Under consideration for publication in Formal Aspects of Computing

Responsiveness of Interoperating Components

J.N. Reed[1], J.E. Sinclair[2] and A.W. Roscoe[3]

[1]Department of Information Technology, Armstrong Atlantic State University, Savannah Ga, US;
[2]Department of Computer Science, University of Warwick, Coventry, UK;
[3]Oxford University Computing Laboratory, University of Oxford, UK

Abstract. This paper investigates the issue of responsiveness of interoperating components: one not causing the other to deadlock. This is obviously related to the question of whether the two deadlock when put in parallel. However, it is different in that we require that a specific process P is not itself blocked by a plugin Q when it could otherwise have progressed, instead of asking that either process can always proceed (deadlock freedom). The issue becomes yet more subtle when dealing with processes which can nondeterministically block, either through graceful termination or unfortunate deadlock. The relational predicate, that is, binary relation on processes, which we provide is refinement-closed. This is significant as it allows components to be developed independently. In addition, it can be mechanically verified. The contribution of this paper is to identify the issue of responsiveness; to define appropriate properties; to demonstrate the suitability of these properties and consider how they can be mechanically verified. The notation used is CSP with automatic model-checking provided by the FDR tool.

Keywords: Responsiveness; Component-based systems; Refinement; CSP; Deadlock.

1. Introduction

Distributed component-based systems are gaining increasing practical acceptance, as testified in a special section of *Communications of the ACM* [Ars02] on enterprise components and services. A component is a unit of composition, which can be composed with other components and integrated into systems in a predictable way. A component communicates with its environment through interfaces. Separation of interface from implementation allows integration and deployment to be independent from life-cycle development. This calls for different types of interfaces. Unfortunately, the current component-based technologies manage only the specification of functional properties, with such properties limited to syntactic lists of operations and attributes, as assessed by Crnkovic *et al.* [CHJK02].

[1] Supported in part by the US Office of Naval Research.
Correspondence and offprint requests to: Jane Sinclair, Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK. Email: jane@dcs.warwick.ac.uk.

In a distributed component-based system it can be the case that a transaction or service involves the interoperation of a chain of components and services collectively required to achieve a desired result. Various components may be selected as off-the-shelf products to plug in to a particular application. The correct operation of the main application depends not only on the integrity of its own functions, but also on its interactions with separately developed components.

Many methodologies and tools are currently used in industry to support a modular approach to system design. The MasterCraft environment [Int] for example, supports the notion of taking different "views" in the development of medium sized systems (typically 5-20 million lines of code). Components may be provided from any source or implemented in any way, as long as they satisfy their interface requirements. Such environments can be of great practical advantage, although there is often no formally defined notion of consistency to ensure compatibility between the different views. Component-based approaches such as CORBA or COM regard a system as a collection of interoperating constituents requiring middleware management for consolidation and integration. Middleware addresses the integration of components by defining interface syntactic-based standards that allow diverse components to interoperate, but in itself does not address the problem of achieving reliable and secure interoperation.

In this paper we specifically deal with the issue of responsiveness of components: one not causing the other to deadlock. This issue becomes subtle when dealing with processes which may nondeterministically block. For example, consider a secure database server which supplies information to authorised clients. A specification for the top-level components should be able to state basic requirements to be satisfied by suitable plug-in components. This strategy is particularly applicable to security functions such as user authentication or data encryption. The top level specification for the client simply needs to know that a task will be accomplished (such as successful distribution of a common session key to both database and client, or a report of failure due to lack of authorisation), with no need to specify the details of any particular protocol. An essential concern at the top level is that the plug-in components charged with the subtasks will respond, that is, a plug-in to the client should not cause it to deadlock. Clients should be able to terminate at will (that is, nondeterministically), after calling on the database services an arbitrary number of times. Note that the same database may be plugged-in simultaneously to several clients, and it should not block any of them.

We address this notion of responsiveness of one component to another by taking a relational view of components. We consider one component Q to be a responsive plug-in to another P provided that Q never refuses to communicate with P when P wants to communicate with it. Our formulation is given in the CSP failures/divergences model.

The most common form of CSP specifications are the so-called *behavioural specifications* which can be written in the form $SPEC \sqsubseteq P$, which states that all behaviours and refinements of a process P conform to a constant specification $SPEC$. Every behavioural specification is both refinement closed (if a process P satisfies this refinement, then so do all P's refinements) and distributive (if $SPEC \sqsubseteq P$ and $SPEC \sqsubseteq Q$ then $SPEC \sqsubseteq P \sqcap Q$). Our formulations of responsiveness turn out not to be behavioural specifications, since our first attempt will not be refinement closed and our final one is not distributive. Indeed it is the fact that the first one is not refinement closed which necessitates the derivation of the second, since this failure in a functional specification of P and Q is paradoxical and would mean that P and Q could not be further developed, or implementation decisions made, without perhaps negating our specification.

Just because our specifications are not behavioural does not mean they cannot be checked by refinement (which is what FDR does): there have been several previous examples of this such as determinism [Laz99] and fault tolerance [Ros98]. The third author has recently [Ros03] (inspired by his early work on the present paper) demonstrated some general techniques for translating more-or-less arbitrary predicates of a single process into refinement checks of the form $F(P) \sqsubseteq G(P)$.

Section 2 contains an overview of CSP. In Section 3 we give intuitive motivation and requirements for the notion that one component behaves as a responsive plug-in to another. We also illustrate that desirable relational properties are not in general preserved by refinement. In Sections 4 and 5, we formalise two relationships: one which is desirable but not preserved by refinement, and a second which is the weakest refinement-preserving strengthening of the first. It is this second relationship which we regard as characterising responsiveness of one component to another. Section 6 considers methods for automatically checking these relationships. Section 7 provides a summary of the technical results. Section 8 presents relations to other work and conclusions.

Authors Reed and Sinclair are responsible for the main concepts of this paper including the definitions of the specifications we study. Roscoe contributed some of the technical results and derived the methods for automated checking.

