Proceedings of the International Multiconference on
Computer Science and Information Technology pp. 745–754

ISSN 1896-7094
© 2007 PIPS

Compositional Abstractions for Interacting Processes

Maciej Koutny[1], Giuseppe Pappalardo[2], and Marta Pietkiewicz-Koutny[1]

[1] School of Computing Science, Newcastle University,
Newcastle upon Tyne NE1 7RU, U.K.
{marta.koutny,maciej.koutny}@ncl.ac.uk
[2] 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. But so far abstraction has been shown to distribute only over network composition which restricts its usefulness for compositional correctness analysis. In this paper we extend the treatment to other process constructs which proved to be useful in the development of complex distributed applications.

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, *conveniently 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 for 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 framework outlined, 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 both the implementation and the target process expressions, 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.

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

In our previous work (see, e.g., [4, 5, 10]), we proposed abstraction-based implementation relations satisfying *realisability*, which is a property ensuring that an implementation may be put to good use, and *compositionality* which requires 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 in avoiding the state explosion problem in carrying out automated verification.

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

Another limitation of our previous results was that interprocess communication was assumed to be point-to-point, with typed input and output channels. We here remove this restriction as well, so that the proposed treatment can 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. The following section outlines our previous approach and results, and the last section sketches the proposed extension. Detailed motivations for the work on abstraction of interprocess communication, illustrative examples, and description of other related work, in particular [1, 2, 7, 11, 13, 15], can be found in [4, 9].

1 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 unproductive internal loops, forming the set of P's *divergences*, δP.

We now recall the standard CSP 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 in the intersection of their alphabets. Parallel composition is commutative and associative. The deterministic choice, $P [] Q$, and non-deterministic choice, $P \sqcap Q$, 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). Prefixing, $a \to P$, is a construct which first executes a and then behaves like P. Hiding a set of actions A in P yields process $P \backslash A$ that behaves like P with the actions occurring at the channels 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.

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. For example, $P = (a \to P) [] (b \to \text{STOP})$ 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). We use the following notations concerning traces.

- The empty trace is denoted by $\langle \rangle$, and $\circ$ is the concatenation operation for traces.
- $w \leq u$ means that trace w is a prefix of u, and $w < u$ that w is a proper prefix of u.
- A mapping f from traces to traces is monotonic if $t \leq u$ implies $f(t) \leq f(u)$.
- If w is a prefix of u then $u - w$ is the suffix of u after deleting the initial w.
- Given a trace t and a set of actions A, the trace $t \upharpoonright A$ is obtained by deleting from t all the actions that do not belong to A.

2 Previous results

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

We also use structured actions of the form $b{:}v$ with v a data *message* and b a communication *channel*, and partition the channels of a process P into the input channels, $in\ P$ (depicted by incoming arrows), and output channels, $out\ P$ (depicted by outgoing arrows). No two processes in a network have a common input (or output) channel.

In [10], we place restriction on the kind of allowed base processes, called *input-output process* (IO processes), by assuming them to be non-diverging processes with

all input channels being *value independent*. 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; 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 the composition does not generate a divergence.

The notion of extraction pattern ep (used in $[5, 9, 4]$) 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). Moreover, mapping $extr$, by way of its domain, 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 $extr$ is 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.

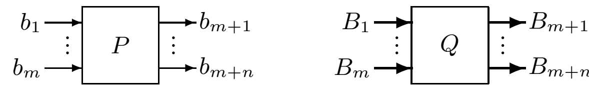

Fig. 1. Base process P and its implementation Q.

Suppose that we intend to implement a base IO process P in Figure 1 using another process Q with a possibly different communication interface, as in Figure 1 where thick arrows represent *sets* of channels.. The correctness of the implementation will be expressed in terms of two sets of extraction patterns, $In = \{ep_1, \ldots, ep_m\}$ and $Out = \{ep_{m+1}, \ldots, ep_{m+n}\}$, where each ep_i is an extraction pattern from B_i to b_i. The former (with sources $in\,Q$ and targets $in\,P$) 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 $Q \preceq^{In}_{Out} P$, if the following hold: (i) all correct traces of Q can be interpreted as traces of P; (ii) it is not possible to execute Q indefinitely without extracting any actions of P; and (iii) a refusal by Q on a source channel set B_i,

exceeding the bound set by ref, 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 $Q \preceq P$.

If $Q \preceq P$ 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 $Q \preceq P$ 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 $\phi Q \subseteq \phi P$.

To start with, it is not difficult to check that $Q \sqsupseteq P$ implies $Q \preceq P$. Moreover, $\preceq$ 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 2.

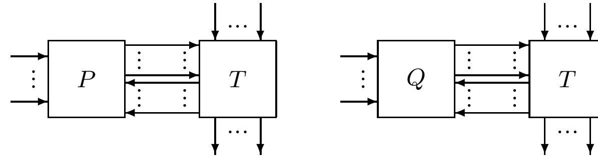

Fig. 2. Relating base (P) and implementation (Q) processes in the context of an environment T.

