface description language which allows software designers to easily specify the interaction pattern of a component that will have to interact with other components. A set of interaction patterns forms a context which may evolve either because of interactions occurring within the context, or because new components dynamically join the context. We show how interaction patterns provide an effective means to efficiently reason on the (partial) correctness of open interacting systems.

1 Introduction.

A major limitation of available component-oriented platforms (such as CORBA [2], DCOM [7] or JavaBeans [1]) is that they do not provide suitable means for describing and reasoning on the concurrent behaviour of interacting component-based systems. Indeed while these platforms provide convenient ways to describe the typed signatures of components via *interface description languages*, they offer a quite low-level support to describe the concurrent behaviour of components.

Several proposals have been put forward in order to enhance component interfaces with a description of their concurrent behaviour. Most of them are based on process algebras languages, such as π -calculus [11], and extend interfaces with behavioural descriptions, such as behavioural types [12] or role-based representation of behaviours [5]. On the one hand, these proposals allow one to prove correctness properties, such as absence of deadlocks, as well as to define compatibility and substitutability relations for pairs of components. The techniques for reasoning on and verifying the resulting systems are typically co-inductive [10], based on the (stepwise) observation of the evolution of potentially non-terminating systems, and rely on the bisimulation relation or on (temporal) modal logics [8, 9, 13]. On the other hand, the major limit of these approaches is the computational cost of proving such properties, which in most cases falls into the class of NP problems, hence preventing their practical usability.

Following [4], and in contrast with the above mentioned proposal, we adopt a radically different approach to facing the complexity of describing and reasoning on the interactions occurring in a distributed system. We argue that trying to describe all the aspects of a distributed system in one shot unavoidably leads to complex formulations of low practical usability. We will instead focus on descriptions of the *finite* concurrent behaviours of components, that we will call *interaction patterns*. Intuitively speaking, an interaction pattern describes only the essential aspects of the interactive behaviour that a component may (repeatedly) show to the external environment.

Software designers can specify the dynamic behaviour of a component (i.e., its interaction pattern) by means of a simple *interface description language*. The software development process then consists of combining separate components into *contexts* of interacting components. A context is a set of interaction patterns which may

evolve both because of the interactions occurring between different components and because new components dynamically join the context.

The ultimate objective of introducing interaction patterns and contexts is to pave the way for the efficient verification of properties of complex interacting systems. We will show how the choice of considering simple fragments of interaction makes the verification of several interesting properties of open systems tractable. More precisely, the type of questions we are interested in is: “*What happens if this component is inserted in the current context ?*”.

It is worth noting that a context represents, in general, an *open* system of interacting components, where new components may join the system while the latter is already running. Typical notions of correctness employed in closed systems are of limited interest for the case of open systems. For instance determining whether a context C is deadlock-free is not quite significant as the introduction of a new component in C may introduce or remove the possibility of a deadlock.

The notion of *feasible* context was introduced in [4] to characterise the (partial) correctness of an open system. Intuitively speaking, the introduction of an interaction pattern P in a context C is considered correct if the resulting context is feasible, that is, if there exists some interaction pattern Q which can make the new context deadlock-free. In other words, the notion of feasibility guarantees that the introduction of a new component does not introduce a deadlock possibility which no other component will be able to remove.

Determining whether a context is feasible or not does not however give any extra information about the relation between the old context and the new context. Typically, while constructing a complex system, one would like to check whether the introduction of a new component somehow makes the system “more complete” or not. The aim of this paper is exactly to contribute to answering this type of questions. More precisely, we will extend the notion of feasibility of [4] in order to define a partial order on the behaviours of contexts.

Throughout the paper, we will try to exemplify the applicability of our approach by discussing a few significant (though simple because of space limits) examples.

2 Behavioural description of open systems.

2.1 Interaction patterns.

As we already anticipated in the Introduction, we first introduce a simple interface description language to express the *interaction pattern* characterising the behaviour of a component. Intuitively speaking, an interaction pattern describes only the essential aspects of the interactive behaviour that a component may (repeatedly) exhibit to the external environment. Syntactically speaking, the interface description language is a subset of π -calculus [11] with the addition of a (derived) event-handling operator.