2. An introduction to CSP

CSP [Hoa85, Ros98] models a system as a *process* which interacts with its environment by means of atomic *events*. Communication is synchronous: an event takes place precisely when both process and environment agree on its occurrence. This, rather than assignments to shared variables, is the fundamental means of interaction between agents. The CSP process algebra can be used to specify and more generally model networks, protocols, and distributed systems.

A related series of semantic models capture different aspects of observable behaviours of processes. The simplest semantic model is the *traces* model which characterises a process as a set of all its finite traces, $traces(P)$, representing observable sequences of events it can perform. These events are drawn from a set Σ, containing all possible communications for processes in the universe of consideration. The traces model is sufficient for reasoning about safety properties, but not liveness properties. In this paper, we use the failures/divergences model in which a process P is modelled as a set of *failures* and *divergences*. A *failure* consists of a pair (s, X) with s a finite trace of events drawn from Σ and X a subset of events of $\Sigma^{\checkmark}$. Here, $\Sigma^{\checkmark}$ denotes the set of all communication events together with a special event $\checkmark$ which signals that a process has cleanly terminated. (In general, if $A \subseteq \Sigma$, then $A^{\checkmark}$ will be used as an abbreviation for $A \cup \{\checkmark\}$.) The pair (s, X) is a failure if P may engage in the sequence s and then refuse all of the events in X. The set X is called a *refusal*. The set $divergences(P)$ is the set of traces on which P can diverge, meaning perform an infinite unbroken sequence of internal events. In this paper we assume all processes are divergence-free, a sensible convention since divergence-freedom is a desirable property which can be mechanically checked.

A process P is a refinement of process S ($S \sqsubseteq P$) if any possible behaviour of P is also a possible behaviour of S:

$$failures(P) \subseteq failures(S)$$

which tells us that any trace of P is a trace of S, and P can refuse a set of events X after engaging in trace s, only if S can refuse X after engaging in s. Intuitively, suppose S (for "specification") is a process for which all behaviours it permits are in some sense acceptable. If P refines S, then any behaviour of P is as acceptable as any behaviour of S. S can represent an idealised model of a system's behaviour, or an abstract property corresponding to a correctness constraint, such as deadlock or livelock freedom. A wide range of correctness conditions can be encoded as refinement checks between processes. Mechanical refinement checking is provided by the model-checker, FDR [For].

Appendix A contains an overview of the syntax of CSP, together with semantics for the failures/divergences model.

3. Intuitive requirements of responsive plug-in relationships

We are studying top-level system specifications structured as sets of interoperable components which act in parallel. In particular, we are interested in auxiliary plug-in services, possibly separately developed, such as one supplied by a local module performing a cryptographic protocol. We also wish to allow these plug-in subtasks to operate remotely, perhaps provided by a remote server.

Arbitrary components may be combined using the CSP parallel operator, but care must be taken to achieve desirable behaviour. A process P, which is itself deadlock-free can be placed in parallel with another process which does not behave in a desirable manner, causing P to block.

Example 1.

Suppose process P makes a request on channel *request* to a server, after which P expects to receive a reply on channel *reply*:

$$P = request \rightarrow reply \rightarrow P$$

Any other events in the system are ignored for the moment. Process Q is a server which is happy to interact with P, by taking in a *request*, and sending back a *reply*:

$$Q = request \rightarrow reply \rightarrow Q$$

We can run P and Q in parallel using the process $P \parallel Q$, which indicates that P and Q must synchronise on the events in $\Sigma = \{request, reply\}$. We regard Q as a responsive plug-in to P, since their parallel behaviour conforms to that desired by P. Clearly not every process which accepts requests should be considered as a responsive plug-in to P. For example, consider Z, which takes in requests, but does not reply to them:

$$Z = request \rightarrow Z$$

This time from P's perspective, Z does not respond to P in a desirable way, that is, Z blocks P. Since Z does not have a cooperating pattern of interaction, $P \parallel Z$ makes no further progress after the first step. The combination is said to deadlock. $\Box$

In general, we regard Q to be a responsive plug-in to P if Q is prepared to co-operate with the pattern set out by P for their shared interface. If P allows an (external) choice then Q may resolve it. If P's actions are nondeterministic, Q must be prepared to deal with each possibility. This can be characterised in terms of deadlock: Q must not block P on the set of their synchronised events when they are run in parallel. We formalise this notion after examining some other examples.

Example 2.

Suppose process P makes a *request* to a server, after which P expects either a *reply* or a *differentreply*:

$$P = request \rightarrow (reply \rightarrow P \,\Box\, differentreply \rightarrow P)$$

The box operator here represents an external (deterministic) choice. Process Q has only one way of responding:

$$Q = request \rightarrow reply \rightarrow Q$$

Here, P is prepared to accept different responses and will be happy with Q since it can supply one of the possible acceptable patterns. $P \parallel Q$ runs successfully (that is makes progress) without deadlocking on $\Sigma = \{request, reply, differentreply\}$, and we regard Q as responsive to P. $\Box$

For deadlock-free processes, testing that $P \parallel Q$ is also deadlock-free is all that is required. But the assumption is too restrictive. Let us imagine that we want P to trigger Q, possibly handing over some parameters which Q processes, subsequently replying back to P. P is in control, and may invoke Q arbitrarily, including never; Q is always required to be ready whenever P is. It is this relationship which we wish to formalise, bearing in mind that individual processes may block on their own. In the following examples, we examine processes which (nondeterministically) block by refusing all events (deadlock with $STOP$), but having them block in a more graceful fashion (clean termination with $SKIP$) would serve our purposes as well.

Example 3.

Suppose P and Q are defined as follows:

$$P = (x \rightarrow P) \sqcap STOP$$
$$Q = (x \rightarrow Q)$$

Here, P is defined using internal (nondeterministic) choice and so may or may not choose to engage in x. We regard Q as exhibiting responsive behaviour, because it is always willing to engage in x with P. However, P is obviously not deadlock-free and neither is $P \parallel Q$. This example also shows that responsiveness is not symmetric. P can choose not to engage in x when required by Q, and so should not be regarded as responsive to Q. $\Box$

Not only is P allowed to block, but there may be legitimate reasons for a specification of a plug-in Q to block in certain situations. For example, Q may be a set-up process which needs only to respond to a one-time invocation from P. $P \parallel Q$ would properly block on their common set of events after Q finishes its work, after which P carries on with other events not requiring any participation from Q. Q needs only to be available when required by P, and it is of no concern to P whether Q blocks afterwards.

