# Coordinating Interaction Patterns

Andrea Bracciali
Computer Science Dept.
corso Italia, 40
56100-I Pisa, Italy
braccia@di.unipi.it

Antonio Brogi
Computer Science Dept.
corso Italia, 40
56100-I Pisa, Italy
brogi@di.unipi.it

Franco Turini
Computer Science Dept.
corso Italia, 40
56100-I Pisa, Italy
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 propose 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 a new component joins the context. The main interest of the overall setting is that it supports the efficient verification — both statically and dynamically — of interesting properties of open interacting systems.

## Keywords

Interaction, interface description languages, open systems

## 1. INTRODUCTION

The development of large software systems currently focuses on the issue of *composing* autonomous (existing) components into open distributed architectures. Modularity and composition, which have always been successfully employed in software development, now assume an even broader meaning and importance. Integration is pursued even among different architectures, models and languages.

Many industrial attempts aim at providing the necessary infrastructure technology. The paradigm "write once, run forever" is currently supported by several component-oriented platforms like CORBA [4], DCOM [10], JavaBeans [2], and the recent integrated framework Visual Studio .NET [3]. The reference model generally consists of an *open distributed architecture*, where components behave in a client/server style. The term *open* refers to systems where components can join and leave the computation at run time, without having to recompile or link the whole application.

The main purpose of all these models is integration. Different autonomous objects/components are composed together, often in spite of the fact that they can be written in different languages or can reside in different locations. Many platforms provide abstraction mechanisms to provide a uniform representation of the interesting features of the different components, like interfaces in most of the OO-like architectures. All these proposals are competing on the market to become a standard, and they witness the success of modular decomposition in reducing the inherent complexity of building large software systems.

A major limitation of available component-oriented platforms 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, e.g. like CORBA's IDL [14], they offer a quite low-level support to describe the concurrent behaviour of components. The Concurrency Control CORBAservice [1], for instance, features a *lock* mechanism which is based on different access capabilities (*read*, *write* and *update*) over shared resources. The concurrent constructs of JavaBeans rely on the synchronized methods of Java (together with events and *wait* and *notify* primitives) for serializing concurrent threads during critical sections [2].

These mechanisms do not seem to match those requirements that are widely considered as an index of quality for distributed software. In particular, they seem to force software designers to take into account too many low-level details, and they do not permit a clear separation of coordination from computation concerns, being the coordination "policies generally hard-wired into application" code [13]. Indeed such a separation is understood as the first step towards (automatic) reasoning about the behaviour of the systems and (formally) proving properties about it, [9, 20, 22].

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 $\pi$-calculus [17], and extend interfaces with behavioural descriptions, such as behavioural types [18] or role-based representation of behaviours [7].

On the one hand, these proposals allow one to prove correctness properties, such as absence of deadlocks, as well as to define compatibility relations, such as "the components can properly interact with one another" or "this component can be substituted with that one". The techniques for reasoning on and verifying the resulting systems are typically co-inductive [16], based on the (stepwise) observation of the evolution of potentially non-terminating systems, and rely on the bisimulation relation or on the modal (temporal) logics [21, 12, 11]

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.

The ultimate objective of this paper is to contribute to stretching the applicability of component-oriented programming for the development of large interactive systems. In contrast with the above mentioned proposals, we will adopt a radically different approach to facing the complexity of describing 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. To help intuition, consider a simple client which sends a request to some server on a communication channel (_s say) and then waits to receive an answer from the same channel. We will express the external behaviour of this component by means of the interaction pattern:

```
[ out(_s,query).in(_s,answer).0 ]
```

A more sophisticated client may wish to avoid the risk of indefinitely waiting for an answer, and to be able to choose to send a `break` message to cancel her request. Such a behaviour will be expressed by the interaction pattern:

```
[ out(_s,query).
  ( in(_s,answer).0 + τ.out(_s, break).0 ) ]
```

where the (silent) action $\tau$ is used (along with the choice operator +) to express a local choice of the component (e.g., detecting an internal timeout in this case). The above pattern describes the fragment of interactive behaviour that a component will (possibly repeatedly) exhibit to the external environment.

