IADIS International Journal on Computer Science and Information Systems
Vol. 3, No. 1, pp. 71-85
ISSN: 1646-3692

COMPOSITIONAL ABSTRACTIONS FOR PROCESS NETWORKS

Maciej Koutny*, **Giuseppe Pappalardo**** and **Marta Pietkiewicz-Koutny***
(*) School of Computing Science, Newcastle University, Newcastle upon Tyne NE1 7RU, U.K.
{maciej.koutny, marta.koutny}@ncl.ac.uk

(**) Dipartimento di Matematica e Informatica, Università di Catania, I-95125 Catania, Italy
pappalardo@dmi.unict.it

ABSTRACT

A promising way of dealing with complex behaviours of networks of communicating processes is to use abstractions. In our previous work, interface abstraction, modelled through a suitable relation, allowed us to `interpret' the behaviour of an implementation process as that of a specification process, even in the event that their interfaces differ. The proposed relation is compositional, in the sense that a composition of communicating sub-systems may be implemented by connecting their respective implementations. In our previous work, abstraction has been shown to distribute only over network composition, which restricts its usefulness for compositional correctness analysis. In this paper we extend our treatment to other process constructs, known to be useful in the development of complex distributed applications.

KEYWORDS

Behaviour abstraction, communicating sequential processes, compositionality, algebra of abstractions.

1. INTRODUCTION

The basic issue we aim at addressing in this work is the notion of implementation, and its relationship to abstraction, in the framework of communicating sequential processes. In general, we say that a process *Q implements* a process *P* when its behaviour, *suitably interpreted through an appropriate form of abstraction*, is a potential behaviour of *P*, i.e., when the interpretation of *Q* is more deterministic than *P*. In the following, we refer to *Q* as the *implementation* (process), and to *P* as the corresponding *specification* or *target* or *base* (process).

Development or *refinement* consists in replacing a target *P* with a (usually) more complex or detailed process *Q* representing a proper implementation of *P* with respect to the intended abstraction. This may possibly be accomplished in a stepwise fashion. Every refinement step should undergo *verification*, to be formally proved correct. A refinement step may be an instance of a known pattern (as in *program transformation* [3]), avoiding the need of verification through a specific, dedicated proof. Moreover, a correctness proof may also be an instance of application of a general theorem and/or methodology, as in compositional verification.

Conventionally, in process algebras, such as [12, 14], the notion of implementation differs from the approach advocated here, in that it does not employ abstraction to interpret the behaviour of the implementation process Q as that of the target process P. The behaviour of a correct Q must indeed simply *be* (part of) that of P. Of course, if we shift our attention from process behaviour (i.e., *meaning*, a semantic notion) to *expressions* of the process algebra at hand (a syntactic notion), we can still see (a different kind of) abstraction operating. Then, process *expression* q may be seen as an implementation of a syntactically different expression p, built out of different parts and operators, if the behaviour denoted by q, say Q, is part of that denoted by p, say P. Thus, "abstraction" here means abstraction from syntactic structure, and is applied to the implementation and the target *process expressions*, i.e. q and p respectively. Though fundamental, this abstraction is irrelevant to our notion of implementation, which deals with "pure" processes, Q and P. In it, a process *is* its behaviour. At the syntactic level, conventional refinement of course allows the designer to change the control structure of the target into the desired implementation. However, as implied by the previous discussion, their *interfaces* must coincide.

On the other hand, in refinement, it is often natural to implement abstract, high-level interaction at an *interface* at a lower level of detail and in a more concrete manner. For example, an ideal channel in the target system may in practice need to be replaced by a data/acknowledgement unreliable channel pair in the implementation. To such issues, our abstraction-oriented approach does provide a solution, based upon an *abstraction implementation relation*, which we deem in principle somehow more flexible than the *action-refinement* approach (cf. [4, 9, 13]).

In our previous work (see, e.g., [4, 5, 10]), we proposed abstraction-based implementation relations satisfying *realisability*, a property ensuring that an implementation may be put to good use, and *compositionality*, which requires the implementation relation to distribute over system composition. Thus, a specification composed of a number of connected systems may be implemented by connecting their respective implementations. Compositionality is important to avoid the state explosion problem when carrying out automated verification.

Previously, we only dealt with compositionality/distributivity over the parallel composition operator, a key tool in the construction of concurrent and distributed systems. However, for practical applications, other process combinators are known to be useful. Therefore, in this paper, we aim at extending proper distributivity results to the relevant operators.

Another limitation of our previous results lies in the assumption of point-to-point interprocess communication, over typed input and output channels. We here remove this restriction as well, and move on to deal with arbitrary patterns of communication, e.g., broadcast.

The paper is organised as follows. In the next section, we briefly recall some basic notions used throughout the paper, and discuss an example of behavioural abstraction. The following section outlines our previous approach and results, and the last section presents the proposed extension. The interested reader can find in [4, 9] detailed motivations for an abstraction of interprocess communication, illustrative examples, and further comments on related work, especially [1, 2, 6, 7, 11, 13, 15].

This paper is a revised and extended version of the paper presented at the PITA'07 conference.

2. PRELIMINARIES

We use the CSP process algebra [8, 14]. A CSP *process* P can be regarded as a black box which may engage in interaction with its environment. Atomic instances of this interaction are called *actions* and must be elements of P's finite *alphabet*, αP. P's *traces*, or τP, are the finite sequences of actions that P can engage in. After a given trace t, process P may *refuse* to engage in a set of actions R, which is a convenient device to model deadlock-like situations as well as non-determinism. All such pairs (t, R) form the set of P's *failures*, φP. Finally, some of the traces may lead to un-productive internal loops, forming the set of P's *divergences*, δP. The following notations are similar to that of [8] (below t, u are traces and A a set of actions):

- $t = <a_1, ..., a_n>$ is the trace whose i-th element is action a_i.
- The empty trace is denoted by $<>$, and $\circ$ is the concatenation operation for traces.
- A^* is the set of all traces of actions from A, including the empty trace.
- $t \leq u$ means that trace t is a prefix of u, and $t<u$ that t is a proper prefix of u.
- A mapping h from traces to traces is monotonic if $t \leq u$ implies $h(t) \leq h(u)$, and strict if $h(<>) = <>$.
- If t is a prefix of u then $u\text{-}t$ is the suffix of u after deleting the initial t.
- The trace $t\lceil A$ is obtained by deleting from t all the actions that do not belong to A.