To help intuition, consider a simple client which sends a request to some server on a communication channel ($_s$ say) and then waits for an answer from the same channel. (When necessary, channel names will be preceded by underscore to distinguish them from application data.) The external behaviour of such a component can be expressed by means of the expression:

$$[\text{out}(_s, \text{query}) . \text{in}(_s, \text{answer}) . 0]$$

where “.” denotes sequencing and 0 the empty action. A more sophisticated client may wish to avoid the risk of indefinitely waiting for an answer, and be able to choose to send a *break* message to cancel her request. Such a behaviour can be expressed by expression:

$$[\text{out}(_s, \text{query}) . (\text{in}(_s, \text{answer}) . 0 + \tau . \text{out}(_s, \text{break}) . 0)]$$

where the (silent) action τ is used (along with the choice operator +) to express a local choice of the component (e.g., detecting an internal timeout in this case).

Besides describing the fragment of behaviour repeatedly exhibited by a component, an interaction pattern also describes the *connectors* (viz., the channels) via which other components may interact with it.

Definition 1 (Interaction pattern) An interaction pattern is a formula $(\bar{X}_E)[E]$, where E is a behavioural expression, and \bar{X}_E is a set of open names. The open names of an expression are a subset of the free names¹ of the expression.

An *interaction pattern* consists of a behavioural expression together with the explicit declaration of its *open names*, that is, the channels that are initially visible by its external environment. For instance, the interaction pattern for the previous simple client is:

$$(_s) [\text{out}(_s, \text{query}) . \text{in}(_s, \text{answer}) . 0]$$

where $_s$ is the only open name of the pattern.

The *behavioural expression* in an interaction pattern is a composition of synchronous communication actions and of silent actions τ . Two communication operations – $\text{in}(x, d)$ and $\text{out}(x, d)$ – are introduced to express input and output actions, respectively, where x is a communication channel name. The second argument d can be either a channel name or application *data*, like 2, break, \bar{w} or any typed expression. Application data may also contain channel names, like $_x$ in $\text{query}(\bar{w}, _x)$.

Actions can be composed by means of the standard prefix (\cdot), parallel (\parallel), and nondeterministic choice ($+$) operators, while 0 denotes the empty behaviour. A behavioural expression can be also prefixed by the silent action τ , which is introduced to explicitly describe local choices of an interaction pattern. Intuitively speaking, the expression $(E + F)$ describes a component that may behave either as E or as F (depending on the other components available in the context). On the other hand, the expression $(\tau.E + \tau.F)$ describes a component that autonomously decides to behave either as E or as F , after performing an internal choice (τ).

Finally, a derived event-handling operator ($>$) is also introduced to model the possibility of a component to react to an external event. Intuitively, the expression $E > (\text{in}(x, d).F)$ will behave as E unless it receives an event d on channel x , in which case the expression will react to the event by behaving as F .

Definition 2 (The language of expressions) A behavioural expression E is defined as follows:

$$\begin{aligned} E & ::= 0 \mid A.E \mid E \parallel E \mid E + E \mid E > (\text{in}(x, d).E) \\ A & ::= \text{in}(x, d) \mid \text{out}(x, d) \mid \tau \end{aligned}$$

Before describing the operational semantics of interaction patterns, let us present a simple example to illustrate the syntax.

Example 1 Consider again the case of a client which sends a request to some server on a communication channel $(_s \text{ say})$ and then waits for an answer on the same channel. The client may also decide to send a break message to cancel her request, to avoid the risk of indefinitely waiting for an answer. Such a behaviour can be expressed by the following interaction pattern:

$$\text{CLIENT} = (_s) [\text{out}(_s, \text{query}) . (\text{in}(_s, \text{answer}) . 0 + \tau . \text{out}(_s, \text{break}) . 0)]$$

Similarly the interaction pattern:

$$\text{SERVER} = (_c) [\text{in}(_c, \text{query}) . (\text{out}(_c, \text{answer}) . 0 > \text{in}(_c, \text{break}) . 0)]$$

describes the behaviour of a server which is ready to receive a query on a channel $(_c)$, and then sends an answer on the same channel unless it receives a break. Although simple, the example illustrates the use of τ and $+$ to model internal (local) choices, and the use of $>$ to model a mechanism of event handling. The example also highlights the very idea of interaction patterns: Each time a server will interact with a client, it will do it following the same scheme, a sort of behavioural fingerprint. \diamond

¹The free names of an expression E are those names which occur in E not bound by an input operation [11]. For instance the name y is bound by the input operation $\text{in}(x, y)$ as well as by $\text{in}(x, m(d, y))$ where $m(d, y)$ is a method invocation.