As we want the combined system to block only when P does, one property to consider might be the relational requirement that the parallel composition should be a refinement of P:

$$P \sqsubseteq P \parallel Q \qquad\qquad (*)$$

since it would follow that if $P \parallel Q$ can block at any point, then P itself must have been able to. However, this property is overly restrictive. It is not satisfied by the processes in Example 2 because the right-hand side has a larger number of failures than the left-hand side. Still, property (*) is worth examining because it illustrates that relational constraints are not necessarily preserved by refinement, as demonstrated in the next example.

Example 4.

This time, P and Q are defined as:

$$P = (x \rightarrow P) \sqcap STOP$$
$$Q = (x \rightarrow Q) \sqcap STOP$$

Here, both P and Q may decide to engage in x but can each, nondeterministically, also choose not to. This P and Q would satisfy $(*)$ since $P \parallel Q = P = Q$. But if refinements P' of P and Q' of Q are defined:

$$P' = x \rightarrow P' \qquad\qquad Q' = STOP$$

it is clear that P' and Q' do not satisfy $(*)$, for example, $(\langle\rangle, \{x\})$ is a failure of $P' \parallel Q'$ but not of P'. This example also shows the rather dramatic effect of internal choice and non-determinism in Q. Q is not responsive to P since Q may refuse to engage in x precisely when P wishes to, causing deadlock. $\square$

As in the previous example, a given relationship may hold between interacting processes and yet it may not follow that component-wise refinements are compatible according to the same criteria. It is an important requirement for our definition of responsiveness that individual component specifications should admit independent refinement and that the implementations should still be responsive in the same way.

We will refer to a relationship between cooperating specifications P and Q as *refinement-closed* if it is preserved through independent refinement of the component specifications.

Definition 1. If ϕ is a relation on specifications P and Q then ϕ is *refinement-closed* if and only if, for all P', Q' such that $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$, it is the case that:

$$P \;\phi\; Q \Rightarrow P' \;\phi\; Q'$$

A relationship which is not refinement-closed for general P and Q may be made refinement-closed by adding further constraints. For example, the $(*)$ property along with the requirement that P is deterministic is refinement-closed. (Of course a relationship with no deterministic pairs of processes satisfying it can only be strengthened to *false*.)

The previous examples provide an intuitive idea of the property we wish to capture. The next step is to formalise its features. In what follows, we identify two relationships. The first one, given in Section 4, uses failures-based properties to define the ideas of blocking and hence of non-blocking behaviour. This property follows the intuition of the preceding examples by attempting a natural encapsulation of the requirement that one process does not block another. However, we will see that though this desirable property would appear necessary, it is not sufficient, resulting in “false positives” corresponding to lack of refinement-closure. We identify some constraints which for some special cases ensure the relationship holds and is refinement-closed. These still do not fully characterise component responsiveness. In Section 5 we present a second less intuitive relational property which characterises the general notion of responsiveness. We also establish that this second property is precisely the weakest refinement-preserving strengthening of the first one.

4. Formalisation of blocking and non-blocking plug-in relationships

Terminology In the following definitions, P and Q represent processes, P' and Q' represent respective refinements, and J represents the set of shared events of P and Q, that is, the events requiring participation of both processes. $J^{\checkmark} = J \cup \{\checkmark\}$, the set J of shared events with the special termination event $\checkmark$ added since CSP parallel operators all effectively synchronise on $\checkmark$ (see [Ros98] for extensive discussion of termination). For simplicity in most of our examples, we take $J = \Sigma$.

A general formulation of responsiveness is required which both ensures refinement-closure and does not unnecessarily constrain processes from blocking.

4.1. Failures-based definitions of blocking

Our first attempt to formalise component responsiveness is via a direct definition of blocking using a failures-based property. We say that Q blocks an "otherwise live" P whenever P and Q operating in parallel can block though P on its own would not. If A is some set of events for which $P \parallel_J Q$ could reach a point where the whole of A is refused and yet would not be refused at this point by P alone, then clearly Q blocks P from making any further progress on events from A.

Definition 2. Q *BlocksLive* P **on** A for non-empty $A \subseteq J^{\checkmark}$ means that there exists trace s such that

$$(s, A) \in \mathit{failures}(P \parallel_J Q) \wedge (s, A) \notin \mathit{failures}(P)$$

Clearly Q causing P to deadlock on all of J would make Q an undesirable plug-in to P, and requiring any potential plug-in to P to fail to satisfy the above relationship for $A = J$ would seem reasonable. That is, we want to ensure that processes P and Q operating in parallel do not block on J when it is the case that P on its own would not block on J. We say that Q *RespondsToLive* P on A if and only if $\neg$ (P *BlocksLive* Q on A). That is,

Definition 3. Q *RespondsToLive* P **on** A for $A \subseteq J^{\checkmark}$ means that for every trace s

$$(s, A) \in \mathit{failures}(P \parallel_J Q) \Rightarrow (s, A) \in \mathit{failures}(P)$$

We say that Q *RespondsToLive* P to mean Q *RespondsToLive* P on $J^{\checkmark}$.

Referring to the processes of Example 2, there is no trace of $P \parallel Q$ after which the whole of the joint alphabet is refused, so in this case Q is obviously not blocking and Q *RespondsToLive* P. In Example 3, any deadlock of $P \parallel Q$ would be caused by P, so in this case also, Q *RespondsToLive* P.

These two definitions have much in common with those of an *ungranted request* and thus *absence of ungranted requests* from earlier work on deadlock (e.g. [BR85, RD87, Ros98]). The differences are that the two definitions have different sets of parameters and that (as will be evident in this paper) we here include reasoning about clean termination $\checkmark$ which was previously avoided. Since ungranted requests have long been known to be the building blocks of deadlock, it is only natural that here we specify the absence of something very similar.

The property is distributive in each argument separately, for example Q *RespondsToLive* P_i for $i \in \{1, 2\}$ implies Q *RespondsToLive* $(P_1 \sqcap P_2)$. This is straightforward to prove, and also follows easily from the representation in Appendix B given for automated checking.