To improve readability, we will sometimes use structured actions of the form $b{:}v$, where v is a *message* and b is a communication *channel*. We will then also talk about the set of channels of a process rather than the set of its actions, and partition the channels of a process P into the input channels, belongs *inP* (depicted by incoming arrows), and output channels, *outP* (depicted by outgoing arrows).

2.1 Failures and divergences

In the standard failures-divergences model of CSP [8, 14] a process P is a triple $(\alpha P, \varphi P, \delta P)$ where αP (the *alphabet*) is a non-empty finite set of actions, φP (the *failures*) is a subset of $\alpha P^* \times \mathbf{P}(\alpha P)$, and δP (the *divergences*) is a subset of αP^*. For a valid process P, its components should satisfy the conditions given below, where τP denotes the set of *traces* of P, defined by $\tau P = \{t \mid (t, R) \in \varphi P\}$:

- τP is non-empty and prefix-closed.
- If $(t, R) \in \varphi P$ and $S \subseteq R$ then $(t, S) \in \varphi P$.
- If $(t, R) \in \varphi P$ and $a \in \alpha P$ and $t \circ <a>$ does not belong to τP, then $(t, R \cup \{a\}) \in \varphi P$.
- If $t \in \delta P$ then $(t \circ u, R) \in \varphi P$, for all $u \in \alpha P^*$ and $R \subseteq \alpha P$.

If $(t, R) \in \varphi P$ then P is said to *refuse* R after performing trace t. Intuitively, this means that P can deadlock, should the environment offer R as the set of possible actions to be executed after t.

If $t \in \delta P$ then P is said to *diverge* after t. In the CSP philosophy this means the process behaves in a totally uncontrollable way.

2.2 Standard CSP process operators

Parallel composition $P||Q$ models synchronous communication between processes in such a way that each of them is free to engage independently in any action that is not in the other's alphabet, but they have to engage simultaneously in any action that is in the intersection of their alphabets. Formally,

- $\alpha(P||Q) = \alpha P \cup \alpha Q$.
- $\delta(P||Q)$ comprises all traces $t \circ u$ such that $(t \lceil \alpha P, t \lceil \alpha Q) \in (\tau P \times \delta Q) \cup (\delta P \times \tau Q)$, and $u \in (\alpha P \cup \alpha Q)^*$.
- $\varphi(P||Q)$ comprises all failures $(t, R \cup S)$ such that $(t \lceil \alpha P, R) \in \varphi P$ and $(t \lceil \alpha Q, S) \in \varphi Q$, as well as all the elements of $\delta(P||Q) \times \mathbf{P}(\alpha(P||Q))$. (Note that $\mathbf{P}(X)$ denotes the powerset of a set X.)

Parallel composition is commutative and associative.

Let P be a process and A be a set of actions of P. Then $P \backslash A$ is a process that behaves like P with the actions in A made invisible. Hiding is here the only operator which may introduce divergence. This happens whenever P can execute an infinite sequence of hidden actions. Formally,

- $\alpha(P \backslash A) = \alpha P - A$.
- $\delta(P \backslash A)$ comprises all traces $t \lceil \alpha(P \backslash A) \circ u$ such that $u \in \alpha(P \backslash A)^*$, and $t \in \delta P$ or there exist $a_1, a_2, \ldots$ in A such that for all n, $t \circ <a_1, \ldots, a_n> \in \tau P$.
- $\varphi(P \backslash A)$ comprises all failures $(t \lceil \alpha(P \backslash A), R)$ such that $(t, R \cup A) \in \varphi P$, as well as all the elements of $\delta(P \backslash A) \times \mathbf{P}(\alpha(P \backslash A))$.

Hiding is associative in that $(P \backslash A) \backslash A' = P \backslash (A \cup A')$.

In the next two operations on processes it is assumed that P and Q have the same alphabets. The deterministic choice and non-deterministic choice offer an alternative between the behaviours of P and Q. In the latter case, no external process has control about which of these options is actually followed (this property is useful, e.g., to model execution errors). Then the external choice ($P \;\square\; Q$) and internal choice ($P \sqcap Q$) are processes with the same alphabet as P and Q, the divergences being the union of those of P and Q, and the failures given respectively by:

- $\varphi(P \square Q)$ comprises all failures $(<>, R) \in \varphi P \cap \varphi Q$, and all failures $(t, R) \in \varphi P \cup \varphi Q$ such that t is non-empty.
- $\varphi(P \sqcap Q)$ is the union of φP and φQ.

The last operator we introduce is prefixing. Assuming that a is an action in the alphabet of P, $a \rightarrow P$ is the process with the same alphabet as P and

- $\delta(a \rightarrow P) = \{<a> \circ t \mid t \in \delta P\}$.
- $\varphi(a \rightarrow P)$ comprises all failures $(<a> \circ t, R)$ such that $(t, R) \in \varphi P$, as well as all the elements of $\{<>\} \times \mathbf{P}(\alpha P - \{a\})$.

There are also atomic CSP processes like $stop_A$ which denotes a terminated process with the alphabet A. One can also use recursive process definitions, e.g., $P = (a \rightarrow P) \,\square\, (b \rightarrow stop)$ defines a process which can execute action a any number of times, and then perhaps execute b and terminate (often *stop*'s subscript alphabet is implicit).