Here P is any base IO process, $Q \preceq P$ its implementation w.r.t. suitable identity extraction patterns, and T an IO process representing the environment. It can then be shown that $Q \otimes T \sqsupseteq P \otimes T$. 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 have 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 3, and let Ep_c, Ep_d, Ep_e, Ep_f, Ep_g and Ep_h be sets of extraction patterns whose targets are respectively the channel sets C, D, E, F, G and H. Then

$$M \preceq^{Ep_c \cup Ep_h}_{Ep_d \cup Ep_e} K \wedge N \preceq^{Ep_d \cup Ep_f}_{Ep_g \cup Ep_h} L \implies M \otimes N \preceq^{Ep_c \cup Ep_f}_{Ep_e \cup Ep_g} K \otimes L \,.$$

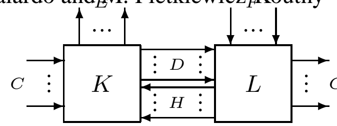

Fig. 3. Base processes used in the formulation of the compositionality result.

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.

3 Extending the 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 $(\alpha P, \phi P)$ (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: (i) do not make any assumptions about the channels a process can use, just work with the generic alphabets; and (ii) 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 $abs \stackrel{\text{df}}{=} (extr, ref)$, where:*

— *$extr : dom \to Trg^*$, 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 (i.e., $extr(\langle\,\rangle) = \langle\,\rangle$), and effective (i.e., for every infinite sequence $t_1 < t_2 < \ldots$ of traces in dom, the sequence $extr(t_1) \leq extr(t_2) \leq \ldots$ is unbounded).*

— *$ref : dom \times 2^{Src} \to 2^{Trg}$ is a total mapping which is subset-monotonic (i.e., if $t \in dom$ and $R \subseteq R' \in 2^{Src}$ then $ref(t, R) \subseteq ref(t, R')$) and terminable (i.e., $ref(t, Src) = Trg$, for all $t \in dom$).*

Moreover, we denote $abs(t, R) \stackrel{\text{df}}{=} (extr(t), ref(t, R))$, for all $t \in dom$ and $R \subseteq Src$, and call $extr$ and ref an extraction and refusal mappings, respectively.

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.

3.1 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.

In what follows, given $dom \subseteq Src^*$ and $dom' \subseteq Src'^*$, we define $dom \| dom' \subseteq (Src \cup Src')^*$ to be the set of all traces 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* $Src \cap Src' = Trg \cap Trg' = \varnothing$.
- overlapping *if* $Src \cap Src' = Trg \cap Trg' = \varnothing$.
- 0–compatible *if* $extr(t) = extr'(t)$*, for all* $t \in dom \cap dom'$.
- 1–compatible *if* $ref(t, R) = ref'(t, R)$*, for all* $t \in dom \cap dom'$ *and* $R \subseteq Src \cap Src'$.
- 2–compatible *if, for all* $t \in dom \cap dom'$ *and* $R, R' \subseteq Src \cap Src'$,

$$ref(t, R \cup R') \cup ref'(t, R \cup R') = ref(t, R) \cup ref'(t, R') \ .$$

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

$$extr_{abs \oplus abs'}(t \circ \langle a \rangle) \stackrel{\text{df}}{=} \begin{cases} extr_{abs \oplus abs'}(t) \circ \\ \quad (extr(t \upharpoonright Src \circ \langle a \rangle) - extr(t \upharpoonright Src)) & \text{if } a \in Src \\ extr_{abs \oplus abs'}(t) \circ \\ \quad (extr'(t \upharpoonright Src' \circ \langle a \rangle) - extr'(t \upharpoonright Src')) & \text{otherwise} \ . \end{cases}$$

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:

$$ref_{abs \oplus abs'}(t, R) \stackrel{\text{df}}{=} ref(t \upharpoonright Src, R \cap Src) \cup ref'(t \upharpoonright Src', R \cap Src') \ .$$

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.*

For a pair of overlapping 0&1–compatible abstractions, abs and abs', we construct an abstraction $abs \sqcap abs'$ from $Src = Src'$ to $Trg = Trg'$, and with the domain $dom_{abs \sqcap abs'} \stackrel{\text{df}}{=} dom \cup dom'$. Moreover, for all $t \in dom_{abs \sqcap abs'}$ and $R \subseteq Src$,

$$(abs \sqcap abs')(t, R) \stackrel{\text{df}}{=} \begin{cases} abs(t, R) & \text{if } t \in dom \\ abs'(t, R) & \text{otherwise .} \end{cases}$$

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 1.*

From a pair of overlapping 0&2–compatible abstractions, abs and abs', we construct an abstraction $abs \| abs'$ from $Src = Src'$ to $Trg = Trg'$, and with the domain $dom_{abs \| abs'} \stackrel{\text{df}}{=} dom \cap dom'$. Moreover, for all $t \in dom_{abs \| abs'}$ and $R \subseteq Src$,

$$(abs \| abs')(t, R) \stackrel{\text{df}}{=} (extr(t), ref(t, R) \cup ref'(t, R)) \ .$$

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.*

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 $dom = A^*$, and such that $idabs_{Src}(t, R) \stackrel{\text{df}}{=} (t, R)$, for all $t \in dom$ and $R \subseteq Src$.