Let us first describe the intensional behaviour of an interaction pattern, independently of the context in which it will be inserted. Intuitively speaking, such an intensional behaviour describes the communications that a pattern may perform and it can be naturally expressed by means of a transition relation \rightarrow defined by a set of inference rules. Most of the following rules correspond to the classical rules for π -calculus, the main difference being the absence of communication at this level of abstraction, since communication is not permitted between (parallel) processes of the same interaction pattern. The relation \rightarrow is defined up to structural congruence for the operators $+$ and \parallel as usually defined [11].

$$\frac{}{A.E \xrightarrow{A} E} \text{ (act)} \quad \frac{E \xrightarrow{A} E'}{E \parallel F \xrightarrow{A} E' \parallel F} (\parallel) \quad \frac{E \xrightarrow{A} E'}{E + F \xrightarrow{A} E'} (+)$$

$$\frac{E \xrightarrow{A} E'}{E > F \xrightarrow{A} E' > F} (>_1) \quad \frac{F \xrightarrow{A} F'}{E > F \xrightarrow{A} F'} (>_2) \quad \frac{}{0 > F \xrightarrow{\tau} 0} (>_3)$$

Rules (act), (\parallel), (+) are the standard rules for prefix, parallel and choice composition. Rules ($>_1$), ($>_2$), and ($>_3$) describe the $>$ operator. Rule ($>_1$) states that an expression $E > (in(x, d).G)$ may behave like E and evolve to $E' > (in(x, d).G)$ without reacting to the (possible) event d . Rule ($>_2$) instead models the event reactive behaviour of an expression $E > F$ which will behave as its event handling part F . Finally, rule ($>_3$) states that an event-handling expression that has terminated its non-reactive part can autonomously reduce to the empty process 0.

As implied by the above rules, the $>$ operator is derived, in the sense that any expression containing $>$ can be rewritten into an equivalent expression not containing $>$. For instance the expression $(A.B) > I$ can be viewed as a shorthand for $(A.(B + I)) + I$, as well as $(A \parallel B) > I$ is a convenient shorthand for $(A.(B + I)) + (B.(A + I)) + I$. Indeed the $>$ event-handling operator features a form of “may preemption” [3], in the sense that an expression $E > F$ may nondeterministically react or not to an occurring event. (Notice that this “may-preemption” is typical of time independent formalisms such as π -calculus, CCS, or CSP, where the semantics of a set of communicating processes does not depend on their relative timings.)

2.2 Contexts.

Interaction patterns may interact with one another within *contexts*. Simply stated, a context is a set of interaction patterns as stated by the next definition.

Definition 3 (Context) A context is a set of interaction patterns with disjoint open names.

Contexts can be constructed by inserting interaction patterns into existing contexts, starting from the empty context. Syntactically, this is performed by a *join* operator: Given a pattern, a name assignment, and an existing context, *join* yields a new context. The name assignment is used to bind together open names of the new pattern with open names of the patterns already present in the context. As a consequence of the assignment, the names that are bound by the assignment are not anymore open in the new context.

Definition 4 (Join operation) Let $\mathcal{C} = \{(\bar{X}_{P_1})[P_1], \dots, (\bar{X}_{P_n})[P_n]\}$ be a context and let $\bar{X}_{\mathcal{C}} = (\bar{X}_{P_1} \cup \dots \cup \bar{X}_{P_n})$ be the set of open names of (the patterns of) \mathcal{C} . Let $(\bar{X}_E)[E]$ be an interaction pattern whose names are disjoint from the names of \mathcal{C} . Let γ be a mapping from $\bar{D} \subseteq (\bar{X}_E \cup \bar{X}_{\mathcal{C}})$ to a set of fresh names. The new context:

$$\mathcal{C}' = \text{join}((\bar{X}_E)[E], \gamma, \mathcal{C})$$

is defined as follows²:

$$\{(\bar{X}_E - \bar{D})[E\gamma]\} \cup \{(\bar{X}_{P_1} - \bar{D})[P_1\gamma], \dots, (\bar{X}_{P_n} - \bar{D})[P_n\gamma]\}$$

²Strictly speaking, the general definition of *join* permits to simultaneously insert several patterns in a context. For the sake of simplicity, we will however use this restricted definition as it makes no difference in the scope of this paper.

where $P_i\gamma$ denotes the expression obtained by applying the substitution γ to P_i .