2.3 Behaviour abstraction

Consider two base (or specification) processes, *Gen* and *Buf*, as shown in Figure 1. *Gen* generates an infinite sequence 010101... of messages, or an infinite sequence 101010... of messages on its output channel *d*, responding to the initial message (0 or 1) received on its input channel, *c*, at the beginning of its execution. Formally, the process executes actions in its alphabet $\alpha Gen = \{c{:}0, c{:}1, d{:}0, d{:}1\}$. *Buf* is a buffer processes of capacity one, forwarding messages received on its input channel, *d*. In terms of CSP, we have:

- $Gen = (c{:}0 \rightarrow Gen_0) \,\square\, (c{:}1 \rightarrow Gen_1)$
- $Gen_i = d{:}i \rightarrow d{:}(1\text{-}i) \rightarrow Gen_i$
- $Buf = (d{:}0 \rightarrow e{:}0 \rightarrow Buf) \,\square\, (d{:}1 \rightarrow e{:}1 \rightarrow Buf)$

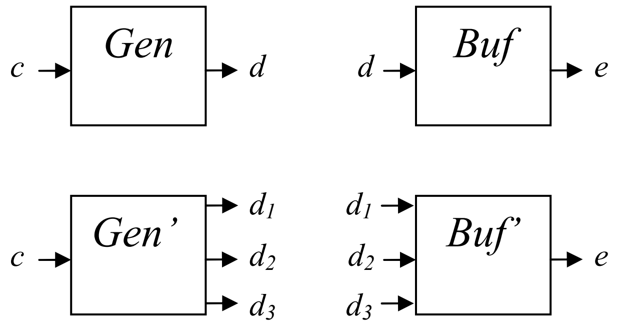

Figure 1. Two base processes (top); and their implementations (bottom)

Suppose that communication between the two processes at the shared channel *d* has been implemented using three channels, d_1, d_2 and d_3, as shown in Figure 1.

The original transmissions on *d* become triplicate and the different copies are sent on the channels d_i ($i = 1, 2, 3$). That is, *Gen'* sends three copies of a message, while *Buf'* forwards the first copy of the message received, ignoring the other two. This simple replication scheme clearly works, as it can be shown that:

- $(Gen \| Buf) \backslash \{d{:}0, d{:}1\} = (Gen' \| Buf') \backslash D$

where $D = \bigcup_{i=1,2,3}\{d_i{:}0, d_i{:}1\}$. Suppose now that the transmission of messages is imperfect and two types of faulty behaviour can occur:

- $Buf'' = Buf' \sqcap stop$ and $Buf''' = Buf' \sqcap Buf'_{23}$

where Buf'_{23} is *Buf'* with all the communication on channel d_1 being blocked. In other words, *Buf''* can break down completely, as *stop*, refusing to input any message; while *Buf'''* can only fail in such a way that although channel d_1 is dead, the other two can still be used to accept messages. (One could think about this scenario as modelling the situation where, in order to improve performance, a “slow” channel d is replaced by three channels: a high-speed yet unreliable channel d_1, and two slow but reliable backup channels d_2 and d_3.). Since it can be shown that:

- $(Gen\|Buf)\backslash\{d{:}0, d{:}1\}$ is *different* from $(Gen'\|Buf'')\backslash D$
- $(Gen\|Buf)\backslash\{d{:}0, d{:}1\}$ is *the same* as $(Gen'\|Buf''')\backslash D$,

it follows that *Buf'''* is much “better” an implementation of the *Buf* process than *Buf''*.

We now observe that the communication between the processes *Gen'* and *Buf'''* can be thought of as adhering to the following two rules:

- [R1] The transmission sequences over d_1, d_2 and d_3 are consistent w.r.t. message content and can thus be directly related to transmissions over d. (This follows from the way *Gen'* produces its output.) The set of all traces over D satisfying such a property will be denoted by *dom*.
- [R2] Transmission over d_2 and d_3 is reliable, but there is no such guarantee for d_1.

On the other hand, communication between the processes *Gen'* and *Buf''* satisfies the first rule, but fails to satisfy the second one. To express this difference formally, we need to render R1 and R2 in some sort of precise notation.

To capture the relationship between traces of the implementations *Buf'*, *Buf''* or *Buf'''* and the relevant target process *Buf* on the corresponding channels, one may employ an (extraction) mapping *extr* which, given a trace in *dom*, over $D=\{d_1{:}0, d_1{:}1, d_2{:}0, d_2{:}1, d_3{:}0, d_3{:}1\}$, returns its interpretation as a trace over $\{d{:}0, d{:}1\}$. For example, interpreting so as to abstract from replication yields

- $\langle\rangle \rightarrow \langle\rangle$
- $\langle d_1{:}0\rangle \rightarrow \langle d{:}0\rangle$
- $\langle d_3{:}0\rangle \rightarrow \langle d{:}0\rangle$
- $\langle d_2{:}1, d_3{:}1\} \rightarrow \langle d{:}1\rangle$
- $\langle d_3{:}1, d_1{:}1, d_1{:}0\rangle \rightarrow \langle d{:}1, d{:}0\rangle$

Although it will play a central role, the extraction mapping alone is not sufficient to identify the “correct” implementation of *Buf* in the presence of faults. What one also needs is an ability to relate the refusals of *Buf''* and *Buf'''* with the possible refusals of the base process *Buf*, so as to discriminate them on this basis. This, however, is much harder than relating traces. As outlined in the next section, in our past research we abandoned the ambition of interpreting refusals performed by implementations, and simply introduced a device (*extraction pattern*) intended to constrain good implementations not to deadlock when they shouldn’t, i.e., essentially, when their interaction is not “complete” yet. In the case of replication and majority voting, an inconsistent trace would be deemed incomplete. E.g., assume (for a different example from the previous one) that at most one of the replicated output channels d_1, d_2, d_3 is faulty; then trace $\langle d_3{:}1, d_1{:}0\rangle$ is incomplete, in that it fails to tell whether 0 or 1 is intended as an output, whereas $\langle d_3{:}1, d_1{:}0, d_2{:}1\rangle$ reveals that output 1 is the desired one (under the said fault assumption) and that replica d_1 was the one to behave erroneously.

