Compositional Development in the Event of Interface Difference

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

[1] Department of Computing Science, University of Newcastle, Newcastle upon Tyne NE1 7RU, U.K.
{j.i.burton,maciej.koutny,marta.koutny}@ncl.ac.uk
[2] Dipartimento di Matematica e Informatica, Università di Catania, I-95125 Catania, Italy,
pappalardo@dmi.unict.it

Abstract. We present here an implementation relation which allows compositional development of a network of communicating processes, in the event that corresponding specification and implementation components have different interfaces. This relation enjoys two basic properties which are fundamental to its success. It is compositional, in the sense that a target composed of several connected systems may be implemented by connecting their respective implementations. In addition, an implementation, when plugged into an appropriate environment, is to all intents and purposes a conventional implementation of the target. We illustrate our approach by outlining the development of a fault-tolerant system based on co-ordinated atomic actions (CA actions). As a formal framework of concurrent computation, we use the model of Communicating Sequential Processes (CSP).
Keywords: Theory of parallel and distributed computation, behaviour abstraction, communicating sequential processes, compositionality, CA actions.

Introduction

Consider the situation that we have a specification network, P_{net}, composed of n processes P_i, where all interprocess communication is hidden. Consider an implementation network, Q_{net}, also composed of n processes, again with all interprocess communication hidden. Assume that there is a one-to-one relationship between component processes in P_{net} and those in Q_{net}. Intuitively, P_i is intended to specify Q_i in some sense. Finally, assume that the interface of Q_{net}, in terms of externally observable actions, is the same as that of P_{net}.

In process algebras, such as those used in [11, 15], the notion that a process Q_{net} *implements* a process P_{net} is based on the idea that Q_{net} is more deterministic than (or equivalent to) P_{net} in terms of the chosen semantics. (In the following, we shall also refer to such specifications as *target* or *base* systems.)

The process of refining the target into the implementation also allows the designer to change the control structure of the latter. In such a case, Q_{net} has

implemented P_{net} by describing its *internal* structure in a more concrete and detailed manner. However, we are able to hide the details of that internal structure, and then verify that this new internal structure still gives correct behaviour at the interface of Q_{net}, which is still that of P_{net}. Indeed, the standard notions of refinement, such as those of [11, 15], are interested only in *observable* actions, i.e., in the behaviour available at the *interface* of processes. However, the interfaces of the specification and implementation processes must be the same to facilitate comparison.

A question naturally arises. What if we wish to approach this verification problem compositionally? What if we want to verify that Q_{net} implements P_{net} simply by verifying that Q_i implements P_i, for each $1 \leq i \leq n$. In general, this is only possible if Q_i and P_i have the same communication interface. Thus, Q_i may implement P_i by describing its computation in a more concrete manner, but it may not do so by refining its interface, at least if we wish to carry out compositional verification.

Yet in deriving an implementation from a specification we will often wish to implement abstract, high-level *interface* actions at a lower level of detail and in a more concrete manner. For example, the channel connecting P_i to another component process P_j may be unreliable and so it may need to be replaced by a data channel and an acknowledgement channel. Or P_i itself may be liable to fail and so its behaviour may need to be replicated, with each new component having its own communication channels to avoid a single channel becoming a bottleneck (such a scenario was one of the major historical motivations behind the present work [3, 4, 8, 9]). Or it may simply be the case that a high-level action of P_i has been rendered in a more concrete, and so more implementable, form. As a result, the interface of an implementation process may end up being expressed at a lower (and so different) level of abstraction to that of the corresponding specification process. In the process algebraic context, where our interest lies only in *observable* behaviour, this means that verification of correctness must be able to deal with the case that the implementation and specification processes have different interfaces.

The relation between processes detailed in the remainder of this paper allows us to carry out such compositional verification in the event that Q_i and P_i have different interfaces.

An important notion in the development of this relation is that of *extraction pattern*. Extraction patterns interpret the behaviour of a system at the level of communication traces, by relating behaviour on a set of channels in the implementation to behaviour on a channel in the specification. In addition, they impose some correctness requirements upon the behaviour of an implementation; for example, the traces of the implementation should be correct with respect to the interface encoded by the extraction pattern. The set of extraction patterns defined for all channels in an implementation system appears as a formal parameter in a generic *implementation relation*.

The motivating framework given above for the implementation relation presented here leads to the identification of two natural constraints which must be placed upon any such implementation relation.

The first, *realisability*, ensures that the abstraction built into the implementation relation may be put to good use. In practice, this means that plugging an implementation into an appropriate environment (see, e.g., [8]) should yield a conventional implementation of the specification. A related requirement arises when the implementation relation is parameterized with a special set of extraction patterns, known as *identity* extraction patterns, which essentially formalise the fact that implementation and specification processes are represented at the *same* level of abstraction; in this case, the implementation relation should reduce to a satisfactory notion of behaviour refinement.

Compositionality, the other constraint on the implementation relation, requires it to distribute over system composition. Thus, a specification composed of a number of connected systems may be implemented by connecting their respective implementations. It is remarkable that although in general Q_i and P_i do not have the same interface, we know that, when all of the components Q_i have been composed, the result — namely Q_{net} — will have the same interface as the corresponding specification process — namely P_{net}. Compositionality is important in avoiding the state explosion problem when we approach automatic verification, algorithms for which have been developed in [3].

The paper is organised as follows. In the next section, we introduce some basic notions used throughout the paper. In section 2, we present extraction patterns — a central notion to characterising the interface of an implementation. Section 3 deals with the implementation relation, while section 4 applies our approach to the compositional development of a concurrent system based on co-ordinated atomic actions. Comparison with other work, in particular [1, 2, 6, 10, 14, 16], can be found in [4, 9].

1 Preliminaries

Traces, failures and divergences Processes are represented in this paper using the failures-divergences model of Communicating Sequential Processes (CSP) [7, 15] — a formal model for the description of concurrent computing systems. A CSP *process* 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 the *alphabet* of the process. A *trace* of the process is a finite sequence of actions that a process can be observed to engage in. In this paper, structured actions of the form $b.v$ will be used, where v is a *message* and b is a communication *channel*. For every channel b, μb is the *message set* of b, i.e., the set of all v such that $b.v$ is a valid action, and $\alpha b \stackrel{\text{df}}{=} \{b.v \mid v \in \mu b\}$ is the *alphabet* of b. For a set of channels B, $\alpha B \stackrel{\text{df}}{=} \bigcup_{b \in B} \alpha b$.

Throughout the paper we use notations similar to those of [7]. A trace $t[b'/b]$ is obtained from trace t by replacing each action $b.v$ by $b'.v$, and $t{\restriction}B$ is obtained by deleting from t all the actions that do not occur on the channels in

B. For example, if $t = \langle b{:}2, c{:}3, b{:}2, c{:}6, d{:}7\rangle$, then $t[e/b] = \langle e{:}2, c{:}3, e{:}2, c{:}6, d{:}7\rangle$ and $t{\restriction}\{b, d\} = \langle b{:}2, b{:}2, d{:}7\rangle$. A mapping from a set of traces to a set of traces $f : T \to T'$ is *monotonic* if $t, u \in T$ and $t \leq u$ implies $f(t) \leq f(u)$, where $\leq$ is the prefix relation on traces. For a set T of traces, $Pref(T)$ is the set of all prefixes of the traces in T. Finally, $\circ$ is the concatenation operation for traces.

We use the standard failures-divergences model of CSP [7, 15] in which a process P is a triple $(\alpha P, \phi P, \delta P)$ where αP (the *alphabet*) is a non-empty finite set of actions, ϕP (the *failures*) is a subset of $\alpha P^* \times \mathbb{P}(\alpha P)$, and δP (the *divergences*) is a subset of αP^*. The conditions imposed on the three components are given below, where τP denotes the *traces* of P, $\tau P \stackrel{\text{df}}{=} \{t \mid \exists R \subseteq \alpha P :\ (t, R) \in \phi P\}$:

- τP is non-empty and prefix-closed.
- If $(t, R) \in \phi P$ and $S \subseteq R$ then $(t, S) \in \phi P$.
- If $(t, R) \in \phi P$ and $a \in \alpha P$ is such that $t \circ \langle a\rangle \notin \tau P$ then $(t, R \cup \{a\}) \in \phi P$.
- If $t \in \delta P$ then $(t \circ u, R) \in \phi P$, for all $u \in \alpha P^*$ and $R \subseteq \alpha P$.