We will show that the choice of considering simple, non-recursive interaction patterns will make the verification of several interesting properties tractable. In some sense, 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 [8]. Similarly we will argue that while the *compatibility* of a set of interaction patterns does not guarantee the correctness of a concurrent system, it can eliminate many system design errors.

In the rest of the paper we will first propose a simple *interface description language* which allows software designers to easily specify the dynamic behaviour of a component that will have to interact with other components. Syntactically speaking, the proposed language is a subset of $\pi$-calculus [17] with the addition of a (derived) event-handling operator.

We will then introduce the notion of *context* of interacting components. Simply stated, a context is a set of coordinated components that interact with one another concurrently. A context may evolve either because of the interactions occurring among the participating components, or because a new component joins the context. The formal semantics of context evolution will be given by means of standard transition systems.

The main interest of the overall setting is that it supports the efficient verification of open interacting systems, both statically and dynamically. Indeed the correctness of a closed context can be efficiently checked thanks to the finiteness of the interaction patterns of the participating components. The *feasibility* of an open context can also be dynamically checked when a new component wants to join a context.

The rest of the paper is organized as follows. Section 2 introduces *interaction patterns*, as an extension of classical interface description languages. Section 3 presents *contexts*, where components interact accordingly with what stated by their interaction patterns, and the context constructor operator *join*. A formal semantics for our framework will be outlined in Section 4, while a practically verifiable definition of *correctness* for closed contexts is presented in Section 5. Section 6 addresses the problem of the notion of correctness for the case of open systems. It introduces the correspondent notion of *feasibility* and, consequently, extends the semantics of contexts to the case of feasibility-preserving accesses of new components in a running context. Several examples, one of which has already appeared in the literature as a paradigmatic example about the subject, will support the intuition in understanding the formal presentation outlined in these sections. Section 7 contains some concluding remarks.

## 2. INTERACTION PATTERNS

We now introduce the syntax of the interface description language for specifying the dynamic behaviour of components. The interactive behaviour of a component is expressed by means of a *behavioural expression* which is a composition of synchronous communication actions and of silent actions $\tau$.

Two communication operations – $in(x,d)$ and $out(x,d)$ – 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`, `W` or any typed expression. Application data may also contain channel names, like `_x` in `query(W, _x)`[1].

Actions can be composed by means of the standard prefix (.), parallel (||), and nondeterministic choice (+) operators, while 0 denotes the empty behaviour. A behavioural expression can be also prefixed by the silent action $\tau$. Intuitively speaking, a $\tau$ action denotes some internal computation step that the component can perform independently of the external environment, i.e. without interacting with other patterns. As we already mentioned in the Introduction, $\tau$ actions are introduced to explicitly describe the local choices of an interaction pattern.

A derived event-handling operator (>) is also introduced to model the possibility of a component to react to an external event. Intuitively, the expression $E > (in(x,d).F)$ will behave as $E$ unless it receives event $d$ on channel $x$, in which case the expression will react to the event by behaving as $F$. As we will discuss further later on, this event-handling mechanism features a form of "may preemption" [5] as the semantics of a set of communicating processes does not depend on their relative timings, as it is usual in time

[1]When necessary, channel names will be preceded by the underscore to distinguish them from application data.

independent formalisms such as $\pi$-calculus, CCS, or CSP.

Definition 1. *[The language of expressions] The set of* behavioural expressions *is defined as follows:*

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

Syntactically speaking, behaviour expressions are defined by a subset of $\pi$-calculus. Indeed behaviour expressions do not contain recursion as they are intended to specify fragments of the component interaction. Moreover, as we will discuss further in the next section, communication actions are intended to express communications between expressions belonging to different components.