Example 2 Consider again the CLIENT and SERVER patterns of Example 1. Suppose that we firstly insert the SERVER pattern into the empty context, that is, let

$$\mathcal{C}_0 = \text{join}(\text{SERVER}, [], \{\}).$$

The context \mathcal{C}_0 simply contains the SERVER itself:

$$\{ (_c) [\text{in}(_c, \text{query}). (\text{out}(_c, \text{answer}).0 > \text{in}(_c, \text{break}).0)]] \}$$

If we now insert the CLIENT pattern as follows:

$$\mathcal{C}_1 = \text{join}(\text{CLIENT}, [_n/_s, _n/_c], \mathcal{C}_0),$$

where $[_x/_y]$ denotes the substitution of $_y$ by $_x$, we obtain the new context \mathcal{C}_1 :

$$\begin{aligned} & \{ () [\text{out}(_n, \text{query}). (\text{in}(_n, \text{answer}).0 + \tau.\text{out}(_n, \text{break}).0)]] , \\ & () [\text{in}(_n, \text{query}). (\text{out}(_n, \text{answer}).0 > \text{in}(_n, \text{break}).0)]] \}. \end{aligned}$$

◇

A context may evolve either because of interactions occurring within the context, or because new components dynamically join the context. Formally, the evolution of a context due to internal interactions can be described by means of a transition relation \mapsto over contexts, which is defined in terms of the previous transition relation \rightarrow .

The following rule describes the way in which a context may evolve because of the interaction between two separate patterns:

$$\frac{E \xrightarrow{\text{in}(X,d)} E' \quad F \xrightarrow{\text{out}(X,d')} F' \quad \text{in}(X,d) \sim_\sigma \text{out}(X,d')}{\{(\bar{X})[E], (\bar{Y})[F]\} \cup \mathcal{C} \mapsto \{(\bar{X})[E'\sigma], (\bar{Y})[F']\} \cup \mathcal{C}} \text{ (comm)}$$

Rule (comm) states that a context may evolve because two of its patterns perform two compatible communication actions. Two actions are compatible if they are complementary (one is a $\text{in}()$ and the other is an $\text{out}()$ operation on the same channel), and if the types of data exchanged are compatible. (In this sense our proposal conservatively extends the signature compatibility of standard component interfaces.) The compatibility relation is denoted by \sim_σ where σ is a name assignment which preserves the free names of the context. Notice that, after the communication has taken place, the receiver suitably stores the data received by means of the substitution σ .

A context may also evolve because a single pattern autonomously performs a silent τ action, as formalized by the following rule:

$$\frac{E \xrightarrow{\tau} E'}{\{(\bar{X})[E]\} \cup \mathcal{C} \mapsto \{(\bar{X})[E']\} \cup \mathcal{C}} \text{ (silent)}$$

Example 3 Consider again the context \mathcal{C}_1 introduced in example 2:

$$\begin{aligned} & \{ () [\text{out}(_n, \text{query}). (\text{in}(_n, \text{answer}).0 + \tau.\text{out}(_n, \text{break}).0)]] , \\ & () [\text{in}(_n, \text{query}). (\text{out}(_n, \text{answer}).0 > \text{in}(_n, \text{break}).0)]] \} \end{aligned}$$

Context \mathcal{C}_1 may evolve, by rule (comm), in the new context \mathcal{C}_2 :

$$\begin{aligned} & \{ () [\text{in}(_n, \text{answer}).0 + \tau.\text{out}(_n, \text{break}).0]] , \\ & () [\text{out}(_n, \text{answer}).0 > \text{in}(_n, \text{break}).0]] \} \end{aligned}$$

The new context may evolve in two different ways, because of the possible local choice of the first pattern. If the first pattern (viz., the client) receives the message sent from the server, then the context will evolve into:

$$\{ () [0] , () [0 > \text{in}(_n, \text{break}).0] \}$$

and then into:

$$\{ () [0] , () [0] \}.$$

If instead the client decides to send a break to the server, then context \mathcal{C}_2 will evolve (because of a silent move) into:

$$\{ () [\text{out}(_n, \text{break}).0] , () [\text{out}(_n, \text{answer}).0 > \text{in}(_n, \text{break}).0] \}$$

and then into:

$$\{ () [0] , () [0] \}.$$

◇

3 Correctness of contexts.

As already mentioned in the Introduction, the ultimate objective of our work is to pave the way for the efficient verification of properties in component-interacting systems. When facing the verification of concurrent systems, the absence of possible deadlocks is the typical property one would like to verify.