Moreover,we will associate with P a set of channels, χP, and stipulate that the alphabet of P is that of χP.

If $(t, R) \in \phi P$ then P is said to *refuse* R after 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 model this means the process behaves in a totally uncontrollable way. Such a semantical treatment is based on what is often referred to as 'catastrophic' divergence whereby the process in a diverging state is modelled as being able to engage in any action and generate any refusal.

CSP operators Although for our purposes neither the syntax nor the semantics of the whole standard CSP [7, 15] are needed — essential are only the parallel composition of processes, hiding of the communication over a set of channels and renaming of channels — we shall now recall additionally other operators used in the examples throughout this paper. Notice that in all the definitions, the direction of a channel (i.e., whether it is an input or output one) remains unchanged after application of the operator.

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 alphabet. Formally, the channels of $P\|Q$ are the union of those of P and Q, and

$$\delta(P\|Q) \stackrel{\text{df}}{=} \{t \circ u \mid (t{\lceil}\chi P, t{\lceil}\chi Q) \in (\tau P \times \delta Q) \cup (\delta P \times \tau Q) \wedge u \in (\alpha P \cup \alpha Q)^*\}$$
$$\phi(P\|Q) \stackrel{\text{df}}{=} \{(t, R \cup S) \mid (t{\lceil}\chi P, R) \in \phi P \wedge (t{\lceil}\chi Q, S) \in \phi Q\} \cup \delta(P\|Q) \times \mathbb{P}(\alpha(P\|Q))\ .$$

Parallel composition is commutative and associative; we shall use $P_1\| \cdots \|P_n$ to denote the parallel composition of processes $P_1, \ldots, P_n$.

Let P be a process and B be a set of channels of P; then $P\backslash B$ is a process that behaves like P with the actions occurring at the channels in B made invisible. Formally, the channels of $P\backslash B$ are those belonging to $\chi P - B$, and

$$\begin{aligned}\delta(P\backslash B) &\stackrel{\text{df}}{=} \{t\lceil\chi(P\backslash B) \circ u \mid u \in \alpha(P\backslash B)^* \wedge (t \in \delta P \vee \\ &\qquad\qquad \exists a_1, a_2, \ldots \in \alpha B\ \forall n \geq 1 : t \circ \langle a_1, \ldots, a_n\rangle \in \tau P)\} \\ \phi(P\backslash B) &\stackrel{\text{df}}{=} \{(t\lceil\chi(P\backslash B), R) \mid (t, R \cup \alpha B) \in \phi P\} \cup \delta(P\backslash B) \times \mathbb{P}(\alpha(P\backslash B))\ .\end{aligned}$$

Hiding is associative in that $(P\backslash B)\backslash B' = P\backslash(B \cup B')$. A crucial property involving the parallel composition and hiding operators states that if P and Q are two processes and $B \subseteq \chi P - \chi Q$ then $(P\backslash B)\|Q = (P\|Q)\backslash B$. Such a property is needed in order to treat process networks in a compositional way.

Let P be a process and $b{\in}\chi P$, $b'{\notin}\chi P$ channels such that $\mu b = \mu b'$. Then $P[b'/b]$ is a process that behaves like P except that each action $b{:}v$ by P is replaced by $b'{:}v$. The channels of $P[b'/b]$ are those of P with b replaced by b', and

$$\begin{aligned}\delta P[b'/b] &\stackrel{\text{df}}{=} \{t[b'/b] \mid t \in \delta P\} \\ \phi P[b'/b] &\stackrel{\text{df}}{=} \{(t[b'/b], R[b'/b]) \mid (t, R) \in \phi P\}\ ,\end{aligned}$$

where $R[b'/b]$ is obtained from R by replacing each action $b{:}v$ by $b'{:}v$.

In the next two operations on processes it is assumed that P and Q have the same channels. Then the deterministic choice $(P[]Q)$ and non-deterministic choice $(P \sqcap Q)$ are processes with the same channels as P and Q, the divergences being the union of those of P and Q, and the failures given respectively by

$$\begin{aligned}\phi(P[]Q) &\stackrel{\text{df}}{=} \{(\langle\,\rangle, R) \mid (\langle\,\rangle, R) \in \phi P \cap \phi Q\} \cup \{(t, R) \mid t{\neq}\langle\,\rangle \wedge (t, R){\in}\phi P \cup \phi Q\} \\ \phi(P \sqcap Q) &\stackrel{\text{df}}{=} \phi P \cup \phi Q\ .\end{aligned}$$

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

$$\begin{aligned}\delta(a \to P) &\stackrel{\text{df}}{=} \{\langle a\rangle \circ t \mid t \in \delta P\} \\ \phi(a \to P) &\stackrel{\text{df}}{=} \{(\langle a\rangle \circ t, R) \mid (t, R) \in \phi P\} \cup \{\langle\,\rangle\} \times \mathbb{P}(\alpha P - \{a\})\ .\end{aligned}$$

We also use $\textsc{Stop}_B$, or simply $\textsc{Stop}$ if B is clear from the context, to denote a deadlocked process with channel set B.

In examples, we use simple (mutually) recursive process definitions of the form $P \stackrel{\text{df}}{=} E$, where E is an expression built using the prefix, deterministic and non-deterministic choice, and $\textsc{Stop}$ constructs. For example, $P \stackrel{\text{df}}{=} (a \to P)[](b \to \textsc{Stop})$ defines a process which can execute action a any number of times, and then perhaps execute b and terminate. It is beyond the scope of this paper to give a precise treatment of recursive processes, and the reader is referred to, e.g., [15] for a full account of this aspect of CSP.

Networks of processes Processes $P_1, \ldots, P_n$ form a *network* if no channel is shared by more than two P_i's. We 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 channels shared by two different processes in the network. Network composition is commutative and associative. As a result, a network can be obtained by first composing some of the processes into a subnetwork, and then composing the result with the remaining processes. Moreover, the order in which processes are composed does not matter.

The definition of process network used in this paper assumes a one-to-one interprocess communication at the specification level. However, more general base process networks, such as those based on multicasting, may be modelled, e.g. by explicitly adding processes replicating messages sent, and forwarding them to several processes.

We can partition the channels of a process P into the input channels, $in\, P$, and output channels, $out\, P$. It is assumed that no two processes in a network have a common input channel or a common output channel. In the diagrams, an outgoing arrow indicates an output channel, and an incoming arrow indicates an input channel (being an input or output channel of a process is, in general, a purely syntactic notion).

Base processes A channel c of a process P is *value independent* if, for all $(t, R) \in \phi P$, $\alpha c \cap R \neq \emptyset$ implies $(t, R \cup \alpha c) \in \phi P$. We then define an *input-output process* (*IO* process) to be a non-diverging process P such that all its input channels are value independent. Intuitively, in an *IO* process the data component of a message arriving on an input channel c is irrelevant as far as its receiving is concerned; if one such message can be refused then so can any other message. In practice, this is not a restrictive property and, in particular, the 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, i.e., $\delta P = \emptyset$, 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).

2 Extraction patterns

We first explain the basic mechanism behind our modelling of behaviour abstraction, viz. the extraction pattern, using a simple example.

Consider a pair of base processes, DBL and BUF, shown in figure 1. DBL receives a signal (0 or 1) on its input channel at the very beginning of its execution, forwards this signal followed by its converse on its output channel and terminates. BUF is a buffer of capacity one, forever forwarding signals received on its input channel. In terms of CSP, we can represent them as

$$\textsc{Dbl} \stackrel{\text{df}}{=} []_{i \in \{0,1\}}\ c{:}i \rightarrow d{:}i \rightarrow d{:}(1{-}i) \rightarrow \textsc{Stop}$$

$$\textsc{Buf} \stackrel{\text{df}}{=} []_{i \in \{0,1\}}\ d{:}i \rightarrow e{:}i \rightarrow \textsc{Buf}\ ,$$

and one can see that $\textsc{Dbl} \otimes \textsc{Buf}$ is semantically equal to $\textsc{Dbl}[e/d]$, i.e., the composition of the two processes behaves like Dbl with its output channel renamed to e.

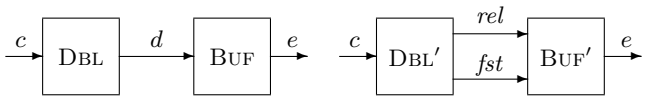

Fig. 1. Two base processes and their implementations.