An *interaction pattern* consists of a behavioural expression together with the explicit declaration of its *open names*, that is, the channels that are visible by the external environment. The set of open names of a behavioural expression $E$ will be written as $\bar{X}_E$.

Definition 2. *[Interaction pattern] An* interaction pattern *is a formula of the form $(\bar{X}_E)[E]$, where $E$ is a behavioural expression, and $\bar{X}_E$ is its set of open names. The open names of an expression are a subset of the free names*[2] *of the expression.*

Let us now discuss two simple examples to illustrate the use of interaction patterns for describing the interactive behaviour of components.

Example 1. (Client/server) *Following the Introduction, the interaction pattern of a client can be expressed as:*

```
CLIENT = (_s)
         [ out(_s,query).
         ( in(_s,answer).0 + τ.out(_s, break).0 ) ]
```

*where* _s *is the only open name of the pattern. (As a convention, all channel names in the examples will begin with the underscore.) Similarly the interaction pattern:*

```
SERVER = (_c)
         [ in(_c,query).
         ( out(_c,answer).0 > in(_c, 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 $\tau$ 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 as a means to express recurrent fragments of interactions: Each time a server will interact with a client, it will do it following the same scheme, a sort of behavioural fingerprint.* ◇

Example 2. (The gas station example) *Let us consider the Gas Station example, firstly presented in [15]. A gas station consists of p pumps and m cashiers. Each driver has to pay an amount of money to an available cashier, she receives a receipt and then takes fuel from an available pump. As stated in [13], from where it has been taken, this example illustrates different coordination problems: information transfer among components (e.g., the amount paid by the driver and transferred from the cashier to the pump), multiaction concurrent synchronization (e.g., first pay then take fuel), shared resources access (e.g., the pumps), open system (drivers join in and leave dynamically). The typical interaction pattern of a pump is:*

```
PUMP = (_c)
       [ in(_c, serve(_w,amnt)). out(_w, fuel). 0 ]
```

*Namely, the pump can receive a request from a cashier on channel* _c*. The request must be of the form* serve(_w,amnt)*, where* serve *is the name of the service offered by the component which requires data (*amnt*) and a channel name (*_w*) via which the pump will "communicate" fuel to a driver. Notice that* _c *is the only free and open name of the pattern. The pattern of a cashier is:*

```
CASH = (_d, _p)
       [ in(_d, payment(amnt)).
       (out(_d, rcpt(_p)). 0 ||
        out(_p, serve(_p,amnt)) .0 ) ]
```

*A cashier offers two channels (*_d *and* _p*) to the external environment. The cashier first waits to receive a payment from a driver on channel* _d*. Then it sends back (on the same channel) a receipt to the driver along with the name* _p *of the channel on which she can communicate with the assigned pump. The cashier also sends on channel* _p *a serve request to a pump. Finally, let us consider the interaction pattern of a generic driver:*

```
DRIVER = (_s)
       [ out(_s, payment(amnt)).
         in(_s, rcpt(_u)). in(_u, fuel). 0 ]
```

*The driver sends her payment to a cashier on channel* _s*, and she waits on the same channel for her receipt and for the name of the channel (*_u*) via which to communicate with the assigned pump. She then waits to receive her fuel on channel* _u*.* ◇

## 3. 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, it yields a new context. The name assignment is used to bind together open names of the new pattern and of the patterns already present in the context. As a consequence of the assignment, the names that are bound by the assignment are not open in the new context.

Definition 4. *[Join operation] Let $\mathcal{C}$ be a context $\{(\bar{X}_{P_1})[P_1],\ldots,(\bar{X}_{P_n})[P_n]\}$ and let $\bar{X}_{\mathcal{C}} = (\bar{X}_{P_1} \cup \ldots \cup \bar{X}_{P_n})$*

[2]The free names of an expression $E$ are those names which occur in $E$ not bound by an input operation [17]. For instance the name $y$ is bound by the input operation $in(x,y)$ as well as by $in(x,m(d,y))$ where $m(d,y)$ is a method invocation.

*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 $\gamma$ be a mapping from $\bar{D} \subseteq (\bar{X}_E \cup \bar{X}_{\mathcal{C}})$ to a set of fresh names. The context*

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

*is defined as follows:*

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

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

A context with an empty set of open names is called *closed*, otherwise it is called *open.*

EXAMPLE 3. *Let us now consider again the gas station example to show how contexts can be incrementally constructed. Let us build a context consisting of a cashier, a pump, and a driver, starting from the empty context $\mathcal{C}_0 = \{\}$. If we insert the cashier pattern in the empty context, we obtain:*

$$\mathcal{C}_1 = join(\texttt{CASH}, [], \mathcal{C}_0)$$

*where $\mathcal{C}_1$ is:*