Example 5.

$$P = (x \to P) \sqcap (y \to P)$$
$$Q = (x \to Q) \sqcap (y \to Q)$$

These processes both display internal choice. Here, $(\langle\rangle, \{x, y\})$ is a failure of $P \parallel Q$ since P and Q could demand to engage in different events resulting in deadlock, but P itself is always willing to make progress on one of x or y. Hence it is not the case that Q *RespondsToLive* P. □

Example 6.

Consider the following processes P and Q. P outputs a request $t \in T$ of its own choice on channel *request*, and expects some arbitrary response $r \in R$ on channel *reply* from Q.

$$P = \sqcap_{t:T}\, \mathit{request}!t \to \mathit{reply}?r \to \mathit{STOP}$$
$$Q = \mathit{request}?t \to \sqcap_{r:R}\, \mathit{reply}!r \to Q$$

Immediately after a request, all of $request = \{request.t \mid t \in T\}$ may be refused by $P \parallel Q$. But all of $request$ may be refused by P then as well, and at all other times when $request$ is refused by $P \parallel Q$. Thus Q *RespondsToLive* P on $request$. That is,

$$(s, request) \in failures(P \parallel Q) \Rightarrow (s, request) \in failures(P)$$

Hence immediately after a request it is acceptable for all of $request$ to be refused by Q, since Q alone would not be the sole source of blocking on $request$. It is also true that Q *RespondsToLive* P on $reply$, since whenever $P \parallel_J Q$ refuses all of $reply$, P can refuse all of $reply$. Likewise, Q *RespondsToLive* P on $J^{\checkmark}$. However, with:

$$Z = request?r \rightarrow Z$$

P could issue a particular request t on $request$, and wait forever for a reply not forthcoming from Z. That is, we have $(\langle request.t \rangle, reply)$ as a failure of $P \parallel_J Z$ but not a failure of P. So Z is the sole cause of the deadlock with P on $reply$, and Z *BlocksLive* P. □

The enormous value of this concept to system development is underlined by the following result.

Theorem 1. Suppose $\mathcal{N} = \{(P_i, A_i) \mid i \in \{1..N\}\}$ is a network of alphabetised CSP processes (in which the alphabet of P_i is A_i). Suppose that Q is a plug-in process whose alphabet J is disjoint from $A_i \cap A_j$ for each $i \neq j$ (i.e. the P_i each communicate with Q individually, not as part of some multi-way synchronisation). Then if

(i) $J \cap A_j \neq \emptyset$ for at least one j,
(ii) $\mathcal{N}$ is deadlock free, and
(iii) Q *RespondsToLive* P_i for each i with $A_i \cap J \neq \emptyset$ (using this set as their synchronisation set),

then $\mathcal{N}' = \mathcal{N} \cup \{(Q, J)\}$ is itself deadlock free.

Thus adding responsive plug-ins, even in complex situations, can never introduce deadlock.

Proof If $\mathcal{N}'$ can deadlock then there is some trace s after which it can refuse $\Sigma^{\checkmark}$. The subnetwork $\mathcal{N}$ itself has a failure (s', X) contributing to this, where $s' = s \upharpoonright \bigcup A (= \{A_i \mid i \in \{1..N\}\})$ and where X is maximal. We can further decompose this into failures (s_i, X_i) of the P_i where again the X_i are maximal. Similarly Q has failure (s'', Y) with Y maximal, and $(X \cap A^{\checkmark}) \cup (Y \cap J^{\checkmark}) = (A \cup J)^{\checkmark}$.

A consequence of the maximality of the failures we have chosen is that each of them is either Σ or contains $\checkmark$ (F4[2] and F3). We know that $\mathcal{N}$ is deadlock free, so $X \neq \Sigma^{\checkmark}$. So either $X = \Sigma$ and no P_i can refuse $\checkmark$, or $\emptyset \neq A^{\checkmark} - X \subseteq J$. Let's consider these two cases separately.

In the first case we know that Q *RespondsToLive* P_j where $A_j \cap J \neq \emptyset$. If Y contained $\checkmark$ we would have $(s'_j, X_j^{\checkmark}) \in failures(P_j \parallel_{A_j} Q)$ but not $(s'_j, X_j^{\checkmark}) \in failures(P_j)$ contradicting the assumption that Q *RespondsToLive* P_j. So this case is impossible.

In the second case we know that there must be some i with $(A_i - X_i) \cap J \neq \emptyset$, and hence (following our observation above) we know that $\checkmark \in X_i$. The facts (a) that we are looking at a deadlock state of $\mathcal{N}'$ and (b) that $A_i \cap A_k \cap J = \emptyset$ for all $k \neq i$ means that $A_i - X_i \subseteq Y$ (i.e., Q must be refusing all events in these two processes' interface that P_i is not). Hence $(A_i \cap J)^{\checkmark} \subseteq X_i \cup Y$, which contradicts our assumption that Q *RespondsToLive* P_i. □

4.2. False positives with the definition

The *RespondsToLive* definition captures the required behaviour in many cases, but closer inspection reveals that there are some situations in which it does not. Note that if Q *RespondsToLive* P on A, either (1) $(s, A) \notin failures(P \parallel Q)$, which means P and Q do not block when operating together, or (2) $(s, A) \in failures(P)$, which means that any blocking might have been due to P. So, if a deadlock *could* have been caused by P, Q is not considered as blocking. This becomes apparent in the following example where P may nondeterministically stop.

[2] Here F3 and F4 are two of the axioms from the failures-divergences model, which are set out in Appendix A.