3.1 Closed systems.

Let us first consider the case of *closed* systems, that is, systems which consist of a fixed set of interacting components (viz., new components cannot join the system after the system is started). In our setting, a closed system corresponds to a *closed context* of the form:

$$\mathcal{C} = \text{join}((\bar{Y}_n)[P_n], \gamma_n, \dots, (\text{join}((\bar{Y}_2)[P_2], \gamma_2, (\text{join}((\bar{Y}_1)[P_1], \gamma_1, \{\}))))$$

where \mathcal{C} has an empty set of open names and it may evolve only by means of (*comm*) and (*silent*) moves.

In order to formally define a notion of *deadlock-free* context, we first introduce the notion of *successful* context. Namely a context is successful if all its patterns have been successfully reduced to the empty behaviour 0.

Definition 5 (Successful context) A context $\mathcal{C} = \{(\bar{X}_1)[P_1], \dots, (\bar{X}_n)[P_n]\}$, is successful if and only if $\forall i \in \{1, \dots, n\} : P_i \equiv 0$.

A closed context is deadlock-free if it is either successful or if all its possible evolutions lead to a successful context.

Definition 6 (Deadlock-free context) A context \mathcal{C} is deadlock-free if and only if either:

- (a) \mathcal{C} is successful, or
- (b) $\forall \mathcal{C}' : \mathcal{C} \mapsto \mathcal{C}'$, \mathcal{C}' is deadlock-free.

Example 4 It is easy to see that context \mathcal{C}_1 of Example 2 is deadlock-free. Indeed \mathcal{C}_1 may evolve in two different ways, and both lead to a successful context.

Consider now a context formed by a client not capable of generating break events, joined with a server capable of handling break events:

$$\{ () [\text{out}(_n, \text{query}). \text{in}(_n, \text{answer}).0] , \\ () [\text{in}(_n, \text{query}). (\text{out}(_n, \text{answer}).0 > \text{in}(_n, \text{break}).0)] \}$$

It is easy to see that also this context is deadlock-free, as it has only one possible evolution leading to a successful context. On the contrary, if we consider the dual situation of a client capable of generating break events, joined with a server not capable of handling break events:

$$\{ () [\text{out}(_n, \text{query}) . (\text{in}(_n, \text{answer}) . 0 + \tau . \text{out}(_n, \text{break}) . 0)] , \\ () [\text{in}(_n, \text{query}) . \text{out}(_n, \text{answer}) . 0] \}$$

we obtain a context which is not deadlock free, since there is an evolution of the context leading to the deadlocked context:

$$\{ () [\text{out}(_n, \text{break}) . 0] , () [\text{out}(_n, \text{answer}) . 0] \}. \quad \diamond$$

Notably, since interaction patterns express only finite fragments of interaction, the deadlock-freedom of a closed context can be verified simply by analysing all its possible evolutions, which form a finite set as stated by the following Property [4], where \mapsto^* denotes the transitive closure of \mapsto .

Property 1 *Let \mathcal{C} be a closed context. The set $\{\mathcal{C}' \mid \mathcal{C} \mapsto^* \mathcal{C}'\}$ is finite.*

3.2 Open systems.

Let us now turn our attention to the case of *open* systems, where new components may dynamically join the system while the latter is running.

Unfortunately, typical notions of correctness employed in closed systems are of limited interest for the case of open systems. For instance determining whether a context \mathcal{C} is deadlock-free is not quite significant since the introduction of a new component in \mathcal{C} may introduce or remove the possibility of a deadlock. On the other hand, during the incremental construction of a system, we may generate a (temporarily) deadlocked context before completing it with another component that will make the context deadlock-free.

As already mentioned in the Introduction, the basic question we aim at addressing is: “*What happens if this component is inserted in the current context ?*”. In describing the evolution of a context, we would like to restrict the access of a new pattern with respect to some condition of correctness. The notion of *feasibility* [4] formalises a condition of (partial) correctness for open systems. Intuitively speaking, a context \mathcal{C} is feasible if there exists some interaction pattern P such that the insertion of P into \mathcal{C} will yield a (closed) deadlock-free context.