```
{
/* CASH */
(_d, _p) [ in(_d, payment(amnt)).
          (out(_d, rcpt(_p)). 0 ||
           out(_p, serve(_p,amnt)). 0) ]
}
```

*Let us now insert the pump pattern in the resulting context, by binding the open names* _p *(of the cashier pattern) and* _c *(of the pump pattern) to the same name* _c1p1*:*

$$\mathcal{C}_2 = join(\texttt{PUMP}, [\texttt{\_c1p1/\_p}, \texttt{\_c1p1/\_c}], \mathcal{C}_1)$$

*where $\mathcal{C}_2$ is:*

```
{
/* CASH */
(_d) [ in(_d, payment(amnt)).
     (out(_d, rcpt(_c1p1)). 0 ||
      out(_c1p1, serve(_c1p1, amnt)). 0) ],
/* PUMP */
()   [in(_c1p1, serve(_w, amnt)). out(_w, fuel). 0]
}
```

*Let us finally insert the driver pattern into the new context $\mathcal{C}_2$, this time binding the open names* _s *(of the driver) and* _d *(of the cashier) to the same name* _d1c1*:*

$$\mathcal{C}_3 = join(\texttt{DRIVER}, [\texttt{\_d1c1/\_s}, \texttt{\_d1c1/\_d}], \mathcal{C}_2)$$

*where $\mathcal{C}_3$ is the following closed context:*