Example 7.

$$P = (x \rightarrow P) \sqcap STOP$$
$$Q = STOP$$

For any trace s, $(s, \{x\})$ is a failure of P since P may choose to stop at any point. Thus P and Q satisfy the *RespondsToLive* definition. However, intuition tells us that Q would really be blocking P if P wished to engage in x. Furthermore, the refinements:

$$P' = x \rightarrow P'$$
$$Q' = STOP$$

show that *RespondsToLive* is not refinement-closed. Here, $P \sqsubseteq P'$, $Q \sqsubseteq Q'$ and P, Q satisfy Q *RespondsToLive* P. But it is not true that Q' *RespondsToLive* P', and indeed, Q' does not behave as a responsive plug-in to P'. □

In fact, it is exactly these processes with badly-behaved refinements which characterise the "false positives" of the *RespondsToLive* definition. Adding the requirement of refinement-closure captures exactly the notion that Q, no matter what its behaviour, can never block P, no matter what its behaviour. This requires that neither Q nor any of its refinements be the sole cause of deadlock of P or any of its refinements. Thus, we want to characterise processes for which the *RespondsToLive* relationship is preserved by refinement.

Upon examination, the presence of nondeterminism reveals further difficulties. It would be useful if a process which was responsive on several channels independently could be taken to be responsive on the union of these channels. However, if Q *RespondsToLive* P on each of channels T and R, it is not necessarily the case that Q *RespondsToLive* P on $T \cup R$. Consider Example 5. Q *RespondsToLive* P on both $\{x\}$ and $\{y\}$ independently since for any $s \in traces(P)$, $(s, \{x\}) \in failures(P)$ and $(s, \{y\}) \in failures(P)$, we have

$$(s, \{x\}) \in failures(P \parallel_J Q) \Rightarrow (s, \{x\}) \in failures(P)$$
$$(s, \{y\}) \in failures(P \parallel_J Q) \Rightarrow (s, \{y\}) \in failures(P)$$

However, $(s, \{x, y\}) \in failures(P \parallel_J Q)$ but $(s, \{x, y\}) \notin failures(P)$, so Q *BlocksLiveP* rather than Q *RespondsToLive* P on $\{x, y\}$.

4.3. Low-deterministic processes

The previous examples illustrate that there is a tension among internal choice, non-determinism and refinement-closure for *RespondsToLive*. Clearly, eliminating non-deterministic behaviours is not the solution, since this would impoverish the modelling capabilities of the language. In Section 5 a more general definition is proposed, but first we note that there are certain types of process with limited nondeterminism for which *RespondsToLive* is adequate. As an example, we give some results showing the rather dramatic effects of reducing non-determinism via *low-determinism* [Ros98], which guarantees that processes are deterministic on a certain subsets of events.

Definition 4. A process is low-deterministic on L if it is divergence free, and after any trace s, cannot both accept and refuse any event a in L.

For instance, in Example 6, P is low-deterministic on *reply*.

Theorem 2. Let processes P and Q synchronise on set J, $P \sqsubseteq P'$, and $Q \sqsubseteq Q'$. If Q *RespondsToLive* P on A for $A \subseteq J^{\checkmark}$ and P is low-deterministic on A, then Q' *RespondsToLive* P' on A.

Proof We use proof by contradiction. Assume the hypothesis and the negation of the conclusion:

$$
\begin{array}{ll}
(s,A) \in \mathit{failures}(P' \parallel_J Q') \ \wedge (s,A) \notin \mathit{failures}(P') & \\
\quad \Rightarrow & [\mathrm{def}^n \text{ of } \sqsubseteq] \\
\qquad (s,A) \in \mathit{failures}(P \parallel_J Q) \ \wedge (s,A) \notin \mathit{failures}(P') & \\
\quad \Rightarrow & [\mathrm{def}^n \text{ of } \mathit{RespondsToLive}\,] \\
\qquad (s,A) \in \mathit{failures}(P) \ \wedge (s,A) \notin \mathit{failures}(P') & \\
\quad \Rightarrow & [\text{F1,F2,F3}] \\
\qquad \exists\, x \in A \bullet (s,\{x\}) \in \mathit{failures}(P) \wedge (s \frown \langle x \rangle) \in \mathit{traces}(P) & \\
\quad \Rightarrow & [\mathrm{def}^n \text{ of low-determinism}] \\
\qquad P \text{ is not low-deterministic on } A &
\end{array}
$$

which contradicts the hypothesis. □

Using Theorem 2, we can deduce for example that in Example 6 *RespondsToLive* is refinement-closed on *reply*. The following result shows another effect of reducing non-determinism: if a process Q *RespondsToLive* P on each of a pair of channels, and P is low-deterministic on either one of them, then Q *RespondsToLive* P on their union.

Theorem 3. Let P and Q be processes which synchronise on set J of events. If Q *RespondsToLive* P on $T \subseteq J^{\checkmark}$, Q *RespondsToLive* P on $R \subseteq J^{\checkmark}$, and P is low-deterministic on R, then Q *RespondsToLive* P on $T \cup R$.

Proof Assume the hypothesis:

$$
\begin{array}{ll}
(s, T \cup R) \in \mathit{failures}(P \parallel_J Q) & \\
\quad \Rightarrow & [\text{F2}] \\
\qquad (s,T) \in \mathit{failures}(P \parallel_J Q) \wedge (s,R) \in \mathit{failures}(P \parallel_J Q) & \\
\quad \Rightarrow & [\mathrm{def}^n \text{ of } \mathit{RespondsToLive}\,] \\
\qquad (s,T) \in \mathit{failures}(P) \wedge (s,R) \in \mathit{failures}(P) & \\
\qquad \wedge & [\mathrm{def}^n \text{ of low determinism}] \\
\qquad \forall\, r \in R \bullet s \frown \langle r \rangle \notin \mathit{traces}(P) & \\
\quad \Rightarrow & [\text{F3}] \\
\qquad (s, T \cup R) \in \mathit{failures}(P) &
\end{array}
$$

□

For the processes of Example 5, P is low-deterministic on *reply* and Q *RespondsToLive* P on each of *reply* and *request*, so Theorem 3 confirms the result that Q *RespondsToLive* P on $\mathit{reply} \cup \mathit{request}$. Analogous results to Theorems 2 and 3 in which Q is taken to be low-deterministic instead of P do *not* hold.

5. Refinement-closed responsive plug-ins

Although the *RespondsToLive* definition is adequate in certain restricted situations, such as with low-deterministic processes, as discussed in Section 4.2, it is not what we want in general. The difficulty with the *RespondsToLive* definition is that, in situations such as that of Example 7, if P *might* have been the cause of deadlock then Q is regarded as responsive even if Q itself could also have been the cause. *RespondsToLive* cannot distinguish between P stopping and Q stopping in this case. Adding constraints to the definition to achieve full generality may be possible but becomes less straightforward, so here we present a property arrived at by a slightly different route.