3. PREVIOUS RESULTS

In our previous work (see, e.g., [4, 5, 9, 10]), we regard processes P_1, ..., P_n as forming a *network* if no channel is shared by more than two processes. (Note that such a process network assumes a one-to-one interprocess communication.) We then define $P_1 \otimes \ldots \otimes P_n$ to be the process obtained by taking the parallel composition of P_1, ..., P_n and then hiding all interprocess communication, i.e., the process $(P_1 \| \ldots \| P_n) \backslash B$, where B is the set of actions shared by any two different processes in the network. Network composition is commutative and associative.

In a recent work [10], we place restriction on the kind of allowed base processes, called *input-output* (IO) *processes*, by assuming them to be non-diverging and with *value independent* input channels. Intuitively, in an IO process, the data component of a message arriving on an input channel c is irrelevant as far as accepting it is concerned; thus, if one such message can be refused, then so can any other message. In practice, standard programming constructs like *c?x* for receiving messages give rise to value independent input channels. The requirement that an IO process P should be non-diverging is standard in a CSP based framework, as divergences basically signify totally unacceptable behaviour. The class of base IO processes is compositional, i.e., a network of IO processes is an IO process provided that no divergence arises from its construction.

The notion of extraction pattern *ep* (used in [4, 5, 9, 10]) relates behaviour on a set of "source" channels, B, in an implementation process, to that on a "base" channel, b, in the target process. It has two main functions: that of interpretation of behaviour, necessitated by interface difference, and the encoding of some correctness requirements. The key part of *ep* is an *extraction* mapping, *extr*, which interprets a trace over the source channels B in terms of a trace over base channel b, thought of as belonging to the target process (see the previous section for more explanations). Moreover, mapping *extr*, by way of its domain *dom*, identifies behaviour over source channels that is correct functionally (i.e., in terms of traces). Indeed, "incorrect" traces over B need not (or cannot sensibly) be interpreted, thus making the domain of *extr* potentially smaller. Another mapping, *ref*, is used to define correct behaviour in terms of failures, as it gives bounds on refusals after execution of a particular trace sequence over the source channels. The extraction mapping *extr* should be monotonic, as receiving more information cannot decrease the current knowledge about the transmission. Both notions can be lifted to a finite set of extraction patterns, operating on disjoint sets of channels.

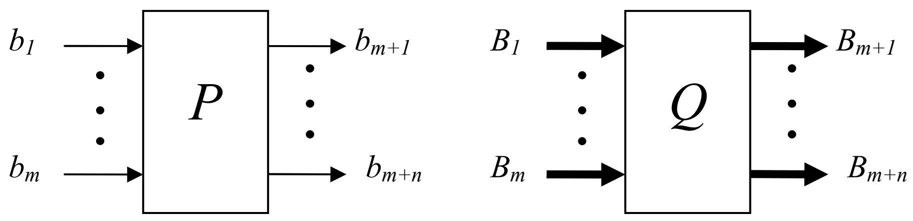

Figure 2. Base process P and its implementation Q

Suppose that we intend to implement a base IO process P in Figure 2 using another process Q with a possibly different communication interface, as in Figure 2 where thick arrows represent *sets* of channels. The correctness of the implementation will be expressed in terms of two sets of extraction patterns, $In = \{ep_1, ..., ep_m\}$ and $Out = \{ep_{m+1}, ..., ep_{m+n}\}$, where each ep_i is an extraction pattern from B_i to b_i. The former set (with sources *inQ* and targets *inP*) will be used to relate the communication on the input channels of P and Q, the latter will serve a similar purpose for the output channels.

Under the above assumptions, Q is an *implementation* of P w.r.t. sets of extraction patterns In and Out, denoted $Q \trianglelefteq_{In:Out} P$, if the following hold:

- all correct traces of Q can be interpreted as traces of P;
- it is not possible to execute Q indefinitely without extracting any actions of P; and
- a refusal by Q on a source channel set B_i, exceeding the bound set by ref_i, must correspond to a refusal by P to interact at all on the target channel b_i.

A direct comparison of an implementation process Q with the base process P is only possible if there is no difference in the respective communication interfaces. This corresponds to the situation that both In and Out are sets of *identity* extraction patterns with $B_i = b_i$ and $extr_i$ an identity mapping (*ref* may be left undefined). In such a case, we simply denote $Q \trianglelefteq P$.

If $Q \trianglelefteq P$ then, in particular, all the refusals on input channels are preserved *entirely*, while for output channels any refusal by Q to output anything on a given channel is also present in P. The latter should indeed be considered as a very satisfactory state of affairs: Q will never fail to provide an output consistent with the specification, unless the specification process itself explicitly allows no output at all to be produced.

One can therefore consider that $Q \trianglelefteq P$ embodies a fully adequate notion of *realisability*. To further justify this, it is interesting to compare it with the standard *refinement* ordering of CSP, denoted by $\sqsupseteq$, such that $Q \sqsupseteq P$ (i.e., Q "CSP implements or *refines*" P) basically amounts to stating that $\varphi Q \subseteq \varphi P$.

To start with, it is not difficult to check that $Q \sqsupseteq P$ implies $Q \trianglelefteq P$. Moreover, $\trianglelefteq$ collapses to $\sqsupseteq$ for the rather wide class of *output-determined* IO base processes (for such a process, the result produced on a given output channel is deterministic at any given point of its execution). Another significant comparison can be made in terms of what can be established by considering the way P and Q interact with a possible environment, as shown in Figure 3.

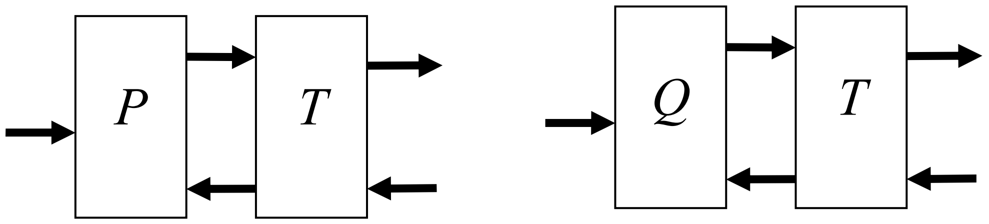

Figure 3. Relating base and implementation processes in the context of an environment

Here P is any base IO process, $Q \trianglelefteq P$ its implementation w.r.t. suitable identity extraction patterns, and T an IO process representing the environment. It can then be shown that we have $Q{\otimes}T \sqsupseteq P{\otimes}T$. Thus $Q{\otimes}T$ is at least as *deterministic* a process as $P{\otimes}T$ in the sense of CSP (see [8, 14]). This makes Q at least as good as P (and possibly much better) as a process to be used in practice.