Suppose now that the signal transmission between the two processes has been implemented using two channels, *rel* and *fst*, as shown in figure 1. The transmissions on d are now duplicated and the two copies sent along *rel* (reliable but slow) and *fst* (fast but unreliable). That is, $\textsc{Dbl}'$ sends a duplicated signal, while $\textsc{Buf}'$ accepts the first received copy of the signal and passes it on ignoring the other one:

$$\textsc{Dbl}' \stackrel{\text{df}}{=} []_{i\in\{0,1\}} c{:}i \to (\mathit{rel}{:}i \to \mathit{rel}{:}(1-i) \to \textsc{Stop} \| \mathit{fst}{:}i \to \mathit{fst}{:}(1-i) \to \textsc{Stop})$$

and $\textsc{Buf}' \stackrel{\text{df}}{=} \textsc{Buf}^{\langle\rangle}$, where:

$$\begin{aligned}
\textsc{Buf}^{\langle\rangle} &\stackrel{\text{df}}{=} []_{i\in\{0,1\}}\ \mathit{rel}{:}i \to e{:}i \to \textsc{Buf}^{\langle \mathit{fst}:i\rangle} \ [] \\
&\qquad []_{i\in\{0,1\}}\ \mathit{fst}{:}i \to e{:}i \to \textsc{Buf}^{\langle \mathit{rel}:i\rangle} \\
\textsc{Buf}^{\langle \mathit{fst}:k\rangle \circ z} &\stackrel{\text{df}}{=} []_{i\in\{0,1\}}\ \mathit{rel}{:}i \to e{:}i \to \textsc{Buf}^{\langle \mathit{fst}:k\rangle \circ z \circ \langle \mathit{fst}:i\rangle} [] \ \mathit{fst}{:}k \to \textsc{Buf}^{z} \\
\textsc{Buf}^{\langle \mathit{rel}:k\rangle \circ z} &\stackrel{\text{df}}{=} []_{i\in\{0,1\}}\ \mathit{fst}{:}i \to e{:}i \to \textsc{Buf}^{\langle \mathit{rel}:k\rangle \circ z \circ \langle \mathit{rel}:i\rangle} [] \ \mathit{rel}{:}k \to \textsc{Buf}^{z}\ .
\end{aligned}$$

In the above, the traces appearing in the superscripts indicate which messages should be ignored by $\textsc{Buf}'$, as their copies have already been received.

Such a scheme clearly works as we have $\textsc{Dbl} \otimes \textsc{Buf} = \textsc{Dbl}' \otimes \textsc{Buf}'$. Suppose next that the transmission of signals is imperfect and two types of faulty behaviour can occur: $\textsc{Dbl}_1 \stackrel{\text{df}}{=} \textsc{Dbl}' \sqcap \textsc{Stop}$ and

$$\textsc{Dbl}_2 \stackrel{\text{df}}{=} \textsc{Dbl}' \sqcap []_{i\in\{0,1\}} c{:}i \to \mathit{rel}{:}i \to \mathit{rel}{:}(1-i) \to \textsc{Stop}\ .$$

In other words, $\textsc{Dbl}_1$ can break down completely, refusing to output any signals, while $\textsc{Dbl}_2$ can fail in such a way that although channel *fst* is completely blocked, *rel* can still transmit the signals ($\textsc{Dbl}_2$ could be used to model the following situation: in order to improve performance, a 'slow' channel d is replaced by two channels, a potentially *fast* yet unreliable channel *fst*, and a slower but *reliable* backup channel *rel*). Since $\textsc{Dbl} \otimes \textsc{Buf} = \textsc{Dbl}_2 \otimes \textsc{Buf}'$ and $\textsc{Dbl} \otimes \textsc{Buf} \neq \textsc{Dbl}_1 \otimes \textsc{Buf}'$, it follows that $\textsc{Dbl}_2$ is much 'better' an implementation of the Dbl process than $\textsc{Dbl}_1$. We will now analyse the differences between the behavioural properties of the two processes and, at the same time, introduce informally some basic concepts used subsequently.

We start by observing that the output of $\textsc{Dbl}_2$ can be thought of as adhering to the following two rules: (R1) the transmissions over *rel* and *fst* are consistent w.r.t. message content (the set of all traces over *rel* and *fst* satisfying such a property will be denoted by *Dom*); and (R2) transmission over *rel* is reliable; there is no such guarantee for *fst*. The output produced by $\textsc{Dbl}_1$ satisfies R1, but fails to satisfy R2, unlike $\textsc{Dbl}_2$ which satisfies R1-2. To express this difference formally, we need to render these two conditions in some form of precise notation.

To capture the relationship between traces of Dbl and $\textsc{Dbl}_2$, we will employ an (extraction) mapping *extr*, which for a trace over *rel* and *fst* returns the corresponding trace over *d*. For example, keeping in mind that duplicates of signals should be ignored by the receiving process, some extraction mappings will be:

$$\begin{aligned} \langle\,\rangle &\mapsto \langle\,\rangle \\ \langle \mathit{rel}{:}0\rangle &\mapsto \langle \mathit{d}{:}0\rangle \\ \langle \mathit{fst}{:}0\rangle &\mapsto \langle \mathit{d}{:}0\rangle \\ \langle \mathit{fst}{:}1, \mathit{rel}{:}1\rangle &\mapsto \langle \mathit{d}{:}1\rangle \\ \langle \mathit{fst}{:}1, \mathit{rel}{:}1, \mathit{fst}{:}0\rangle &\mapsto \langle \mathit{d}{:}1, \mathit{d}{:}0\rangle\ . \end{aligned}$$

Notice that the extraction mapping need only be defined for traces satisfying R1, i.e., those in *Dom*. We further observe that, in view of R2, some of the traces in *Dom* may be regarded as *incomplete*. For example, $\langle \mathit{fst}{:}1, \mathit{rel}{:}1, \mathit{fst}{:}0\rangle$ is such a trace since channel *rel* is reliable and so the duplicate of $\mathit{fst}{:}0$ (i.e., $\mathit{rel}{:}0$) is bound to be eventually offered for transmission. The set of all other traces in *Dom* — i.e., those which in principle may be *complete* — will be denoted by *dom* (in general, $\mathit{Dom} = \mathit{Pref}(\mathit{dom})$, meaning that each interpretable trace has, at least in theory, a chance of being completed). For our example, *dom* will contain all traces in *Dom* where the transmission on *fst* has not overtaken that on *rel*. (As another example, suppose that the whole sequence of actions $a_1, \ldots, a_k$ is extracted to a single action a, i.e., $\langle a_1, \ldots, a_i\rangle \mapsto \langle\rangle$, for $i < k$, and $\langle a_1, \ldots, a_k\rangle \mapsto \langle a\rangle$; then we do not consider such a transmission complete unless the whole sequence $a_1, \ldots, a_k$ has been transmitted.)

Although it will play a central role, the extraction mapping alone is not sufficient to identify the 'correct' implementation of Dbl in the presence of faults since $\tau\textsc{Dbl} = \mathit{extr}(\tau\textsc{Dbl}_1) = \mathit{extr}(\tau\textsc{Dbl}_2)$, while $\textsc{Dbl}_1$ is incorrect. What one also needs is an ability to relate the refusals of potential implementations $\textsc{Dbl}_1$ and $\textsc{Dbl}_2$ with the possible refusals of the base process Dbl. This, however, is much harder than relating traces. For suppose that we attempted to 'extract' the refusals of $\textsc{Dbl}_2$ using the mapping *extr*. Then, we would have had $(\langle \mathit{c}{:}0\rangle, \{\mathit{fst}{:}0\}) \in \phi\textsc{Dbl}_2$, while $\mathit{extr}(\langle \mathit{c}{:}0\rangle, \{\mathit{fst}{:}0\}) = (\langle \mathit{c}{:}0\rangle, \{\mathit{d}{:}0\}) \notin \phi\textsc{Dbl}$. This indicates that the crude extraction of refusals is not going to work. What we need is a more sophisticated device, which in our case comes in the form of another mapping, *ref*, constraining the possible refusals a process can exhibit on channels in the implementation, after a given trace $t \in \mathit{Dom}$. For example, we should not allow the refusal of $\mathit{rel}{:}0$, after an incomplete communication $t = \langle \mathit{fst}{:}0\rangle$.