```
{
/* CASH */
() [ in(_d1c1, payment(amnt)).
    (out(_d1c1, rcpt(_c1p1)). 0 ||
    out(_c1p1, serve(_c1p1, amnt)). 0) ],
/* PUMP */
() [ in(_c1p1, serve(_w, amnt)). out(_w, fuel). 0],
/* DRIVER */
() [ out(_d1c1, payment(amount)).
    in(_d1c1, recp(_u)). in(_u, fuel). 0 ]
}
```

◇

## 4. SEMANTICS

Once a context has been constructed, it may evolve because of the interactions between the participating patterns. The way in which such an interaction takes place can be formally described by means of two transition systems.

We first model the intensional behaviour of an interaction pattern, independently of the context in which it will operate. Intuitively, the intensional behaviour describes the communications that a pattern may perform. Such a behaviour can be naturally expressed by means of a transition system $\rightarrow$, defined by a set of inference rules. Most of the rules correspond to the classical rules for $\pi-$calculus, the main difference being the absence of communications at this level of abstraction, since communication is not allowed between (parallel) processes of the same interaction pattern. The relation $\rightarrow$ is defined up to structural congruence for the operators + and || as usually defined [17].

$$\frac{}{\tau\,.\,E \overset{\tau}{\rightarrow} E}\ (\tau) \qquad \frac{}{A\,.\,E \overset{A}{\rightarrow} E}\ (act)$$

$$\frac{E \overset{A}{\rightarrow} E'}{E \,||\, F \overset{A}{\rightarrow} E' \,||\, F}\ (||) \qquad \frac{E \overset{A}{\rightarrow} E'}{E + F \overset{A}{\rightarrow} E'}\ (+)$$

$$\frac{E \overset{A}{\rightarrow} E'}{E > (in(X,d).F) \overset{A}{\rightarrow} E' > (in(X,d).F)}\ (>_1)$$

$$\frac{F \overset{A}{\rightarrow} F'}{E > F \overset{A}{\rightarrow} F'}\ (>_2) \qquad \frac{}{0 > F \overset{\tau}{\rightarrow} 0}\ (>_3)$$

Rules $(\tau)$, $(act)$, $(||)$, $(+)$ are the standard rules for the prefix, parallel and the choice composition. The need for keeping two distinct rules $(\tau)$ and $(act)$ will be clear when we will model the evolution of an entire context. Rules $(>_1)$, $(>_2)$, and $(>_3)$ describe the $>$ operator. Rule $(>_1)$ states that the expression $E > (in(X,d).F)$ may behave like $E$ and evolve to $E' > (in(X,d).F)$ 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 $F$. Rule $(>_3)$ states that an event-handling expression that has terminated its non-reactive part can autonomously reduce to 0. The above rules describe the possible behaviours of an interaction pattern independently of the context in which it will operate. We now introduce a second transition system $\Longrightarrow$ to describe the evolution of contexts. As one may expect, the transition system $\Longrightarrow$ is defined in terms of the previous transition system $\rightarrow$ and it models the way in which separate patterns interact.

More precisely, a context may evolve either because two separate patterns synchronize or because a single pattern autonomously performs a silent $\tau$ action. The first situation is described by the following rule:

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

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 an $in()$ and the other is an $out()$ 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 $\sim_\sigma$ where $\sigma$ 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 $\sigma$.

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

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

Rules (*comm*) and (*silent*) define the transition relation $\Longrightarrow$ which models all possible evolutions of a context. Some remarks are worth making here:

1. The patterns in a context can exchange data, method invocations as well as channel names. Initially, each pattern knows the open names of the other patterns which have been bound to (some of) its open names. Then, as the context evolves, other channel names can be shared as in standard $\pi$-calculus.

2. In contrast with standard $\pi$-calculus, communication between parallel terms of the same pattern is not allowed. The only type of communication allowed in a context is between expressions belonging to separate patterns.

3. Global and local choices are explicitly distinguished. For instance, a server willing to offer two services ($s1$ and $s2$ say) can be described by a pattern of the form:
   `(_x) [ in(_x,s1).P + in(_x,s2).Q ],`
   where the choice of which service has to be actually offered is *global* as it depends on the interaction with other patterns in the context. On the other hand, in the client example:
   `(_s) [ out(_s,query).`
   `( in(_s,answer) + τ.out(_s,break) ) ]`
   the choice of sending a `break` is *local* as it depends on an internal action of the client.

4. It is worth observing that the event-handling operator $>$ is a derived operator, 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 $(A||B) > I$ is a convenient shorthand for $(A.(B+I)) + (B.(A+I)) + I$
   The $>$ operator features a "may preemption" mechanism, since formalisms based on time independence (like CCS or $\pi-$calculus) cannot express *instantaneous* reactions to external events [5]. The $>$ resembles the LOTOS *disable* operator [6], as an expression $E > I$ "specifies the non-deterministic set of behaviours that could be observed in an instantly reactive system according to the relative timings of the communications in $E$ and in $I$, without having to specify these timings" [5]. For instance, the context:
   `{ ()[in(_x, d)], ()[out(_x, d) > in(_y, e)],`
   `()[out(_y, e)] }`
   may nondeterministically evolve either into
   `{ ()[], ()[], ()[out(_y, e)] }`
   or into
   `{ ()[in(_x, d)], ()[], ()[] } .`

EXAMPLE 4. *Consider again the* `CLIENT` *and* `SERVER` *patterns of example 1, and consider the context:*

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

*that is:*