5.1. Definition of responsiveness

For Q to be responsive to P, if at any point in their execution P can require that Q participate in any one event from some set, s, then Q must be willing to engage in at least one of these events. Some examples:

- if $P = (x \rightarrow P) \;\Box\; (y \rightarrow P)$ then, after any trace, P can demand cooperation in at least one event from $\{x, y\}$;
- if $P = (x \rightarrow P) \sqcap (y \rightarrow P)$ then at any point P might demand cooperation in an event from $\{x\}$, or it might demand cooperation in an event from $\{y\}$;
- if $P = (x \rightarrow P) \sqcap (y \rightarrow P) \sqcap STOP$ then P can make the same demands as in the previous case.

To formulate this we introduce some more CSP notation. An *initials* set refers to the set of all possible first events for a particular process. That is, $\mathit{initials}(P) = \{a \mid \langle a \rangle \in \mathit{traces}(P)\}$. To refer to the set of possible next events for P after trace s has occurred we write $\mathit{initials}(P/s)$ where $P/s = \{t \in P \mid s \preccurlyeq t\}$ (read "P after s": the set of all traces which extend s). The set of all initial events of P which are also in some set A is denoted $\mathit{initialsInA}(P)$ and defined as $\mathit{initials}(P) \cap A$.

If (s, X) is a failure of P, then $\mathit{initialsInA}^{\checkmark}(P/s) - X$ contains all events outside of the refusal set X which are both possible next events for P/s *and* are in $A^{\checkmark}$. These sets for each failure (s, X), are the ones for which P may demand Q's participation. Therefore, after s, Q should not refuse the whole of this corresponding set. Sets which are (1) the complement of some *refusal*, and (2) contained in $\mathit{initials}(P)$ have been termed *acceptance sets* (see [Ros98]) though the same term is sometimes used for subtly different concepts.

Definition 5. For processes Q and P with joint alphabet J, Q *RespondsTo* P means that for all s, X:

$$(s, X) \in \mathit{failures}(P) \;\wedge\; \mathit{initialsInJ}^{\checkmark}(P/s) - X \;\neq\; \{\} \;\Rightarrow$$
$$(s, \mathit{initialsInJ}^{\checkmark}(P/s) - X) \notin \mathit{failures}(Q)$$

□

We regard Q to be a responsive plug-in to P if and only if Q *RespondsTo* P.

Example 8.

$$P = (\mathit{request}1 \rightarrow \mathit{reply} \rightarrow P) \sqcap (\mathit{request}2 \rightarrow \mathit{reply} \rightarrow P)$$
$$Q = (\mathit{request}1 \rightarrow \mathit{reply} \rightarrow Q) \;\Box\; (\mathit{request}2 \rightarrow \mathit{reply} \rightarrow Q)$$

After $\langle\rangle$ (or any even length trace) possible refusal sets and corresponding acceptance sets of P are:

Refusal	**Acceptance**
$\{\mathit{request}1, \mathit{reply}\}$	$\{\mathit{request}2\}$
$\{\mathit{request}2, \mathit{reply}\}$	$\{\mathit{request}1\}$
$\{\mathit{request}1\}$	$\{\mathit{request}2\}$
$\{\mathit{request}2\}$	$\{\mathit{request}1\}$
$\{\mathit{reply}\}$	$\{\mathit{request}1, \mathit{request}2\}$
$\{\}$	$\{\mathit{request}1, \mathit{request}2\}$

P may require Q's cooperation in an event from any of these acceptance sets. Since Q does not refuse any of these sets at this point in the execution, Q is responding to P's requirements.

Similarly, after any odd length trace, possible refusal sets for P are: $\{\mathit{request}1, \mathit{request}2\}$, $\{\mathit{request}1\}$, $\{\mathit{request}2\}$, $\{\}$. The corresponding acceptance set for each of these is $\{\mathit{reply}\}$ which is not refused at this point by Q. Hence Q *RespondsTo* P. □

Example 9.

Revisiting the process of Example 7 which illustrated the weakness of the previous definition:

$$P = (x \rightarrow P) \sqcap STOP$$
$$Q = STOP$$

$(\langle\rangle, \{\})$ is a failure of P with acceptance set $\{x\}$, and indeed, P may demand Q's participation in event x. However, $(\langle\rangle, \{x\})$ is a failure of Q, and so Q is not responsive to P. □

5.2. The relationship between the definitions

We now have two definitions: *RespondsToLive*, which did not fully capture the properties we required but was useful when restricted to certain classes of process; and *RespondsTo* which appears to be both necessary and sufficient in itself. The theorems in this section confirm the relationship between the properties, that is: Q *RespondsTo* P exactly when the relation Q' *RespondsToLive* P' holds for all $P' \sqsupseteq P$ and $Q' \sqsupseteq Q$.

Theorem 4. For any processes P and Q, the following are equivalent:

- Q *RespondsTo* P,
- For any refinements P' of P and Q' of Q, Q' *RespondsToLive* P'.

Proof This follows from Lemmas 1 and 2 below. □

Lemma 1 establishes that *RespondsTo* implies refinement-closed *RespondsToLive*.

Lemma 1. For processes, P, P', Q and Q', if Q *RespondsTo* P, $Q \sqsubseteq Q'$, and $P \sqsubseteq P'$, then

Q' *RespondsToLive* P'.