The refusal bounds given by *ref* may be thought of as ensuring a kind of liveness or progress condition on sets of channels upon which composition will occur

when implementation components are composed to build the full implementation system Q_{net}. Since these channels are to be composed upon and so hidden, the progress enforced manifests itself in the final system as the occurrence of a τ (invisible) transition, which leads to the instability of the states in which those τ transitions are enabled. This then means that these states will not contribute a failure of Q_{net}. Conversely, if we may not enforce progress after a *complete* behaviour, then it is possible that the relevant state reached *will* contribute to a failure, (t, R), of Q_{net}. Since, in the failures model of CSP, if Q_{net} 'implements' P_{net}, $\phi Q_{net} \subseteq \phi P_{net}$, then we must ensure that the relevant failure, (t, R), also occurs in P_{net}. We do this by ensuring that progress will not be possible on the corresponding channel in the specification component. Here, lack of progress on internal channels leads to stability and the fact that the relevant state *will* give rise to a failure of P_{net}.

Therefore, a sender implementation process, like $\textsc{Dbl}_2$, can admit a refusal disallowed by $ref(t)$ if the target process, Dbl, admits after the extracted trace $extr(t)$ the refusal of all communication on the corresponding channel and, moreover, the trace t itself is complete, i.e., $t \in dom$.

Finally, it should be stressed that $ref(t)$ gives a refusal bound on the sender side (more precisely, the process which implements the sender target process). But this is enough since, if we want to rule out a deadlock in communication between the sender and receiver (on a localised set of channels), it is now possible to stipulate on the receiver side that no refusal is such that, when combined with any refusal allowed by $ref(t)$ on the sender side, it can yield the whole alphabet of the channels used for transmission.

Formal definition of extraction pattern The notion of extraction pattern (introduced and used in [8, 9], and slightly simplified here) relates behaviour on a set of channels in an implementation process to that on a channel 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.

Definition 1. *An* extraction pattern *is a tuple* $ep \stackrel{\text{df}}{=} (B, b, dom, extr, ref)$*, where:* $B \neq \emptyset$ *is a set of* source *channels, and* b *is a* target *channel;* $dom \neq \emptyset$ *is a set of traces over the sources;* $extr$ *is a monotonic mapping defined for traces* $t \in Dom \stackrel{\text{df}}{=} Pref(dom)$ *such that* $extr(t)$ *is a trace over the target and* $extr(\langle\,\rangle) = \langle\,\rangle$*; and* ref *is a mapping defined for traces* $t \in Dom$ *such that* $ref(t)$ *is a non-empty family of proper subsets of* αB *(it is assumed that* $R \in ref(t)$ *and* $R' \subset R$ *always implies* $R' \in ref(t)$*).*

As already mentioned, the mapping $extr$ interprets a trace over the source channels B (in the implementation process) in terms of a trace over a channel b (in the target process), and defines functionally correct (i.e., in terms of traces) behaviour over those source channels by way of its domain. The 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. dom contains those traces in Dom for which the communication over B may

be regarded as complete; the constraint on refusals given by *ref* is only allowed to be violated for such traces. The intuition behind this requirement is that we cannot regard as correct a situation where deadlock occurs in the implementation process when behaviour is incomplete, for regarding this as correct behaviour would imply that the specification process could in some sense deadlock while in the middle of executing a single (atomic) action. The extraction mapping *extr* is monotonic as receiving more information cannot decrease the current knowledge about the transmission. $\alpha B \notin ref(t)$ will be useful to forbid the sender to refuse all possible transmissions after an unfinished communication t.

The extraction pattern discussed informally for the example in figure 1 can be formalised as ep_0, where $B \stackrel{\text{df}}{=} \{rel, fst\}$ and $b \stackrel{\text{df}}{=} d$. To define the remaining components, for a trace t over B and a channel $x \in B$, we denote by t_x the trace obtained by first projecting t onto x and then renaming the channel x to d, i.e., $t_x \stackrel{\text{df}}{=} (t{\restriction}x)[d/x]$. Then *dom* is the set of all traces t over B such that $t_{fst} \leq t_{rel}$, and so *Dom* is the set of all traces t over B such that $t_{fst} \leq t_{rel}$ or $t_{rel} \leq t_{fst}$. Moreover, for every trace t in *Dom*, $extr(t)$ is the longest of the traces t_{fst} and t_{rel}, and $ref(t)$ is the set of all sets $R \subseteq \alpha B$ such that $rel{:}0 \notin R$ or $rel{:}1 \notin R$.

Intuitively, the extraction mapping always returns a trace derived from the longer of the two communications over *fst* and *rel* (this is acceptable since these communications are consistent, see the definition of *Dom*). Complete traces are those where *rel* has not fallen back behind *fst* in transmitting the signals. The $ref(t)$ component states that if behaviour is not complete on *rel* and *fst*, then at least one action must be possible on *rel*.

To relate incoming communication of DBL and DBL_2, we will need another kind of extraction pattern. An *identity* extraction pattern for a channel c, id_c, is one for which $B \stackrel{\text{df}}{=} \{c\}$, $b \stackrel{\text{df}}{=} c$, $dom = Dom$ is the set of all traces over channel c, $extr(t) \stackrel{\text{df}}{=} t$ and $ref(t)$ is the set of all proper subsets of αc. The idea here is that the extraction mapping is simply the identity mapping, i.e., the specification and implementation processes have the same input interface. Any such communication can therefore be a terminated one, i.e., it can be regarded as complete.

We lift two of the notions introduced above to any set of extraction patterns $Ep = \{ep_1, \ldots, ep_n\}$, where $ep_i = (B_i, b_i, dom_i, extr_i, ref_i)$. Dom_{Ep} is the set of all traces t over channels $B_1 \cup \ldots \cup B_n$ such that $t{\restriction}B_i \in Dom_i$, for every $i \leq n$. Moreover, $extr_{Ep}(\langle\,\rangle) \stackrel{\text{df}}{=} \langle\,\rangle$ and, for every $t \circ \langle a \rangle \in Dom_{Ep}$ with $a \in \alpha B_i$, $extr_{Ep}(t \circ \langle a \rangle) \stackrel{\text{df}}{=} extr_{Ep}(t) \circ u$, where (possibly empty) u is such that

$$extr_i(t{\restriction}B_i \circ \langle a \rangle) = extr_i(t{\restriction}B_i) \circ u\ .$$

In what follows, different extraction patterns will have disjoint sources and distinct targets.

3 The implementation relation

Suppose that we intend to implement a base process P using another process Q with a possibly different communication interface. The correctness of the

implementation will be expressed in terms of two sets of extraction patterns, *In* and *Out*. 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.

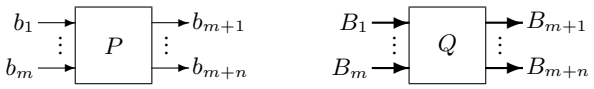

Fig. 2. Base process P and its implementation Q.

Let P be a base *IO* process as in figure 2 and, for every $i \leq m+n$, let $ep_i \stackrel{\text{df}}{=} (B_i, b_i, dom_i, extr_i, ref_i)$ be an extraction pattern. We will denote by *In* the set of the first m extraction patterns ep_i, and by *Out* the remaining n extraction patterns. We then take any non-diverging process Q with the input channels $B_1 \cup \ldots \cup B_m$ and output channels $B_{m+1} \cup \ldots \cup B_{m+n}$, as shown in figure 2, where thick arrows represent *sets* of channels. We will further say that channels B_i are *blocked* at a failure $(t, R) \in \phi Q$ if either $i \leq m$ and $\alpha B_i - R \in ref_i(t{\restriction}B_i)$, or $i > m$ and $\alpha B_i \cap R \notin ref_i(t{\restriction}B_i)$. (Note that in both cases this signifies that the refusal bound imposed by ref_i has been breached.) We denote this by $i \in Blocked(t, R)$.