We finally recall a fundamental result, that the implementation relation is compositional. Let K and L be two base IO processes whose composition is non-diverging, as in Figure 4, and let Epc, Epd, Epe, Epf, Epg and Eph be sets of extraction patterns whose targets are respectively the channel sets C, D, E, F, G and H. Then:

$M \trianglelefteq_{Epc \cup Eph\,:\,Epd \cup Epe} K$ and $N \trianglelefteq_{Epd \cup Epf\,:\,Epg \cup Eph} L$ imply $M \otimes N \trianglelefteq_{Epc \cup Epf : Epe \cup Epg} K \otimes L$.

Hence the implementation relation is preserved through, or distributes over, network composition, and the only restriction on combining base processes is that their network should be designed in a divergence-free way.

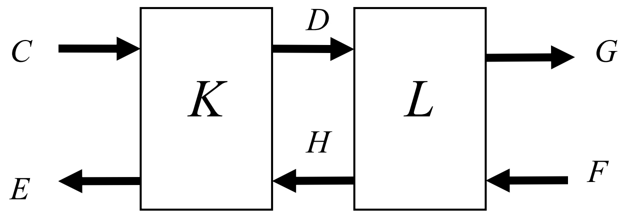

Figure 4. Base processes used in the formulation of the compositionality result

4. EXTENDED MODEL OF PROCESS ABSTRACTION

In the proposed extension of the previous approach, we make a simplifying assumption that all processes are divergence-free. This is aimed mainly at easing the presentation, without effectively restricting the applicability of the resulting technique. As a consequence, a CSP process P can be identified with the pair (αP, φP) (ignoring the δP attribute). Furthermore, since hiding can introduce divergence, we assume in this section that it is a partial operation defined only if divergence is not generated. In fact, hiding leading to a divergence indicates a serious mistake in the construction of a process; this should be detected and eliminated before the proposed abstraction approach is applied, in full agreement with the standard CSP philosophy. It should be stressed that in this section we:

- do not make any assumptions about the channels a process can use, just work with the generic alphabets; and
- no longer assume any special properties of the base processes.

To begin with, we introduce a general definition of an abstraction mapping, which generalises the role played by the extraction patterns.

Definition 1. Let *Src* (sources) and *Trg* (targets) be finite non-empty sets of actions. An *abstraction* from *Src* to *Trg* is a pair of mappings *abs* = (*extr*, *ref*), where:

- $extr : dom \rightarrow Trg^*$, where $dom \subseteq Src^*$, is a mapping from traces over *Src* to the traces over *Trg*. It is assumed that the domain *dom* is non-empty and prefix-closed, and that *extr* is monotonic, strict and effective. The latter means that, for every infinite sequence $t_1 < t_2 < \ldots$ of traces in domain *dom*, the sequence $extr(t_1) \leq extr(t_2) \leq \ldots$ is unbounded.
- $ref : dom \times \mathbf{P}(Src) \rightarrow \mathbf{P}(Trg)$ is a total mapping which is *subset-monotonic* and *terminable*. The former means that $R \subseteq R'$ implies $ref(t, R) \subseteq ref(t, R')$, and the latter that $ref(t, Src) = Trg$.

Moreover, we denote $abs(t, R) = (extr(t), ref(t, R))$, for all $t \in dom$ and $R \subseteq Src$, and call *extr* and *ref* an *extraction* and *refusal* mappings, respectively. Note that *ref* denotes here a completely different mapping from that used in the definition of an extraction pattern in the previous section; indeed, it overcomes our previous difficulties to interpret refusals.

The intuition behind the above definition is that if (t, R) is a trace/refusal pair for a set of actions *Src* in an implementation process, then these should be interpreted as the trace/refusal pair $abs(t, R)$ for a set of actions *Trg* in a base process. Since *extr* is effective, it is impossible for an implementation to execute an infinite trace which is "invisible" as far as the base process is concerned. In some sense this can be viewed as generalising a divergence freedom requirement on the application of abstraction. Moreover, as *ref* is terminable, any termination in an implementation process must correspond to termination in the specification process.

5. AN ALGEBRA OF ABSTRACTIONS

To develop a satisfactory treatment of (interface) abstraction for constructors other than network composition, we now look at the problem of composing abstractions so as to match the way CSP processes are constructed. Note that in [10] we already demonstrated that in some cases it is possible to combine interface abstractions expressed through extraction patterns; here, however, we aim at a general treatment.

In what follows, given $dom \subseteq Src^*$ and $dom' \subseteq Src'^*$, we define $dom \| dom'$ to be the set of all traces over $(Src \cup Src')^*$ which, when projected on *Src*, give a trace in *dom*, and likewise for *Src'*.

We first characterise pairs of abstractions which can be composed. Below, we assume that $abs = (extr, ref)$ and $abs' = (extr', ref')$ are two abstractions, respectively from *Src* to *Trg*, and from *Src'* to *Trg'*, and with domains *dom* and *dom'*.

Definition 2. Two abstractions, *abs* and *abs'*, are:

- *disjoint* if $Src \cap Src' = Trg \cap Trg' = \emptyset$.
- *overlapping* if $Src = Src'$ and $Trg = Trg'$.
- *extr-compatible* if $extr(t) = extr'(t)$, for all $t \in dom \cap dom'$.
- *ref-compatible* if $ref(t, R) = ref'(t, R)$, for all $t \in dom \cap dom'$ and $R \subseteq Src \cap Src'$.