```
{ () [ out(_n,query).
       ( in(_n,answer).0 + τ.out(_n,break).0 ) ] ,
  () [ in(_n,query).
       ( out(_n,answer).0 > in(_n, break).0 ) ] }
```

*According to the definition of the $\Longrightarrow$ relation, the above context may first evolve into the new context $\mathcal{C}_1$:*

```
{ () [ in(_n,answer).0 + τ.out(_n,break).0 ] ,
  () [ out(_n,answer).0 > in(_n, break).0 ] }
```

*Then the 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 > in(_n, break).0 ] }`
*and finally into:* `{ () [ 0 ] , () [ 0 ] }`
*If instead the client decides to send a break to the server, then context $\mathcal{C}_1$ will evolve (because of a* silent *move) into:*

```
{  () [out(_n,break).0] ,
   () [out(_n,answer).0 > in(_n,break).0] }
```

*and then into:* `{ () [ 0 ] , () [ 0 ] } .` ◇

EXAMPLE 5. *Consider again the gas station example. The context $\mathcal{C}_3$ (as defined at the end of example 3) may evolve because of the communications taking place among the participating patterns. The* `CASH` *pattern can indeed perform all its communication actions and, after the corresponding three applications of the rule (comm), context $\mathcal{C}_3$ evolves into the new context $\mathcal{C}_4$:*

```
{ /* CASH */
  () [],
  /* PUMP */
  () [out(_c1p1, fuel). 0],
  /* DRIVER */
  () [in(_c1p1, fuel). 0]     }
```

*It is worth observing how, in the style of $\pi-$calculus, channel names have been communicated among different patterns. (viz., the name* `_c1p1` *used to link the driver to the assigned pump). More generally, such a mechanism can be used to model an* object request broker *in a distributed component-based system. We can also observe that in context $\mathcal{C}_4$ the cashier pattern successfully accomplished its task, while the other two patterns are ready to complete their tasks too.* ◇

## 5. VERIFICATION OF CORRECTNESS

A crucial issue in the development of a complex software system is the verification of the correctness of the system. In our setting this corresponds to verifying whether a constructed context is correct or not.

In order to formally define the notion of *correct 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] Given a context* $\mathcal{C} = \{(\bar{X}_1)[P_1], \ldots, (\bar{X}_n)[P_n]\}$, *it is* successful *if and only if* $\forall i \in [1, \ldots, n] : \; P_i \equiv 0$.

We now define the notion of correctness for a (closed) context.

Definition 6. *[Totally correct context] Given a closed context* $\mathcal{C}$, *it is* totally correct *if and only if either:*

*(a)* $\mathcal{C}$ *is successful or*

*(b)* $\forall \mathcal{C}' : \; \mathcal{C} \Longrightarrow \mathcal{C}', \; \mathcal{C}'$ *is* totally correct.

Intuitively speaking, a context is totally correct if it is successfull, or if all its possible evolutions lead to a successfull context.

Example 6. *It is worth reconsidering the client/server example introduced in example 1. The example describes the interactive behaviour of a client which sends a request to a server and waits for a reply, and a server which waits for a request from a client and sends an answer. Moreover the client may choose to send a break to the server and the server has the capability of reacting to such a break event. It is easy to see (cfr. example 4) that the context obtained by the first inserting one server, and then inserting one client, is totally correct. Indeed such a context may evolve in two different ways, both leading to a successfull context.*

*It is worth considering the situation in which a client not capable of generating break events is put together with a server capable of handling break events. The resulting context will be:*

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

*which is still a totally correct context. (Indeed the context may perform only one sequence of steps and reach a successfull context.)*

*On the other hand, let us consider the dual situation in which a client capable of generating break events is put together with a server not capable of handling break events. The resulting context will be:*

```
{ () [ out(_n,query).
       ( in(_n,answer).0 + τ.out(_n,break).0 ) ] ,
  () [ in(_n,query).  out(_n,answer).0 ) ] }
```

*Following the intuition, the above context is* not *totally correct since there is a derivation leading to the non-successfull context:*