Definition 7 (Feasible context) *A context \mathcal{C} , with a set of open names $\bar{X}_{\mathcal{C}}$, is feasible if and only if there exist a pattern $(\bar{Y}_E)[E]$, name disjoint from \mathcal{C} , and a mapping γ from $(\bar{X}_{\mathcal{C}} \cup \bar{Y}_E)$ to a set of fresh names such that the context:*

$$\mathcal{C}' = \text{join}((\bar{Y}_E)[E], \gamma, \mathcal{C})$$

is deadlock-free. The pattern $(\bar{Y}_E)[E]$ is called a completion for the context \mathcal{C} .

Notice that the condition that $\text{dom}(\gamma) = (\bar{X}_{\mathcal{C}} \cup \bar{Y}_E)$ implies that \mathcal{C}' will be a closed context (viz., *join* binds all the open names of both \mathcal{C} and E). The above definition reduces the feasibility of open contexts to the correctness of closed ones. In this sense, the interaction pattern $(\bar{Y}_E)[E]$ gives an indication, informally speaking, of the interactions necessary to the patterns already in the context for completing their coordinated tasks. (Usually, this kind of contribution will be given by more components joining the context at different instants and, possibly, introducing new interactions and new communication channels that do not appear in $(\bar{Y}_E)[E]$.)

Once the notion of feasibility has been introduced, we can formalise the dynamic evolution of an open context by means of a transition relation \Longrightarrow over contexts, which properly extends the previous transition relation \mapsto . The new relation \Longrightarrow is defined by two rules:

$$\frac{\mathcal{C} \mapsto \mathcal{C}'}{\mathcal{C} \Longrightarrow \mathcal{C}'} \text{ (lift)} \qquad \frac{\mathcal{C}' = \text{join}((\bar{Y}_F)[F], \gamma, \mathcal{C}) \quad \mathcal{C}' \text{ is feasible}}{\mathcal{C} \Longrightarrow \mathcal{C}'} \text{ (join)}$$

Rule *(lift)* includes the \mapsto relation in the \Longrightarrow relation. Rule *(join)* describes the feasibility preserving evolution of context. Intuitively speaking, the feasibility of the new context guarantees that the introduction of the new component will not introduce a deadlock possibility. As said, each interaction pattern presents to

the environment some “entry points” (channels) by means of which other interaction patterns will be able to communicate with it. When a pattern joins a context, the name assignment γ (possibly) binds some of the open names of the pattern with some open names of the context. Afterwards, as the context evolves, other channel names can be shared by means of communications, as it happens in the standard π -calculus.

The feasibility of a context can be checked by a nondeterministic generation of the appropriate completion. Indeed, as stated by Proposition 1, the set of the traces of a context is a finite structure. By exploring this structure that contains all the information about the possible behaviours of the context, it is possible to determine the existence of a proper completion.

Notice that Example 4 contains a case of non feasible context. The deadlocked patterns present in the context are both trying to output messages over the name $_n$. There is not complementary communication action in the context, nor it will be after the joining of new components, since the name $_n$ can not be further shared anymore. The context is, and will always be, deadlocked over these patterns.

4 Comparing contexts.

The possibility of efficiently verifying the *feasibility* of a context is quite important during the incremental construction of a complex interacting system. Namely, one can verify whether the introduction of a new component in a partially constructed system (viz., an open context) will introduce or not the possibility of a deadlock in the system. Checking the feasibility of a context is also important for open systems in general. For instance, in the case of distributed applications supporting code mobility, the arrival of new foreign code in a location should not spoil the feasibility of the site itself.

While feasibility is an important invariant, it is definitely not the only property that one would like to verify. When a component joins an existing context, one would also like to be able to check whether the new component “completes” or at least “makes more complete” the existing context. Let us try to illustrate the point with a simple example.

Example 5 Consider the interaction pattern of a client that sends a query on an open channel $_s$, receives the answer on the same channel, and then forwards the answer on a different open channel $_c$:

$$\text{CLIENT} = (_s, _c) [\text{out}(_s, \text{query}) . \text{in}(_s, \text{answer}) . \text{out}(_c, \text{answer}) . 0]$$

Suppose that the CLIENT pattern is inserted in the empty context:

$$\mathcal{C}_1 = \text{join}(\text{CLIENT}, [], \{\}).$$

Consider now a server matching the first part of the interaction pattern of CLIENT:

$$\text{SERVER1} = (_c1) [\text{in}(_c1, \text{query}) . \text{out}(_c1, \text{answer}) . 0]$$

and suppose to insert it in the context \mathcal{C}_1 so as to obtain the new context:

$$\mathcal{C}_2 = \text{join}(\text{SERVER1}, [_n/_s, _n/_c1], \mathcal{C}_1)$$

as illustrated in Figure 1. The introduction of the SERVER1 pattern (and the binding of its channel with the channel of CLIENT) has somehow partly “completed” the context, that is, it has somehow restricted the set of behaviours of the context observable from the external environment. Conversely, the previous introduction of the CLIENT pattern somehow “extended” the possible behaviours of the initially empty context.