For a pair of disjoint abstractions, *abs* and *abs'*, we now construct an abstraction from $Src \cup Src'$ to $Trg \cup Trg'$ with the domain $dom \| dom'$, denoted by $abs \oplus abs'$. The extraction mapping is defined by induction on the length of traces, by stipulating that it be strict and, for all traces $t \circ \langle a \rangle$ in the domain of $abs \oplus abs'$, $extr_{abs \oplus abs'}(t \circ \langle a \rangle)$ is equal to:

- $extr_{abs \oplus abs'}(t) \circ (extr(t \lceil Src \circ \langle a \rangle) - extr(t \lceil Src))$ if $a \in Src$; and
- $extr_{abs \oplus abs'}(t) \circ (extr'(t \lceil Src' \circ \langle a \rangle) - extr'(t \lceil Src'))$ otherwise.

The definition of the refusal mapping is more straightforward, as for all $t \in dom_{abs \oplus abs'}$ and $R \subseteq Src \cup Src'$, we have:

- $ref_{abs \oplus abs'}(t, R) = ref(t \lceil Src, R \cap Src) \cup ref'(t \lceil Src', R \cap Src')$.

Intuitively, $abs \oplus abs'$ captures the situation when the two abstractions operate on disjoint parts of the interface of a system, and the interpretations of behaviours executed at these interfaces are independent of each other (though the behaviours may be related in the semantic or operational sense).

Proposition 1. $abs \oplus abs'$ is an abstraction in the sense of Definition 1.

Proof. The mapping $extr'' = extr_{abs \oplus abs'}$ is strict by definition. To show that it is monotonic and effective we proceed as follows.

- $extr''$ is monotonic as $extr_{abs \oplus abs'}(t \circ \langle a \rangle)$ has the from $extr_{abs \oplus abs'}(t) \circ u$.
- To see that $extr''$ is effective we first observe that, for all $t \in dom \| dom'$, we have $extr''(t) \lceil Trg = extr(t \lceil Src)$ and $extr''(t) \lceil Trg' = extr(t \lceil Src')$. Thus, since both $extr$ and $extr'$ are effective, an unboundedly growing sequence of traces in the domain of $extr''$ will result in an extracted unbounded sequence of traces.

To show that $ref'' = ref_{abs \oplus abs'}$ is subset-monotonic and terminable we proceed as follows.

- Let $t \in dom \| dom'$ and $R \subseteq R' \subseteq Src \cup Src'$. We then have:

  $$ref''(t, R) = ref(t \lceil Src, R \cap Src) \cup ref'(t \lceil Src', R \cap Src') \subseteq$$
  $$ref(t \lceil Src, R' \cap Src) \cup ref'(t \lceil Src', R' \cap Src') = ref''(t, R'),$$

  where the inclusion holds since both ref and ref' are subset-monotonic.
- $ref''(t, Src \cup Src') = ref(t \lceil Src, Src) \cup ref'(t \lceil Src', Src') = Trg \cup Trg'$, where the second equality holds since ref and ref' are both terminable.

This completes the proof. □

For a pair of overlapping extr-compatible and ref-compatible abstractions, abs and abs', we construct an abstraction from $Src = Src'$ to $Trg = Trg'$ with the domain $dom \cup dom'$, denoted by $abs \sqcap abs'$. It is assumed that, for all traces t in the domain of $abs \sqcap abs'$ and $R \subseteq Src$, we have:

- $abs \sqcap abs'(t, R) = abs(t, R)$ if $t \in dom$; and
- $abs \sqcap abs'(t, R) = abs'(t, R)$ otherwise.

Intuitively, $abs \sqcap abs'$ captures the situation when the two abstractions characterise the behaviour of two alternative versions of a subsystem.

Proposition 2. $abs \sqcap abs'$ is an abstraction in the sense of Definition 2.

Proof. The mapping $extr'' = extr_{abs \sqcap abs'}$ is strict by definition. To show that it is monotonic and effective we proceed as follows.

- To see that $extr''$ is monotonic we observe that if $t \leq u$ and $u \in dom \cup dom'$, then we have $t,u \in dom$ or $t,u \in dom'$. We can therefore apply the monotonicity of $extr$ or $extr'$.
- To see that $extr''$ is effective we first observe that if $t_1 < t_2 < \ldots$ are traces in the domain $dom \cup dom'$, then they are all in dom or in dom' (or in $dom \cap dom'$). We can therefore apply the effectiveness of $extr$ or $extr'$.

That $ref'' = ref_{abs \sqcap abs'}$ is subset-monotonic and terminable follows from its definition and the fact that both ref and ref' are subset-monotonic and terminable. □

From a pair of overlapping extr-compatible abstractions, *abs* and *abs'*, we construct an abstraction from $Src = Src'$ to $Trg = Trg'$ with the domain $dom \cap dom'$, denoted by $abs \| abs'$. It is assumed that, for all traces *t* in the domain of $abs \sqcap abs'$ and $R \subseteq Src$,

- $abs \| abs'(t, R) = (extr(t), ref(t, R) \cup ref'(t, R))$.

Intuitively, $abs \| abs'$ captures the situation when the two abstractions characterise the behaviour of two parallel subsystems at a shared interface.

Proposition 3. $abs \| abs'$ is an abstraction in the sense of Definition 1.

Proof. The mapping $extr'' = extr_{abs\|abs'}$ is strict, monotonic and effective since *extr* is. To show that $ref'' = ref_{abs\|abs'}$ is subset-monotonic and terminable we proceed as follows.

- Let $t \in dom \cap dom'$ and $R \subseteq R'$. We then have: $ref''(t, R) = ref(t, R) \cup ref'(t, R) \subseteq ref(t, R') \cup ref'(t, R') = ref''(t, R')$, where the inclusion holds since both *ref* and *ref'* are subset-monotonic.
- $ref''(t, Src) = ref(t, Src) \cup ref'(t, Src) = Trg$, where the equality holds since both *ref* and *ref'* are terminable.