```
() [out(_n,break).0], () [out(_n,answer).0]}        ◇
```

As already stated in the Introduction, the main interest of our proposal is the possibility of *efficiently* verifying properties of interactive systems. Indeed the fact that interaction patterns (and hence contexts) express only finite fragments of interaction supports the efficient analysis of contexts. The total correctnes of a context can be verified by simply running the context itself and by analysing all its possible evolutions, thanks to the following property.

Proposition 1. *Let* $\mathcal{C}$ *be a context. The set* $\{\mathcal{C}' | \mathcal{C} \Longrightarrow^* \mathcal{C}'\}$ *is finite.*[3]

[3]Notice that the absence of recursion in the patterns plays a fundamental role here. Notice also that structural congruence, as usually defined, respects the structural complexity w.r.t. the number of actions of each expression.

By Proposition 1, the verification of the correctness of a closed context amounts to exploring a finite set of states.

## 6. DYNAMIC CONTEXTS

In the previous sections we have introduced the notions of interaction pattern and context in order to describe and reason on interacting systems. The operational semantics given in Section 4 formally describes all possible evolutions of a closed context composed of a statically fixed number of patterns.

In fact, in open systems, components may dynamically join the system. In other words, a context may evolve either because of the interactions between the participating patterns, or because a new pattern joins the system. Intuitively speaking, this would corresponding to extending the transition relation $\Longrightarrow$ with a rule of the form:

$$\frac{\mathcal{C}' \; = \; join((\bar{Y})[E], \; \gamma, \; \mathcal{C})}{\mathcal{C} \Longrightarrow \mathcal{C}'}$$

The verification of the correctness of a system is obviously a crucial question also in the case of open systems. On the other hand, when considering open systems, the notion of correctness has to be suitably refined to cope with the incompleteness of dynamically evolving contexts.

We hence introduce a weaker notion of correctness, and we call it *feasibility.* Intuitively speaking, an open context is feasible if there exists an interaction pattern whose insertion in the context makes it closed and totally correct.

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* $\gamma$ *from* $(\bar{X}_{\mathcal{C}} \cup \bar{Y}_E)$ *to a set of fresh names such that:*

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

*is a* totally correct *context.*

Notice that the condition that $dom(\gamma) = (\bar{X}_{\mathcal{C}} \cup \bar{Y}_E)$ implies that $\mathcal{C}'$ is a closed context: All the open names have been bound. The above definition reduces the feasibility of open contexts to the correctness of closed ones. In this sense, the interaction pattern $(\bar{Y}_E)[E]$ represents, informally speaking, all the interaction 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 in the context at different instants and, possibly, introducing new interactions and new communication channels that do not appear in $(\bar{Y}_E)[E]$.

Having introduced the definition of feasibility for open context, it is now possible to extend the semantics of the context with a feasibility preserving access operation for a component in a context.

Definition 8. *[*in*-rule] Let* $\mathcal{C}$ *be a feasible context with a set of open names* $\bar{X}_{\mathcal{C}}$, *let* $(\bar{Y}_E)[E]$ *be an interaction pattern, and let* $\gamma$ *be a mapping from* $\bar{D} \subseteq (\bar{X}_{\mathcal{C}} \cup \bar{Y}_E)$ *to a set of fresh names. The following rule* $(in)$ *extends the semantics of the contexts:*

$$\frac{\mathcal{C}' \; = \; join((\bar{Y})[E], \gamma, \mathcal{C}) \quad \mathcal{C}' \text{is feasible}}{\mathcal{C} \Longrightarrow \mathcal{C}'} \; (in)$$

Note that by this rule the Proposition 1 does not hold anymore: the correctness of closed systems can be investigated by means of a finite amount of information, while in open systems, by means of a finite process, we can only check for feasibility. The feasibility of a (open) context can be checked by a nondeterministic generation of the appropriate complementary interaction pattern.

## 7. CONCLUDING REMARKS

Our proposal relates to *coordination* and *composition*. We investigated the possibility of including in the interface of a generic component a description of its interactive behaviour. Hence, we have defined a formal framework suitable for describing systems composed of such components and verifying the correctness of the construction.