Notice also that there are situations in which the introduction of a new patterns neither restricts nor extends the behaviours of a context. Consider for instance another server that besides featuring the same interactions of SERVER1, also sends on a different open channel $_w$ a record of the performed transaction:

$$\text{SERVER2} = (_c2, _w) [\text{in}(_c2, \text{query}) . \text{out}(_c2, \text{answer}) . \text{out}(_w, \text{rec}) . 0]$$

The insertion of SERVER2 in the context \mathcal{C}_1 yields the new context:

$$\mathcal{C}_3 = \text{join}(\text{SERVER2}, [_n/_s, _n/_c2], \mathcal{C}_1)$$

which will offer a quite different behaviour from \mathcal{C}_2 . ◊

As motivated by Example 5, we aim at introducing a partial order relation \prec over contexts. Intuitively speaking, the relation $\mathcal{D} \prec \mathcal{C}$ will state that context \mathcal{D} restricts the set of behaviours of context \mathcal{C} . In the

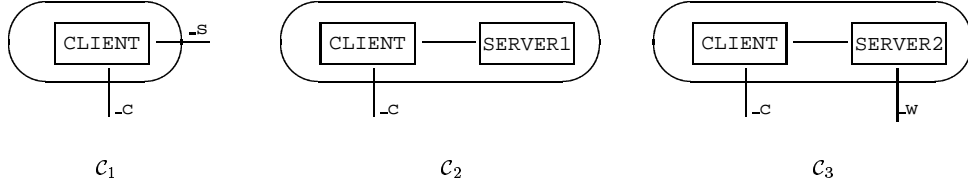


Figure 1: Different contexts for Example 5.

previous example, an automatic verifier would warn the user that the introduction of the CLIENT pattern in the empty context \mathcal{C}_0 will expand the context ($\mathcal{C}_0 \prec \mathcal{C}_1$), that the introduction of the SERVER1 pattern in the context \mathcal{C}_1 will restrict the latter ($\mathcal{C}_2 \prec \mathcal{C}_1$), while the introduction of the SERVER2 pattern in the context \mathcal{C}_1 will neither expand nor restrict \mathcal{C}_1 .

A natural way of defining the \prec ordering on contexts is to refer to the completions of the contexts (introduced in Definition 7). Intuitively speaking, the completion of a context provides a synthetic representation of the external interactions still needed by the context in order to become a closed context. Following this intuition, a context \mathcal{D} restricts another context \mathcal{C} if the completion of \mathcal{D} is a “sub-process” of the completion of \mathcal{C} . The following definition formalizes the above intuition, with the addition of considering that there is in general more than one possible completion for a context.

Definition 8 (\prec relation) A context \mathcal{D} restricts a context \mathcal{C} ($\mathcal{D} \prec \mathcal{C}$) if and only if:

$$\forall d \in COMP(\mathcal{D}) \exists c \in COMP(\mathcal{C}) : d \sqsubseteq c$$

where $COMP()$ denotes the set of completions for a context.

The relation \sqsubseteq over completions defines a partial order on interaction patterns and it is defined as follows.

Definition 9 (\sqsubseteq relation) Given two interaction patterns $d = (\bar{X}_E)[E]$ and $c = (\bar{Y}_F)[F]$, the relation $d \sqsubseteq c$ holds if and only if:

1. $|\bar{X}_E| \leq |\bar{Y}_F|$ (d has less open names of c), and
2. if σ a total mapping from \bar{X}_E to \bar{Y}_F , then the relation $E\sigma \sqsubseteq F$ is defined by the following inference rules:

$$\frac{}{0 \sqsubseteq P} \quad \frac{}{P \sqsubseteq A.P} \quad \frac{}{P \sqsubseteq P}$$

$$\frac{P \sqsubseteq Q}{P \sqsubseteq A.Q} \quad \frac{P \sqsubseteq Q}{A.P \sqsubseteq A.Q} \quad \frac{P \sqsubseteq Q}{P + R \sqsubseteq Q + R} \quad \frac{P \sqsubseteq Q}{P || R \sqsubseteq Q || R} \quad \frac{P \sqsubseteq Q \quad Q \sqsubseteq R}{P \sqsubseteq R}$$

where A is a communication or a τ action, and P, Q, R are behavioural expressions. The rules are up to structural congruence. The rules for $>$ are omitted as $>$ is a derived operator.