Proof Proof by contradiction. Assume the hypothesis and negation of the conclusion.

$$
\begin{array}{llr}
(s, J^{\checkmark}) \in \mathit{failures}(P' \parallel_J Q') \wedge (s, J^{\checkmark}) \notin \mathit{failures}(P') & (1) & \\
\quad \Rightarrow & & [\ \mathrm{def}^n \text{ of } \parallel\ ] \\
\qquad \exists\, X, Y \subseteq \Sigma^{\checkmark} \bullet & & \\
\qquad J^{\checkmark} = X \cup Y \wedge (s, X) \in \mathit{failures}(P') \wedge (s, Y) \in \mathit{failures}(Q') & (2) & \\
\quad \wedge & & [\ \mathrm{def}^n \text{ of } \sqsubseteq\ ] \\
\qquad (s, X) \in \mathit{failures}(P) \wedge (s, Y) \in \mathit{failures}(Q) & & \\
\quad \wedge & & [\ (1), \text{F3}\ ] \\
\qquad \mathit{initialsInJ}^{\checkmark}(P'/s) - X \neq \{\} & & \\
\quad \wedge & & [\ \mathrm{def}^n \text{ of } \sqsubseteq\ ] \\
\qquad \mathit{initialsInJ}^{\checkmark}(P/s) - X \neq \{\} & &
\end{array}
$$

Also, since Q *RespondsTo* P on $J^{\checkmark}$,

$$
\begin{array}{lr}
(s, \mathit{initialsInJ}^{\checkmark}(P/s) - X) \notin \mathit{failures}(Q) & \\
\quad \Rightarrow & [\ \mathrm{def}^n \text{ of } \sqsubseteq\ ] \\
\qquad (s, \mathit{initialsInJ}^{\checkmark}(P/s) - X) \notin \mathit{failures}(Q') & \\
\quad \Rightarrow & [(2), \text{F2}\ ] \\
\qquad \mathit{initialsInJ}^{\checkmark}(P/s) - X \nsubseteq Y & \\
\quad \wedge & [\ \text{set theory}\ ] \\
\qquad \exists\, z \in \mathit{initialsInJ}^{\checkmark}(P/s) - X \bullet z \notin Y & \\
\quad \wedge & [\ \text{set theory}\ ] \\
\qquad z \in J^{\checkmark} \wedge z \notin X \wedge z \notin Y &
\end{array}
$$

which contradicts (2). □

Lemma 2 states that refinement-closed *RespondsToLive* implies *RespondsTo* .

Lemma 2. If Q *RespondsToLive* P, and for any refinements P' of P and Q' of Q, Q' *RespondsToLive* P', then Q *RespondsTo* P.

Proof Proof by contradiction. Assume the hypothesis and negation of the conclusion.

For some s,X:
$(s, X) \in \mathit{failures}(P)$ and $\mathit{initialsInJ}^{\checkmark}(P/s) - X \neq \{\}$ (1)
$\wedge$ $(s, \mathit{initialsInJ}^{\checkmark}(P/s) - X) \in \mathit{failures}(Q)$ (2)
$\Rightarrow$ [(1), Lemma 3]
there exists a refinement P' such that
Also, $(s, J^{\checkmark}) \notin \mathit{failures}(P') \wedge (s, X) \in \mathit{failures}(P')$ (3)
$\wedge$ [F2]
$(s, X \cap J^{\checkmark}) \in \mathit{failures}(P')$

Also $(s, \mathit{initialsInJ}^{\checkmark}(P/s)) \in \mathit{failures}(P' \parallel_J Q)$ (4) [(1), (2), (3), def^n $\parallel$]

And, $\mathit{initialsInJ}^{\checkmark}(P'/s) \subseteq \mathit{initialsInJ}^{\checkmark}(P/s)$
$\Rightarrow$ [(4), F2]
$(s, \mathit{initialsInJ}^{\checkmark}(P'/s)) \in \mathit{failures}(P' \parallel_J Q)$
$\Rightarrow$ [F3]
$(s, J^{\checkmark}) \in \mathit{failures}(P' \parallel_J Q)$

By hypothesis Q *RespondsToLive* P'
$\Rightarrow$ [def^n *RespondsToLive*]
$(s, J^{\checkmark}) \in \mathit{failures}(P')$

which contradicts (3). □

Lemma 3. Assume $(s, X) \in \mathit{failures}(P)$ and $\mathit{initialsInJ}^{\checkmark}(P/s) - X \neq \{\}$. Then there is a refinement P' of P such that $(s, J^{\checkmark}) \notin \mathit{failures}(P')$ and $(s, X) \in \mathit{failures}(P')$.

Proof Assume the hypothesis. We will construct a set T of traces and a set F of failures for a process P' which form a refinement of P. The construction depends on whether or not the special termination event $\checkmark$ belongs to $\mathit{initialsInJ}^{\checkmark}(P/s) - X$.

Case 1: $\checkmark \in \mathit{initialsInJ}^{\checkmark}(P/s) - X$. Choose

$T = \mathit{traces}(P)$ and $F = \mathit{failures}(P) - \{(s, R) \mid \checkmark \in R\}$

Axioms F1 – F4 follow straightforwardly, and hence T and F define a refinement P' of P. We use the hypothesis of the case to establish that P' can refuse X: since $\checkmark \in \mathit{initialsInJ}^{\checkmark}(P/s) - X$, clearly

$\checkmark \notin X$ and hence, $(s, X) \in \mathit{failures}(P')$

By construction,

$(s, \{\checkmark\}) \notin \mathit{failures}(P')$, and hence $(s, J^{\checkmark}) \notin \mathit{failures}(P')$.

thereby establishing Case 1.

Case 2: $\checkmark \notin \mathit{initialsInJ}^{\checkmark}(P/s) - X$. Choose

$T = \mathit{traces}(P) - \{t \mid \exists\, x : X \bullet s \frown \langle x \rangle \leq t\}$
$F = (\{(t, H) \in \mathit{failures}(P) \mid t \in T\} - \{(s, H) \mid H \cap \mathit{initialsInJ}^{\checkmark}(P/s) - X \neq \{\}\}$

Notice that these traces and failures satisfy:

1. The traces of T and failures of F coincide with those of P except after s.
2. There are no extensions of s from X appearing as a trace in T.
3. No refusal of F contains any element of $\mathit{initialsInJ}^{\checkmark}(P/s) - X$.