Definition 2. *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, where $All \stackrel{\text{df}}{=} In \cup Out$.*[1]

1. $\tau Q \subseteq Dom_{All}$ *and* $extr_{All}(\tau Q) \subseteq \tau P$.
2. *If* $t \leq t' \leq \ldots$ *are unboundedly growing traces of Q, then the sequence of traces* $extr_{All}(t) \leq extr_{All}(t') \leq \ldots$ *also grows unboundedly.*
3. *If* $(t, R) \in \phi Q$ *and* $i \in Blocked(t, R)$, *then* $t{\restriction}B_i \in dom_i$.
4. *If* $(t, R) \in \phi Q$ *is such that* $t{\restriction}B_i \in dom_i$ *for all* $i \leq m+n$, *then*

$$\Big(\; extr_{All}(t) \; , \bigcup_{i \in Blocked(t,R)} \alpha b_i \; \Big) \in \phi P \; .$$

In the above definition, (1) states that all traces of Q can be interpreted as traces of P. According to (2), it is not possible to execute Q indefinitely without extracting any actions of P. According to (3), if refusals grow in excess of their bounds on a source channel set B_i, communication on B_i may be interpreted as locally completed. Finally, (4) states a condition for refusal extraction, whereby

[1] In our previous work (see [3, 4, 8, 9]), the condition $\tau Q \subseteq Dom_{All}$ is replaced by a weaker *rely/guarantee* property (in the sense of [5]), which states that if a trace of Q projected on the input channels can be interpreted by *In*, then it must be possible to interpret its projection on the output channels by *Out*.

if a trace is locally completed on all channels, then any local blocking on a source channel set B_i in Q is transformed into the refusal of the whole αb_i in P.

A direct comparison of an implementation process Q with the corresponding base process P is only possible if there is no difference in the communication interfaces. This corresponds to the situation that, in the definition of $\preceq_{Out}^{In}$, both *In* and *Out* are sets of *identity* extraction patterns. In such a case, we simply denote $Q \preceq P$ and then we can directly compare the semantics of the two processes in question.

Theorem 1. *Let Q be a divergence-free process with the same set of input channels and the same set of output channels as an IO process P. Then $Q \preceq P$ if and only if for every $(t, R) \in \phi Q$,*

$$\Big(t,\ (\alpha in\, P \cap R)\ \cup \bigcup_{b \in out\, Q,\, \alpha b \subseteq R} \alpha b \Big) \in \phi P\ .$$

Proof. For identity extraction patterns, the condition $i \in Blocked(t, R)$ can be reduced to $\alpha b_i \cap R \neq \emptyset$ for an input channel b_i, and to $\alpha b_i \subseteq R$ for an output channel b_i. Then the proof follows directly from definition 2, the definition of an identity extraction pattern, and the fact that P is an *IO* process. □

That is, $Q \preceq P$ implies that Q is a process whose functionality in terms of traces conforms to that of the specification process P (to see this, it suffices to take $R = \emptyset$). Moreover, all the *essential* refusals of Q are also present in P. That is, all the refusals on input channels are preserved *entirely*, while for output channels any refusal to output anything on a given channel b 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 explicitly allows no output at all to be produced. We therefore consider that the above result embodies a fully adequate notion of *realisability* in any practical framework consistent with our setup. It is also worth mentioning that $\preceq$ is a preorder (i.e., it is a reflexive and transitive relation), and is preserved by the hiding of communication channels.

Proposition 1. *If $Q \preceq P$ and B is a set of channels of P such that $P\backslash B$ is divergence-free, then $Q\backslash B \preceq P\backslash B$.*

Proof. By theorem 1, $\tau Q \subseteq \tau P$. Hence, since Q is divergence-free and $P\backslash B$ is divergence-free, $Q\backslash B$ is also divergence-free.

Suppose that $(t, R) \in \phi(Q\backslash B)$ and that B' is the set of all $b \in out\,(P\backslash B) = out\,(Q\backslash B)$ such that $\alpha b \subseteq R$. Then, since $Q\backslash B$ is divergence-free, there is $(w, R \cup \alpha B) \in \phi Q$ such that $w{\restriction}\chi(Q\backslash B) = t$. By $Q \preceq P$ and theorem 1, $(w, (\alpha in\, P \cap R) \cup \alpha B' \cup \alpha B) \in \phi P$. Thus $(t, (\alpha in\, P \cap R) \cup \alpha B') \in \phi(P\backslash B)$, and so $Q\backslash B \preceq P\backslash B$, by theorem 1. □

In the light of theorem 1, relation $\preceq$ provides us with a direct measure of the closeness of the approximation of the base process P by an implementation

process Q. It therefore deserves further discussion, in particular with regard to its relationship with the standard *refinement* ordering of CSP, denoted by $\sqsupseteq$. In our framework, $Q \sqsupseteq P$ (i.e., Q 'CSP implements or *refines*' P) basically amounts to stating that $\phi Q \subseteq \phi P$.[2]

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* processes. A process P is said to be output-determined if, for any traces $t \circ \langle c{:}v \rangle$ and $t \circ \langle c{:}v' \rangle$ of P such that c is an output channel, it is the case that $v = v'$ (i.e., the result produced by P on a given output channel c is determined at any given point of its execution).

Theorem 2. *If P is output-determined, then $Q \preceq P$ implies $Q \sqsupseteq P$.*

Proof. Since, by definition, Q is divergence-free, we need to show that the failures of Q are included in those of P. Suppose that $(t, R) \in \phi Q$ and that B' is the set of all $b \in \mathit{out}\, P$ such that $\alpha b \subseteq R$. Moreover, assume that R is a maximal refusal set after t, i.e., $(t, R') \in \phi Q$ and $R \subseteq R'$ implies $R = R'$.

Suppose now that $(t, R) \notin \phi P$. By theorem 1, $(t, R'') \in \phi P$, where $R'' = (\alpha \mathit{in}\, P \cap R) \cup \alpha B'$. Since $(t, R'') \in \phi P$ and $(t, R) \notin \phi P$, there exists $c{:}v \in R - R''$ such that $t \circ \langle c{:}v \rangle \in \tau P$. Clearly, $c \in \mathit{out}\, P$ and $c \notin B'$. Hence, since P is output-determined, $t \circ \langle c{:}v' \rangle \notin \tau P$, for all $v' \in \mu c - \{v\}$. Thus, since $\tau Q \subseteq \tau P$, we also have $t \circ \langle c{:}v' \rangle \notin \tau Q$, for all $v' \in \mu c - \{v\}$. Consequently, since R is a maximal refusal set after t, we have $c{:}v' \in R$, for all $v' \in \mu c - \{v\}$. Together with $c{:}v \in R$ this means that $c \in B'$, a contradiction. □

We will now investigate what can be established by considering the way P and Q interact with a possible environment. Let P be a (specification) base *IO* process, which is not assumed to be output-determined, and Q be its implementation w.r.t. suitable identity extraction pattern, i.e., let $Q \preceq P$. Therefore Q can be used in place of P in an environment T accepting all their outputs, as shown in figure 3. Our aim now is to relate the behaviour of Q and P in the environment provided by the process T.

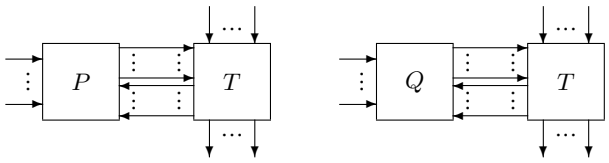

Fig. 3. Relating base and implementation processes in the context of an environment.

We assume that, besides P, also T is an *IO* process, their composition is non-diverging, and $\mathit{out}\, P \subseteq \mathit{in}\, T$. We then obtain

[2] In general, one would also require that $\delta Q \subseteq \delta P$, which in our case always holds as Q is divergence-free.

Theorem 3 ([4]). *If $Q \preceq P$ then $Q \otimes T \sqsupseteq P \otimes T$.*

Thus $Q \otimes T$ is at least as *deterministic* a process as $P \otimes T$ in the sense of CSP (see [7, 15]). This makes Q at least as good as P (and possibly much better) as a process to be used in practice. Hence we conclude that $Q \preceq P$ captures an adequate notion of realisability in the context of an environment T.

We finally present a fundamental result, that the implementation relation is compositional.

Theorem 4 ([4]). *Let K and L be two base IO processes whose composition is non-diverging, as in figure 4, 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*

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

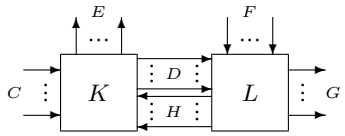

Fig. 4. Base processes used in the formulation of the compositionality theorem.

Hence the implementation relation is preserved through network composition, and the only restriction is that the network of the base processes should be designed in a divergence-free way. However, the latter is a standard requirement in the CSP approach (recall again that divergences are regarded as totally unacceptable).