Example 6 Consider again example 5. The introduction of SERVER1 in context \mathcal{C}_1 restricts the context, that is: $\mathcal{C}_2 \prec \mathcal{C}_1$. The interaction pattern:

$$p2 = (_z) [\text{in}(_z, \text{answer}).0]$$

is the (minimal) completion for \mathcal{C}_2 . Consider for instance the pattern:

$$p1 = (_ss, _cc) [\text{in}(_ss, \text{query}).\text{out}(_ss, \text{answer}).\text{in}(_cc, \text{answer}).0]$$

which is a completion for \mathcal{C}_1 . It is easy to show that $p2 \sqsubseteq p1$. Indeed $|_z| < |_ss, _cc|$, and let $\sigma(_z) = _cc$. Indeed, since $P \sqsubseteq A.P$, we have that:

$$\text{in}(_cc, a).0 \sqsubseteq \text{out}(_ss, a).\text{in}(_cc, a).0$$

and:

$$\text{out}(_ss, a) . \text{in}(_cc, a) . 0 \sqsubseteq \text{in}(_ss, q) . \text{out}(_ss, a) . \text{in}(_cc, a) . 0$$
 and hence, by transitivity:

$$\text{in}(_cc, a) . 0 \sqsubseteq \text{in}(_ss, q) . \text{out}(_ss, a) . \text{in}(_cc, a) . 0$$
 Similarly, one can verify that $\mathcal{C}_2 \prec \mathcal{C}_3$. ◊

5 Concluding remarks.

We have described a simple interface description language which allows software designers to specify the interaction pattern of a component that will have to interact with other components. As we stated in the Introduction, the ultimate objective of our research is to pave the way for the efficient verification of properties of complex interacting systems.

We have tried to illustrate that the choice of considering simple, non-recursive interaction patterns makes the verification of several interesting properties of concurrent systems tractable. In a way, the introduction of interaction patterns in concurrent systems resembles the introduction of types in conventional programming languages. While type checking cannot in general guarantee the correctness of a program, it does eliminate the vast majority of programming errors [6]. Similarly we argue that while the *feasibility* of a context of interaction patterns does not guarantee the correctness of a concurrent system, it can eliminate many system design errors.

We have also tried to illustrate part of our ongoing work, which is aimed at defining a precise and effective methodology for verifying other important properties of interacting systems. We view the definition of a behavioural ordering on contexts (sketched in Section 4) as the first step in this direction. We are currently analysing an alternative characterization of such an ordering, expressed in terms of process simulation in the tradition of π -calculus. The other direction of our ongoing work is the study of security properties in our setting.

6 References.

- [1] JavaBeans Documentation. <http://java.sun.com/beans/docs/>.
- [2] The Object Management Group. <http://www.omg.org>.
- [3] G. Berry. Preemption in concurrent systems. In *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, Bombay, India, 1993. Springer-Verlag.
- [4] A. Bracciali, A. Brogi, and F. Turini. Coordinating Interaction Patterns. To appear in the *Proceedings of the ACM Symposium on Applied Computing*, Las Vegas, USA, 2001.
- [5] C. Canal, L. Fuentes, J. Troya, and A. Vallecillo. Adding semantic information to IDLs. Is it really practical? In *Proceedings of the OOPSLA'99 Workshop on Behavioral Semantics*, Denver, Colorado, 1999.
- [6] L. Cardelli. Type systems. *Handbook of Computer Science and Engineering*, Chapter 103, CRC Press, 1997.
- [7] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, Redmond, WA, 1996.
- [8] E. Clarke, J. Wing, et al. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.
- [9] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: a semantics-based tool for the verification of concurrent systems. *ACM Trans. Program Lang. Syst.*, 15(1):36–72, Jan. 1993.
- [10] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bull.*, 62:222–259, 1997.
- [11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.
- [12] E. Najm, A. Nimour, and J. Stefani. Infinite types for distributed objects interfaces. In *Proceedings of FMOODS'99*. Kluwer, 1999.
- [13] A. Pnueli. A temporal logic of concurrent programs. *Theor. Comput. Sci.*, 13:45–60, 1981.