This completes the proof. □

As in the case of extraction patterns and the realisability results recalled in the previous section, we also need some kind of abstraction allowing a direct comparison of interactions. The *identity* abstraction for a set of actions *A* is an abstraction $idabs_A$ from $Src = A$ to $Trg = A$ with the domain A^*, and such that $idabs_A(t, R) = (t, R)$, for all $t \in A^*$ and $R \subseteq A$.

Proposition 4. If $A \cap A' = \varnothing$ then $idabs_{A \cup A'} = idabs_A \oplus idabs_{A'}$.

Proof. We have the following, for all $t \in (A \cup A')^*$ and $R \subseteq A \cup A'$:

$$idabs_A \oplus idabs_{A'}(t, R) = (t, ref_{idabsA}(t \lceil A, R \cap A) \cup ref_{idabsA'}(t \lceil A', R \cap A')) =$$
$$(t, R \cap A \cup R \cap A') = (t, R) = idabs_{A \cup A'}(t, R).$$

This completes the proof. □

5.1 Implementation relation

We now introduce the central notion of this paper which, despite its simplicity, is all we need in order to capture what it means for one process to correctly implement another, base, process with a possibly different communication interface.

Suppose that we intend to implement a base process *P* using another implementation process *Q* with possibly different alphabet. The correctness of the implementation will be expressed in terms of an abstraction from the alphabet of *Q* to that of *P*.

Definition 3. Let *P* and *Q* be processes and *abs* be an abstraction from αQ to αP. Then *Q* is an *implementation* of *P* w.r.t. *abs* if $\tau Q \subseteq dom$ and $abs(\varphi Q) \subseteq \varphi P$. We denote this by $Q \trianglelefteq_{abs} P$.

(Note that we have $\tau Q = \{t \mid (t, R) \in \varphi Q\}$ and so $\tau Q \subseteq dom$ ensures that all failures of *Q* can be interpreted by the abstraction *abs*.)

Hence it is possible to interpret the whole behaviour of an implementation process as (part of) the behaviour of a base process. This is, clearly, very close to the idea of one process being a refinement of another in the standard treatment of CSP processes.

A direct comparison of an implementation process Q with the corresponding base process P is only possible if they have the same alphabet A. Then, if $Q \trianglelefteq_{abs} P$, where $abs = idabs_A$, we simply denote $Q \trianglelefteq P$ and obtain the strongest possible realisability result.

Theorem 1. $Q \trianglelefteq P$ if and only if $Q \sqsupseteq P$.

Proof. Follows immediately from the definitions. □

5.2 Compositionality results

We now establish compositionality properties linking the proposed algebra of abstractions with the algebra of CSP processes.

We start by assuming that there are two base processes, P and P', working in parallel, and two implementation processes, Q and Q', also working in parallel. In addition to that there are four abstractions:

- abs from $A = \alpha Q\text{-}\alpha Q'$ to $B = \alpha P\text{-}\alpha P'$;
- abs' from $C = \alpha Q'\text{-}\alpha Q$ to $D = \alpha P'\text{-}\alpha P$; and
- abs'' and abs''', both from $E = \alpha Q \cap \alpha Q'$ to $F = \alpha P \cap \alpha P'$, which are extr-compatible.

Theorem 2. Assuming the above, if $Q \trianglelefteq_{abs\oplus abs''} P$ and $Q' \trianglelefteq_{abs'\oplus abs'''} P'$, then:

$$Q \| Q' \trianglelefteq_{abs\oplus abs'\oplus(abs''\|abs''')} P \| P'.$$

Proof. Let $abso = abs\oplus abs'\oplus(abs''\|abs''')$. From $Q \trianglelefteq_{abs\oplus abs''} P$ and $Q' \trianglelefteq_{abs'\oplus abs'''} P'$ it follows that:

$$\tau Q \subseteq dom\|dom'',\ abs\oplus abs''(\varphi Q)\subseteq\varphi P,\ \tau Q'\subseteq dom'\|dom''',\ abs'\oplus abs'''(\varphi Q')\subseteq\varphi P'.$$

Hence we have:

$$\tau(Q\|Q') \subseteq \tau Q\|\tau Q' \subseteq (dom\|dom'')\|(dom'\|dom''') =$$
$$dom\|dom'\|(dom''\cap dom''') = dom_{abso}.$$

Suppose now that $(t, R)\in\varphi(Q\|Q')$. Then we have:

$$abso(t, R) = (extr_{abso}(t),\ ref(t\lceil A, R\cap A)\cup ref'(t\lceil C, R\cap C)\cup ref_{abs''\|abs'''}(t\lceil E, R\cap E)) =$$
$$(extr_{abso}(t),\ ref(t\lceil A, R\cap A)\cup ref'(t\lceil C, R\cap C)\cup ref''(t\lceil E, R\cap E)\cup ref'''(t\lceil E, R\cap E)) =$$
$$(extr_{abso}(t),\ ref_{abs\oplus abs''}(t\lceil A\cup E, R\cap(A\cup E))\cup ref_{abs'\oplus abs'''}(t\lceil C\cup E, R\cap(C\cup E))).$$

We now observe that $(t\lceil A\cup E, R\cap(A\cup E))\in\varphi Q$ and so:

$$(extr_{abso}(t)\lceil B\cup F,\ ref_{abs\oplus abs''}(t\lceil A\cup E, R\cap(A\cup E))) =$$
$$(extr_{abs\oplus abs''}(t\lceil A\cup E),\ ref_{abs\oplus abs''}(t\lceil A\cup E, R\cap(A\cup E))) \in \varphi P$$

And, similarly,

$$(extr_{abso}(t)\lceil D\cup F,\ ref_{abs'\oplus abs'''}(t\lceil C\cup E, R\cap(C\cup E))) \in \varphi P'.$$

Hence $abso(t, R)\in\varphi(P\|P')$. □

The next compositionality result concerns non-deterministic choice.