Returning to the example in figure 1, it can be shown that $\textsc{Dbl}_2 \preceq^{id_c}_{ep_0} \textsc{Dbl}$ and $\textsc{Buf}' \preceq^{ep_0}_{id_e} \textsc{Buf}$. Hence, it follows from theorem 4 and $\textsc{Dbl} \otimes \textsc{Buf} = \textsc{Dbl}[e/d]$ that $\textsc{Dbl}_2 \otimes \textsc{Buf}' \preceq \textsc{Dbl}[e/d]$. Thus, by theorem 2, $\textsc{Dbl}_2 \otimes \textsc{Buf}' \sqsupseteq \textsc{Dbl}[e/d]$, as process $\textsc{Dbl}[e/d]$ is easily seen to be output-determined. But we can go one step further since $\textsc{Dbl}[e/d]$ is a *deterministic* process in the sense of CSP (see [7, 15]). This means, in particular, that $Q \sqsupseteq \textsc{Dbl}[e/d]$ implies $Q = \textsc{Dbl}[e/d]$, for any process Q. Thus, we can finally conclude that $\textsc{Dbl}_2 \otimes \textsc{Buf}' = \textsc{Dbl}[e/d]$ and that we obtained such a result by purely compositional argument, using the results presented in this section together with a well-known property of deterministic CSP processes.

4 Compositionality and CA actions

The above approach to verification, based on abstraction of interface difference and compositionality, will now be applied to an analysis of Coordinated Atomic

(CA) actions [12, 13]. This concept represents an approach to structuring complex activities in a distributed environment, aimed at supporting fault tolerance in object-oriented systems.

The following are some of the essential characteristics of the model: (CA1) a CA action has roles which are activated by some external participants (processes or threads); (CA2) a CA action starts when all the roles have been activated and finishes when each role has completed its execution; (CA3) the execution of a CA action updates the system state (represented by a set of external objects) atomically; (CA4) roles can access local objects as well as participate in nested CA actions; and (CA5) CA actions provide a basic framework for exception handling that can support a variety of fault tolerance mechanisms.

We will interpret CA3 as stating that the desired behaviour of external objects constitutes a specification of the system design based on CA actions, and so the CA actions design should respect the main objective which is that the external objects must complete successfully all the stages through which they are passing while being manipulated by the CA actions.

Modelling the production cell In the rest of this section we will discuss an example inspired by the 'production cell' case study in [13]. The initial architecture of the design is shown in figure 5, where MAN is a *manager* process initially holding n external objects, represented throughout as $\xi^1, \ldots, \xi^n$, which are supposed to pass through the required stages of processing (three stages, in our case). The external objects are passed as messages and every object must be received by processes ST_1, ST_2 and ST_3, representing the three stages. To keep track of progress, the three processes report to the manager after each stage has been successfully completed. The manager MAN, in turn, sends out, over the channels res_i, messages that inform the external environment how the system as a whole is progressing. The initial system specification is given as $\mathrm{SYS} \stackrel{\mathrm{df}}{=} \mathrm{MAN} \otimes \mathrm{ST}_1 \otimes \mathrm{ST}_2 \otimes \mathrm{ST}_3$, where:

$$
\begin{aligned}
\mathrm{MAN} &\stackrel{\mathrm{df}}{=} \mathrm{MAN}^0 \\
\mathrm{ST}_1 &\stackrel{\mathrm{df}}{=} []_{i \le n}\ a{:}\xi^i \to b{:}i \to c{:}\xi^i \to \mathrm{ST}_1 \\
\mathrm{ST}_2 &\stackrel{\mathrm{df}}{=} []_{i \le n}\ c{:}\xi^i \to d{:}i \to e{:}\xi^i \to \mathrm{ST}_2 \\
\mathrm{ST}_3 &\stackrel{\mathrm{df}}{=} []_{i \le n}\ e{:}\xi^i \to f{:}i \to \mathrm{ST}_3
\end{aligned}
$$

$$
\mathrm{MAN}^k \stackrel{\mathrm{df}}{=} \begin{cases} a{:}\xi^{k+1} \to \mathrm{MAN}^{k+1} [] \\ []_{i \le n,\ ch \in \{b,d,f\}}\ ch{:}i \to res_i{:}ch \to \mathrm{MAN}^k \text{ if } k < n \\ \\ []_{i \le n,\ ch \in \{b,d,f\}}\ ch{:}i \to res_i{:}ch \to \mathrm{MAN}^k \text{ if } k = n\ . \end{cases}
$$

Note that the meaning of, e.g., message $res_i{:}f$ is to inform the external environment that ξ^i has successfully passed through the third stage of processing. It is not difficult to see that each of the processes used to define SYS is a base *IO* process, and so is SYS.

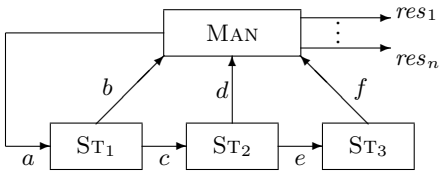

Fig. 5. Initial architecture of the system based on a manager process, and three processes responsible for the three stages of processing.

As far as the overall correctness is concerned, we are interested in establishing that every object ξ^i has successfully passed through each stage, and has done so in the prescribed order. Such a property may be verified by showing that, when restricted to a single output channel res_i, the system behaves as the process $\mathrm{SPEC}_i \stackrel{\mathrm{df}}{=} res_i{:}b \rightarrow res_i{:}d \rightarrow res_i{:}f \rightarrow \mathrm{STOP}$. Indeed, one can show (e.g., using the tool FDR [15]), that $\mathrm{SYS}@res_i = \mathrm{SPEC}_i$, for every $i \leq n$, where

$$P@res_i \stackrel{\mathrm{df}}{=} P\backslash\{res_1, \ldots, res_{i-1}, res_{i+1}, \ldots, res_n\} \ ,$$

for any process P with the same channels as SYS.

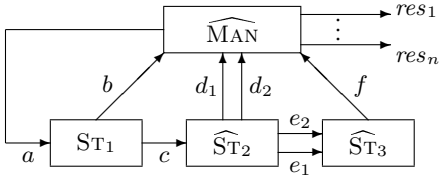

Fig. 6. A modified design, with different communication interfaces between two pairs of processes.

Introducing exception handling As stated by CA5, CA actions provide a basic framework for exception handling. At the level where the roles of CA actions are described, exceptions could be modelled as actions which are followed by exception handling processes. At the current, higher level of abstraction, we will model the effects of exceptions rather than the exceptions themselves. In particular, we can imagine the situation when an exception raised within some CA action, representing an implementation of one of the three stages, is handled by invoking another, alternative CA action. Suppose this can occur during the second stage. To capture this, we replace ST_2 by another process, $\widehat{\mathrm{ST}_2}$, which allows two alternative ways of processing a recently received object ξ^i: it is passed

to the next stage using exactly one of the channels, e_1 or e_2, which has replaced e, and the manager is informed about the successful completion using the corresponding new channel, d_1 or d_2, which has replaced d. The process $\widehat{\text{ST}_2}$ uses a non-deterministic choice operator to choose between the two ways of processing an object, and so that process rather than the environment is responsible for resolving this choice: this models the possibility of a fault occurring. The new system is given by $\widehat{\text{SYS}} \stackrel{\text{df}}{=} \widehat{\text{MAN}} \otimes \text{ST}_1 \otimes \widehat{\text{ST}_2} \otimes \widehat{\text{ST}_3}$, where (see also figure 6):

$$\widehat{\text{MAN}} \stackrel{\text{df}}{=} \widehat{\text{MAN}}^0$$

$$\widehat{\text{ST}_2} \stackrel{\text{df}}{=} [\!]_{i \le n}\ c{:}\xi^i \to (d_1{:}i \to e_1{:}\xi^i \to \widehat{\text{ST}_2} \sqcap d_2{:}i \to e_2{:}\xi^i \to \widehat{\text{ST}_2})$$

$$\widehat{\text{ST}_3} \stackrel{\text{df}}{=} [\!]_{i \le n,\, ch \in \{e_1, e_2\}}\ ch{:}\xi^i \to f{:}i \to \widehat{\text{ST}_3}$$

$$\widehat{\text{MAN}}^k \stackrel{\text{df}}{=} \begin{cases} (a{:}\xi^{k+1} \to \widehat{\text{MAN}}^{k+1}) [\!] Z^k [\!] W^k & \text{if } k < n \\ Z^k [\!] W^k & \text{if } k = n \end{cases}$$

$$Z^k \stackrel{\text{df}}{=} [\!]_{i \le n,\, ch \in \{b, f\}}\ ch{:}i \to res_i{:}ch \to \widehat{\text{MAN}}^k$$

$$W^k \stackrel{\text{df}}{=} [\!]_{i \le n,\, ch \in \{d_1, d_2\}}\ ch{:}i \to res_i{:}d \to \widehat{\text{MAN}}^k\ .$$