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

3.2 Implementation relation

We now introduce the central notion of this paper which, despite its simplicity, is all we need 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 $abs(\phi Q) \subseteq \phi P$.*[3] *We denote this by $Q \hookrightarrow_{abs} P$.*

[3] Note that $\tau Q \stackrel{\text{df}}{=} \{t \mid (t, R) \in \phi Q\}$ and so the first condition 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 $Q \hookrightarrow_{idabs_A} P$, we simply denote $Q \hookrightarrow P$ and obtain the strongest possible realisability result.

Theorem 1. *$Q \hookrightarrow P$ if and only if $Q \sqsupseteq P$.*

3.3 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: (i) abs from $\alpha Q \setminus \alpha Q'$ to $\alpha P \setminus \alpha P'$; (ii) abs' from $\alpha Q' \setminus \alpha Q$ to $\alpha P' \setminus \alpha P$; and (iii) abs_0 and abs'_0, both from $\alpha Q \cap \alpha Q'$ to $\alpha P \cap \alpha P'$, which are $0\&2$–compatible and overlapping.

Theorem 2. *Assuming the above, if $Q \hookrightarrow_{abs \oplus abs_0} P$ and $Q' \hookrightarrow_{abs' \oplus abs'_0} P'$, then:*

$$(Q \| Q') \hookrightarrow_{abs \oplus abs' \oplus (abs_0 \| abs'_0)} (P \| P') \ .$$

The next compositionality result concerns non-deterministic choice.

Theorem 3. *Let $Q \hookrightarrow_{abs} P$ and $Q' \hookrightarrow_{abs'} P'$ where abs and abs' are $0\&1$–compatible overlapping abstractions. Then $(Q \sqcap Q') \hookrightarrow_{abs \sqcap abs'} (P \sqcap P')$.*

The last compositionality result deals with hiding.

Theorem 4. *Let $Q \hookrightarrow_{abs \oplus abs'} P$, and $P \setminus Trg$ be a well-defined defined process. Then $Q \setminus Src$ is also defined and $(Q \setminus Src) \hookrightarrow_{abs'} (P \setminus Trg)$.*

Proof. (Sketch) 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 $Q \setminus Src$ is defined, and using the terminability of abs we may guarantee that any total blocking on the hidden actions in the implementation process has a corresponding total blocking on the hidden actions in the base process. This allows one to prove the inclusions of abstracted traces and failures. □

We have 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.

4 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 in the future work, such as applying the proposed approach to case studies and adding 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.

Acknowledgement This research was supported by the EC IST grant 511599 (RODIN).

References

1. M. Abadi and L. Lamport: The Existence of Refinement Mappings. *Theoretical Computer Science* 82 (1991) 253-284.
2. E. Brinksma, B. Jonsson, and F. Orava: Refining Interfaces of Communicating Systems. Proc. of *Coll. on Combining Paradigms for Software Development*, Springer-Verlag, Lecture Notes in Computer Science 494 (1991) 297-312.
3. R. M. Burstall and J. Darlington: A Transformation System for Developing Recursive Programs. *Journal of the ACM* 24 (1977) 44-67.
4. J. Burton, M. Koutny and G. Pappalardo: Relating Communicating Processes with Different Interfaces. *Fundamenta Informaticae* 59 (2004) 1-37. 87-96
5. J. Burton, M. Koutny, G. Pappalardo and M. Pietkiewicz-Koutny: Relating Communicating Processes with Different Interfaces. In: *Concurrency in Dependable Computing*, P. Ezhilchelvan and A. Romanovsky (Eds.). Kluwer Academic Publishers (2002) 1-20.
6. P. Collette and C. B. Jones: Enhancing the Tractability of Rely/Guarantee Specifications in the Development of Interfering Operations. CUMCS-95-10-3, Department of Computing Science, Manchester University (1995).
7. R. Gerth, R. Kuiper and J. Segers: Interface Refinement in Reactive Systems. Proc. of *CONCUR '92*, Springer-Verlag, Lecture Notes in Computer Science 630 (1992) 77-93.
8. C. A. R. Hoare: *Communicating Sequential Processes*. Prentice Hall (1985).
9. M. Koutny and G. Pappalardo: Behaviour Abstraction for Communicating Sequential Processes. *Fundamenta Informaticae* 48 (2001) 21-54.
10. M. Koutny, G. Pappalardo and M. Pietkiewicz-Koutny: Towards an Algebra of Abstractions for Communicating Processes. Proc. of *ACSD'06*, IEEE Computer Society (2006) 239-250.
11. L. Lamport: The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks* 2 (1978) 95-114.
12. R. Milner: *Communication and Concurrency*. Prentice Hall (1989).
13. A. Rensink and R. Gorrieri: Vertical Implementation. *Information and Computation* 170 (2001) 95-133.
14. A. W. Roscoe: *The Theory and Practice of Concurrency*. Prentice-Hall (1998).
15. H. Schepers and J. Hooman: Trace-based Compositional Reasoning About Fault-tolerant Systems. Proc. of *PARLE'93*, Springer-Verlag, Lecture Notes in Computer Science 694 (1993) 197-208.