Theorem 3. Let $Q \trianglelefteq_{abs} P$ and $Q' \trianglelefteq_{abs'} P'$ where abs and abs' are extr-compatible and ref-compatible overlapping abstractions. Then

$$Q\sqcap Q' \trianglelefteq_{abs\sqcap abs'} P\sqcap P'.$$

Proof. From $Q \trianglelefteq_{abs} P$ and $Q' \trianglelefteq_{abs'} P'$ we have: $\tau Q \subseteq dom$, $abs(\varphi Q) \subseteq \varphi P$, $\tau Q' \subseteq dom'$ and $abs(\varphi Q') \subseteq \varphi P'$. Hence $\tau(Q \sqcap Q') = \tau Q \cup \tau Q' \subseteq dom \cup dom' = dom_{abs \sqcap abs'}$. Suppose now that $(t,R) \in \varphi(Q \sqcap Q')$. Without loss of generality, we may assume that $t \in \tau Q$. Hence

$$abs_{abs \sqcap abs'}(t, R) = abs(t, R) \in \varphi P \subseteq \varphi(P \sqcap P')$$

where the $abs(t, R) \in \varphi P$ follows from $Q \trianglelefteq_{abs} P$. Hence the result holds. □

The last compositionality result deals with hiding.

Theorem 4. Let $Q \trianglelefteq_{abs \oplus abs'} P$, and $P \backslash Trg$ be a well-defined process. Then $Q \backslash Src$ is also defined and

$$Q \backslash Src \trianglelefteq_{abs'} P \backslash Trg.$$

Proof. The fact that *abs* is effective ensures that hiding the actions *Src* in *Q* does not create divergence in view that hiding the actions *Trg* does not create divergence in *P*. Thus $Q \backslash Src$ is defined. From $Q \trianglelefteq_{abs \oplus abs'} P$ we have $\tau Q \subseteq dom \| dom'$ and $abs \oplus abs'(\varphi Q) \subseteq \varphi P$. Hence since $Q \backslash Src$ does not create divergence, $\tau(Q \backslash Src) \subseteq dom'$. Suppose now that $(t, R) \in \varphi(Q \backslash Src)$. Then, since $Q \backslash Src$ does not create divergence, there is $(u, R \cup Src) \in \varphi Q$ such that $t = u \lceil Src'$. Moreover, $abs \oplus abs'(u, R \cup Src) \in \varphi P$. We then observe that the following hold:

$$abs \oplus abs'(u, R \cup Src) = (extr_{abs \oplus abs'}(u), ref(u \lceil Src, Src) \cup ref'(u \lceil Src', R)) =$$
$$(extr_{abs \oplus abs'}(u), Trg \cup ref'(t, R)).$$

Hence $(extr_{abs \oplus abs'}(u) \lceil Trg', ref'(t, R)) \in \varphi(P \backslash Trg)$. But $extr_{abs \oplus abs'}(u) \lceil Trg' = extr'(t)$, which implies that $abs'(t, R) \in \varphi(P \backslash Trg)$. □

We have thus demonstrated how one can deal with abstractions in the context of parallel composition, hiding and non-deterministic choice. Other standard operators of CSP, such as prefixing, can be treated in a similar way.

6. CONCLUSIONS

We have outlined a general compositional approach which allows one to deal with abstractions in networks of communicating processes. There are several issues which are of an immediate interest for further work, such as applying the proposed approach to significant case studies and providing the treatment for other CSP operators. However, we feel that the crucial one is a provision of algorithms and tools for checking whether an implementation relation between two processes indeed holds. Such a problem can be attempted using techniques similar to those introduced in [4] for the setup based on extraction patterns.

This research was supported by the EC IST grant 511599 and the NSFC project 60433010.

REFERENCES

[1] Abadi, M. and Lamport, L., 1991. *The Existence of Refinement Mappings*. In Theoretical Computer Science, Vol. 82, pp 253-284.

[2] Brinksma, E. et al., 1991. Refining Interfaces of Communicating Systems. Proceedings of Coll. on Combining Paradigms for Software Development. LNCS, Vol. 494, Springer, pp 297-312.

[3] Burstall, R.M. and Darlington, J., 1977. *A Transformation System for Developing Recursive Programs*. In Journal of the ACM, Vol. 24, pp 44-67.

[4] Burton, J. et al., 2004. *Relating Communicating Processes with Different Interfaces*. In Fundamenta Informaticae, Vol. 59, pp 1-37.

[5] Burton, J. et al., 2002. *Compositional Development In the Event of Interface Difference*. In Concurrency in Dependable Computing, Kluwer Academic Publishers, pp 1-20.

[6] Jonsson, B., 1994. Compositional Specification and Verification of Distributed Systems. In ACM TOPLAS, Vol. 16, pp 259-303.

[7] Gerth, R. et al., 1992. *Interface Refinement in Reactive Systems*. Proceedings of CONCUR'92. LNCS, Vol. 630, Springer, pp 77-93.

[8] Hoare, C.A.R., 1985. *Communicating Sequential Processes*. Prentice Hall.

[9] Koutny, M and Pappalardo, G., 2001. *Behaviour Abstraction for Communicating Sequential Processes*. In Fundamenta Informaticae, Vol. 48, pp 21-54.

[10] Koutny, M. et al., 2006. *Towards an Algebra of Abstractions for Communicating Processes*. Proceedings of ACSD'06, IEEE Computer Society, pp 239-250.

[11] Lamport, L., 1978. *The Implementation of Reliable Distributed Multiprocess Systems*. In Computer Networks, Vol. 2, pp 95-114.

[12] Milner, R., 1989. *Communication and Concurrency*. Prentice Hall.

[13] Rensink, A. and Gorrieri, R., 2001. *Vertical Implementation*. In Information and Computation, Vol. 170, pp 95-133.

[14] Roscoe, A.W., 1998. *The Theory and Practice of Concurrency*. Prentice-Hall.

[15] Schepers, H. and Hooman, J., 1993. *Trace-based Compositional Reasoning About Fault-tolerant Systems*. Proceedings of PARLE'93. LNCS, Vol. 694, Springer, pp 197-208.