To verify the correctness of $\widehat{\text{SYS}}$ compositionally, we can use the model presented earlier on in this paper. To establish that $\widehat{\text{MAN}}$, $\widehat{\text{ST}_2}$ and $\widehat{\text{ST}_3}$ are respectively implementations of MAN, ST_2 and ST_3, we will use two extraction patterns, both instances of a generic *merge* extraction pattern, mrg_x, for $x \in \{d, e\}$. The sources of mrg_x are two channels, x_1 and x_2, and the target is a channel x such that $\mu x_1 = \mu x_2 = \mu x$. The valid traces are all traces over $\alpha x_1 \cup \alpha x_2$ and $Dom = dom$. The extraction mapping is a trace homomorphism such that $extr(x_i{:}v) \stackrel{\text{df}}{=} \langle x{:}v \rangle$, for $i = 1, 2$. The *ref* mapping is such that, for every trace $t \in Dom$, $ref(t)$ comprises all proper subsets of $\alpha x_1 \cup \alpha x_2$. One can then show (e.g., using the techniques proposed in [3]) that

$$\widehat{\text{MAN}} \preceq^{mrg_d, id_b, id_f}_{id_a, id_{res_1}, \ldots, id_{res_n}} \text{MAN}\ ,\quad \widehat{\text{ST}_2} \preceq^{id_c}_{mrg_d, mrg_e} \text{ST}_2 \ \text{ and } \ \widehat{\text{ST}_3} \preceq^{mrg_e}_{id_f} \text{ST}_3\ .$$

Hence, by theorem 4, we obtain that $\widehat{\text{SYS}} \preceq \text{SYS}$ and so $\widehat{\text{SYS}}@res_i = \text{SPEC}_i$, for every $i \le n$, as the implementation relation is preserved by hiding of channels (see proposition 1), $\text{SYS}@res_i = \text{SPEC}_i$, and each process SPEC_i is output-determined (see theorem 2) and deterministic (in the CSP sense).

Refining a stage A possible implementation of $\widehat{\text{ST}_2}$ is to employ two processes, ACT and ACT′, which represent two intended CA actions: a primary CA action responsible for processing objects at the second stage, and another CA action invoked when the first one cannot be completed successfully (see figure 7). The implementation of $\widehat{\text{ST}_2}$ is $\widetilde{\text{ST}_2} \stackrel{\text{df}}{=} \text{ACT} \otimes \text{ACT}'$, where:

$$\text{ACT} \stackrel{\text{df}}{=} [\!]_{i \le n}\ c{:}\xi^i \to (d_1{:}i \to e_1{:}\xi^i \to \text{ACT} \sqcap h{:}\xi^i \to g{:}go \to \text{ACT})$$

$$\text{ACT}' \stackrel{\text{df}}{=} [\!]_{i \le n}\ h{:}\xi^i \to d_2{:}i \to e_2{:}\xi^i \to g{:}go \to \text{ACT}'$$

Since $\widehat{\text{ST}_2} = \widetilde{\text{ST}_2}$ (which can be checked using, e.g., the FDR tool [15]), we can now refine the architecture of the design, replacing $\widehat{\text{ST}_2}$ with $\widetilde{\text{ST}_2}$, and obtaining a new system composed of five processes: $\widetilde{\text{SYS}} \stackrel{\text{df}}{=} \widehat{\text{MAN}} \otimes \text{ST}_1 \otimes \text{ACT} \otimes \text{ACT}' \otimes \widehat{\text{ST}_3}$. As $\widetilde{\text{SYS}} = \widehat{\text{SYS}}$, we do not lose the correctness already established for $\widehat{\text{SYS}}$, i.e., $\widetilde{\text{SYS}}@res_i = \text{SPEC}_i$, for every $i \leq n$.

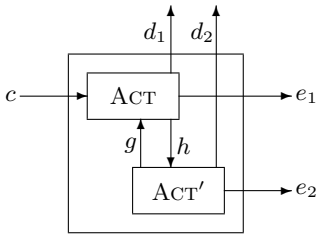

Fig. 7. An architecture for a distributed implementation of the second stage of processing.

Since all the components in $\widetilde{\text{SYS}}$ are *IO* processes, we can again apply the framework developed in this paper, and indeed continue the cycle of development until a sufficiently detailed level of modelling has been reached, at each stage applying a compositional argument as outlined above. We stop this discussion here, but instead look at the way in which ACT could be implemented by a process CA, shown in figure 8, modelling more closely the intended features of CA actions.

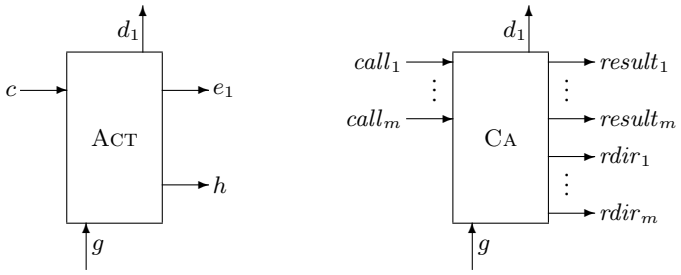

Fig. 8. ACT implemented as CA action process CA.

Modelling CA actions CA has $call \stackrel{\text{df}}{=} \{call_1, \ldots, call_m\}$ channels which are intended to carry messages with some input data (parameters) requesting an access to CA, each such message corresponding to starting a role (see requirement

CA1). CA waits for a set of m messages, each such message arriving along a different channel $call_i$, before accepting them for further processing (see CA2). In general, it is not the case that any message set will be accepted as a valid call. It is therefore assumed that there is a non-empty set of *valid inputs* (e.g., two out of three results are successful in majority voting, see [8]), *Valid*, and a *consistent input* is any set of messages which is contained in some valid input. Each $V \in$ *Valid* is a set of m messages, $\{call_1{:}v'_1, \ldots, call_m{:}v'_m\}$. It is also assumed that for such a V there is a non-empty set $result(V)$ which comprises all possible outcomes of the processing of messages in V. Each element U of $result(V)$ is a set of m messages $\{result_1{:}v_1, \ldots, result_m{:}v_m\}$. It is assumed at this level of abstraction that an execution of any CA action produces a single result for each of the participating threads. This is a simplifying assumption which could easily be relaxed. Note also that an aborted execution of the CA action can be modelled by $\{result_1{:}abort, \ldots, result_m{:}abort\} \in result(V)$. Below, for any consistent input V, $\rho(V) \stackrel{\text{df}}{=} \{i \leq m \mid \alpha call_i \cap V = \emptyset\}$ identifies those channels $call_i$ on which a suitable input is still expected and, for every $i \in \rho(V)$, the set of all $v \in \mu call_i$ such that $V \cup \{call_i{:}v\}$ is still a consistent input is denoted by $\pi_i(V)$.