The main novelty of our proposal consists, at the best of our knowledge, in the original approach to the representation of interaction via finite fragments of behaviour, i.e. interaction patterns. Thanks to this perspective, we have been able to use a subset of the $\pi-$calculus, exploiting many of its nice features as a tool for concurrency, without paying the price of the high computational costs of many of its applications. On the other hand, we gained in ease of practical usage by having introduced a (derived) event-handling operator.

The developed framework applies uniformly to both closed and open systems, the latter being characterized by the dynamical participation of the components and hence by an incomplete information about their state and correctness. An interesting notion of correctness for open systems as possibility of reaching a successful state has been defined. An effective verification of such a property reduces to exploring a finite structure, namely the patterns in the context.

Future work will be devoted to extend the framework. Exploiting further the coordination features of the model, along the line of other similar approaches, e.g. [19], it is easy to imagine to model other situations, besides *join*, by new operations, like, e.g. *leave*, *test*, ..., in analogy with the tuple space operators *out*, *read* ..., with interaction patterns as tuples. The relationships between *behavioural interfaces*, in terms of their *compatibility*, need further studies, which might hopefully result in the definition of a hierarchy and an associate notion of *subtyping*.

Applications to model specific architectures seem worthwhile of further studies, too. As said, contexts and their dynamic access can be used to model mobile code for distributed applications as well as (visual) component-based development tools. For instance the JavaBeans model of events, properties and methods, could be extended with behaviours providing the ground to prove behavioural correctness, as well as the expressiveness of broker-based frameworks, like CORBA, could be enhanced by behavioural descriptions.

## 8. ACKNOWLEDGMENTS

This work was partly supported by the M.U.R.S.T. project "Theory of Concurrency, Higher Order and Types (TOSCA)".

## 9. REFERENCES

[1] Concurrency Control CORBAservice. `http://www.omg.org/techonology/documents/formal/concurrency_service.htm`.

[2] JavaBeans Documentation. `http://java.sun.com/beans/docs/`.

[3] .NET Programming the Web. `http://msdn.microsoft.com`.

[4] The Object Management Group. `http://www.omg.org`.

[5] G. Berry. Preemption in concurrent systems. In *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, Bombay, India, 1993. Springer-Verlag.

[6] E. Brinksma. A Formal Description Tecnique based on the Tempotal Ordering of Observable Behaviours. *Information Processing Systems - Open Systems Interconnection*, 1988. ISO8807.

[7] 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.

[8] L. Cardelli. Type systems. *Handbook of Computer Science and Engineering*, Chapter 103, CRC Press, 1997.

[9] N. Carriero and D. Gelernter. Coordination languages and their significance. *CACM*, 35(2):97–107, 1992.

[10] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, Redmond, WA, 1996.

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

[12] 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.

[13] J. Cruz and S. Ducasse. A group based approach for coordinating active objects. In *Coordination Languages and Models - COORDINATION'99*, volume 1594 of *LNCS*. Springer-Verlag, 1999.

[14] M. Dolgicer. Inside CORBA services. *Application Development Trends*, pages 63–71, June 1997.

[15] D. Helmbold and D. Luckman. Debugging ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[16] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bull.*, 62:222–259, 1997.

[17] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.

[18] E. Najm, A. Nimour, and J. Stefani. Infinite types for distributed objects interfaces. In *Proceedings of FMOODS'99*. Kluwer, 1999.

[19] A. Omicini and F. Zambonelli. Coordination for Internet application development. *Journal of Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999.

[20] G. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46, 1998.

[21] A. Pnueli. A temporal logic of concurrent programs. *Theor. Comput. Sci.*, 13:45–60, 1981.

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