After assembling a valid input, CA proceeds in one of two ways. First of all, it can report on channel d_1 that the action has been completed successfully, and then non-deterministically produce one of the possible outcomes in $result(V)$. Alternatively, it can invoke the CA action process CA′ implementing ACT′, by redirecting the data received on *call* using the channels $rdir_1, \ldots, rdir_m$, and wait for a synchronisation message $g{:}go$ from CA′ before starting to construct another valid input. We can model this by $\text{CA} \stackrel{\text{df}}{=} \text{CA}_\emptyset$. To define CA_V, some notations are helpful: $redir(V)$ is V with each $call_i{:}v$ replaced by $rdir_i{:}v$, ψ : *Valid* $\to \{\xi^1, \ldots, \xi^n\}$ is a mapping which describes how an input to CA process is interpreted as an object at the higher level, $\iota(\xi^i) \stackrel{\text{df}}{=} i$ is the index of every processed object. We may now define:

$$\text{CA}_V \stackrel{\text{df}}{=} \begin{cases} \Box_{i \in \rho(V),\, v \in \pi_i(V)}\ call_i{:}v \to \text{CA}_{V \cup \{call_i:v\}} \Box \\ \Box_{i \in \rho(V),\, v \in \mu call_i - \pi_i(V)}\ call_i{:}v \to \text{CA}_V & \text{if } |V| < m \\ \bigsqcap_{U \in result(V)}\ (d_1{:}\iota(\psi(V)) \to \widehat{\text{CA}}_U)\ \sqcap\ \widetilde{\text{CA}}_{redir(V)} & \text{if } |V| = m \end{cases}$$

$$\widehat{\text{CA}}_U \stackrel{\text{df}}{=} \begin{cases} \Box_{a \in U}\ a \to \widehat{\text{CA}}_{U-\{a\}} & \text{if } U \neq \emptyset \\ \text{CA}_\emptyset & \text{if } U = \emptyset \end{cases}$$

$$\widetilde{\text{CA}}_U \stackrel{\text{df}}{=} \begin{cases} \Box_{a \in U}\ a \to \widetilde{\text{CA}}_{U-\{a\}} & \text{if } U \neq \emptyset \\ g{:}go \to \text{CA}_\emptyset & \text{if } U = \emptyset\ . \end{cases}$$

Verifiying the CA action design To show that CA indeed implements ACT, we need to find suitable extraction patterns. Below we show how to devise one

capable of relating communication on channels *call* to that on channel c; for the other channels one may proceed similarly.

We define by induction *Dom* and *extr*, together with an auxiliary mapping $\zeta : Dom \to \mathbb{P}(\alpha call)$, which for any trace in *Dom* yields the last unfinished consistent input built along this trace. To begin with, we have $\langle\,\rangle \in Dom$, $extr(\langle\,\rangle) \stackrel{\rm df}{=} \langle\,\rangle$ and $\zeta(\langle\,\rangle) \stackrel{\rm df}{=} \emptyset$. Suppose now that $t \in Dom$ and $v \in \mu call_i$. Then $u = t \circ \langle call_i{:}v\rangle \in Dom$ if $i \in \rho(\zeta(t))$. Let $V \stackrel{\rm df}{=} \zeta(t) \cup \{call_i{:}v\}$ if $v \in \pi_i(\zeta(t))$, and $V \stackrel{\rm df}{=} \zeta(t)$ otherwise. If $|V| = m$ then $extr(u) \stackrel{\rm df}{=} extr(t) \circ \langle c{:}\psi(V)\rangle$ and $\zeta(u) \stackrel{\rm df}{=} \emptyset$; otherwise $extr(u) \stackrel{\rm df}{=} extr(t)$ and $\zeta(u) \stackrel{\rm df}{=} V$. Finally, *dom* is the set of all $t \in Dom$ such that $\zeta(t) = \emptyset$, and for every $t \in Dom$, $ref(t)$ comprises all subsets, R, of $\alpha call$ such that $\alpha call_i \not\subseteq R$, for at least one $i \in \rho(\zeta(t))$.

Having designed the extraction patterns establishing that CA is an implementation of ACT, one might refine our design both by 'looking inside' CA, capturing its internal architectural details (we can handle this using, for instance, FDR and the standard CSP theory) and/or further refining the interface of the process, for example, by specifying in detail communication protocols used to receive inputs on the channels *call* (we can handle this using the approach proposed in this work, as CA is an *IO* process, and applying the algorithms presented in [3]).

5 Conclusions

Compositional development, as outlined in the case study of the previous section, is a cyclic process, which starts with an initial system design given in the form of a process network $\textsc{Sys}_0 \stackrel{\rm df}{=} P_1 \otimes \cdots \otimes P_r$ ($r \geq 1$). For such a high-level description, one can relatively easily verify the relevant correctness requirements, e.g., using FDR or a similar tool. Suppose now that, after a number of iterations, the current system description comes in the form of a network $\textsc{Sys}_i \stackrel{\rm df}{=} Q_1 \otimes \cdots \otimes Q_s$ satisfying $\textsc{Sys}_i \preceq \textsc{Sys}_0$. According to the argument put forward in the previous section, $\textsc{Sys}_i$ constitutes a suitable implementation of the initial (correct) design. There are now two directions for further development:

- Find a process of $\textsc{Sys}_i$, e.g., Q_s, and implement it as a sub-network

$$Q'_1 \otimes \cdots \otimes Q'_q \sqsupseteq Q_s\ .$$

  By compositionality, the resulting network,

$$\textsc{Sys}_{i+1} \stackrel{\rm df}{=} Q_1 \otimes \cdots \otimes Q_{s-1} \otimes Q'_1 \otimes \cdots \otimes Q'_q$$

  satisfies $\textsc{Sys}_{i+1} \preceq \textsc{Sys}_i$ as $Q \sqsupseteq P$ always implies $Q \preceq P$, and so we have $\textsc{Sys}_{i+1} \preceq \textsc{Sys}_0$.
- Find *IO* processes of $\textsc{Sys}_i$, e.g., Q_{s-1} and Q_s, whose intercommunication interface can be refined, and replace them by their respective implementations, Q'_{s-1} and Q'_s, designed according to suitably chosen extraction pattern(s). The resulting network,

$$\textsc{Sys}_{i+1} \stackrel{\rm df}{=} Q_1 \otimes \cdots \otimes Q_{s-2} \otimes Q'_{s-1} \otimes Q'_s$$

satisfies $\textsc{Sys}_{i+1} \preceq \textsc{Sys}_i$, and so we have $\textsc{Sys}_{i+1} \preceq \textsc{Sys}_0$.

In further stages, if the second kind of development is to be applied, both Q'_{s-1} and Q'_s must be *IO* processes and, in the previous section, this was always the case. However, if more complex refinement steps are carried out, the situation may require a degree of care. A typical example would be to implement a communication on a channel *ch*, from Q_{s-1} to Q_s, using two communication channels, *data* and *ack* passing messages of a feedback-controlled protocol. In such a situation, it is usually not the case that *ack* is a value-independent channel of Q'_s and, strictly speaking, such a process is no longer an *IO* process with *ack* treated as its input channel. Fortunately, channels like *ack* will generally be value-independent in Q'_{s-1}, and a simple way out of the problem is to syntactically make *ack* an output channel of Q'_s and an input channel of Q'_{s-1}, before proceeding with further development of the network.

Of course, the first kind of iteration step can also present us with the problem of maintaining the property of value-independence for the channels internal to the sub-network $Q'_1 \otimes \cdots \otimes Q'_q$. This, however, is usually easy to address by looking up the code of the processes.

Acknowledgments This research was supported by an EPSRC studentship grant and the EU-funded DSoS project.

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) .
3. J. Burton, M. Koutny and G. Pappalardo: Verifying Implementation Relations in the Event of Interface Difference. Proc. of *FME 2001*, Springer-Verlag, Lecture Notes in Computer Science 2021 (2001) 364-383.
4. J. Burton, M. Koutny and G. Pappalardo: Implementing Communicating Processes in the Event of Interface Difference. Proc. of *ACSD 2001*, IEEE Computer Society (2001) 87-96.
5. 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).
6. 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.
7. C. A. R. Hoare: *Communicating Sequential Processes*. Prentice Hall (1985).
8. M. Koutny, L. Mancini and G. Pappalardo: Two Implementation Relations and the Correctness of Communicated Replicated Processing. *Formal Aspects of Computing* 9 (1997) 119-148.
9. M. Koutny and G. Pappalardo: Behaviour Abstraction for Communicating Sequential Processes. *To appear in Fundamenta Informatica* (2002) .

10. L. Lamport: The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks* 2 (1978) 95-114.
11. R. Milner: *Communication and Concurrency*. Prentice Hall (1989).
12. B. Randell, A. Romanovsky, R. J. Stroud, J. Xu and A. F. Zorzo: Coordinated Atomic Actions: From Concept to Implementation. CSTR-595, University of Newcastle (1997).
13. B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, A. F. Zorzo, D. Schwier and F. von Henke: Coordinated Atomic Actions: Formal Model, Case Study and System Implementation. Manuscript (1997).
14. A. Rensink and R. Gorrieri: Vertical Implementation. *To appear in Information and Computation* (2002) .
15. A. W. Roscoe: *The Theory and Practice of Concurrency*. Prentice-Hall (1998).
16. 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) .