Let us first observe that if T and F form a valid set of traces and failures for a refinement P', then by 2. and 3. above,

$(s, X) \in \mathit{failures}(P') \wedge (s, J^{\checkmark}) \notin \mathit{failures}(P')$

thereby establishing the required properties for P'.
We must establish that T and F satisfy axioms F1–F4. F1 and F2 follow straightforwardly. Let us now establish F3:

$$(t, R) \in F \ \wedge \ \forall y : Y \bullet t \frown \langle y \rangle \notin T \Rightarrow (t, R \cup Y) \in F$$

Case 2.F3.1: $t \neq s$. Follows straightforwardly from 1. and F3 for P.
Case 2.F3.2: $t = s$. Assume

$$(s, R) \in F \ \wedge \ \forall y : Y \bullet s \frown \langle y \rangle \notin T$$

and let

$$\begin{array}{lll} R1 = R \cap X & \text{and} & R2 = R - \mathit{initialsInJ}^{\checkmark}(P/s) \\ Y1 = Y \cap X & \text{and} & Y2 = Y - \mathit{initialsInJ}^{\checkmark}(P/s) \end{array}$$

which implies

$$\begin{array}{lr} R = R1 \cup R2 & \text{Earlier assumptions about } F \\ Y = Y1 \cup Y2 & \text{Definition of } T \\ (s, R1 \cup Y1) \in \mathit{failures}(P) & [\text{F2}] \\ (s, (R1 \cup Y1) \cup (R2 \cup Y2)) \in \mathit{failures}(P) & [\text{F3}] \\ (s, R \cup Y) \in F & [\text{set theory}] \end{array}$$

thereby establishing Case 2.F3.2, and F3.
To establish F4, we must show that $t \frown \langle \checkmark \rangle \in T \Rightarrow (t, \Sigma) \in F$
Assume $t \frown \langle \checkmark \rangle \in T$. We again take cases.
Case 2.F4.1: $t \neq s$. Since $t \frown \langle \checkmark \rangle \in \mathit{traces}(P)$, it follows $(t, \Sigma) \in \mathit{failures}(P)$ and $(t, \Sigma) \in F$.
Case 2.F4.2: $t = s$. Here we use the hypothesis of Case 2. Either $\checkmark \notin \mathit{initialsInJ}^{\checkmark}(P/s)$, or $\checkmark \in \mathit{initialsInJ}^{\checkmark}(P/s)$ and $\checkmark \in X$. In either case, $t \frown \langle \checkmark \rangle \notin T$, establishing Case 2.F4.2, F4, Case 2 and the lemma. □

5.3. A further property of the *RespondsTo* definition

Theorem 4 establishes that Q *RespondsTo* P exactly when Q *RespondsToLive* P holds and is refinement-closed. We can also derive from this result the fact that the *RespondsTo* relation is itself always refinement-closed. This is captured in the following theorem.

Theorem 5. *RespondsTo* is refinement-closed.

Proof Assume Q *RespondsTo* P. Theorem 4 ensures that for refinements: $P \sqsubseteq P'$, $Q \sqsubseteq Q'$, $P' \sqsubseteq P''$, and $Q' \sqsubseteq Q''$, we have Q'' *RespondsToLive* P''. Hence by the same theorem, Q' *RespondsTo* P'. □

Proof By contradiction. □

6. Mechanical verification of responsiveness

The properties described above provide a way of checking whether one component will respond to another in a suitable way. Refinement-closure of *RespondsTo* ensures that any further refinements will also work correctly together. This allows a service to be selected by considering a top-level specification, without worrying about the details of its implementation. A mechanised check would allow selection of services to take place without human intervention, for example by an agent which needs to make autonomous decisions to guide its choices. Both properties are semantically formulated, and hence, not directly mechanically verifiable.

[Ros03] establishes that any closed and refinement-closed predicate can be formulated as a machine-checkable predicate of the form $SPEC \sqsubseteq G(P)$ in the stable-failures model. He also establishes that any distributive predicate can be formulated as a machine-checkable predicate of the form $F(P) \sqsubseteq G(P)$, where G is a distributive; recalling that a distributive predicate is defined as: if P and P^* satisfy it then $P \sqcap P^*$ does.

Since *RespondsTo* is refinement-closed in both its variables, it follows that there are machine-checkable assertions of the form $SPEC1 \sqsubseteq G1(P)$ and $SPEC2 \sqsubseteq G2(Q)$, for predicates $R1_Q(P)$ and $R2_P(Q)$ where $R1_Q(P) = R2_P(Q) = P \mathit{RespondsTo}\ Q$. The significance of the refinement closure of *RespondsTo* is apparent in that any refinements of the right-hand sides of the assertions given above provably satisfy the left-hand specifications since left-hand sides of these assertions are constant. This construction is not necessarily mechanically efficient due to its generality. Appendix B contains a more efficient version for *RespondsTo* .

Since *RespondsToLive* is not refinement-closed though it is distributive in each of its variables, it follows that it cannot be represented as $SPEC \sqsubseteq G(P)$, with $SPEC$ constant, though it can be represented with the more general $F(P) \sqsubseteq G(P)$. Appendix B also contains a more efficient machine-checkable formulation for *RespondsToLive* .

The checks given in Appendix B are designed to work with the FDR model-checker [For]. FDR (Failures-Divergences Refinement) verifies statements of the form $SPEC \sqsubseteq P$ by checking that all behaviours of P are acceptable behaviours of $SPEC$. The conditions in Appendix B are all of this form.

7. Summary

We have investigated two relations of interest: *RespondsToLive* and *RespondsTo*.

- *RespondsToLive* states that one component Q cannot cause another component P to deadlock whenever P itself is deadlock-free. This relationship is desirable for plug-in components but it is not in general refinement-closed. Interestingly, it is distributive and hence, mechanically verifiable.
- *RespondsTo* is the weakest refinement-closed strengthening of *RespondsToLive* . That is, if Q *RespondsTo* P, then for any refinements P' and Q', it follows that Q' *RespondsToLive* P'. Since *RespondsTo* is refinement-closed, it is mechanically verifiable.

We view the *RespondsTo* relation as characterising responsiveness for plug-in components: Q is a responsive plug-in to P if and only if Q does not cause P to deadlock, and furthermore, no refinement of Q causes any refinement of P to deadlock.

We have used the failures/divergences model for our formulations, though we have assumed divergence-freedom for all processes under consideration. Since we deal with the interplay between non-determinism and blocking, including both graceful termination and deadlock, it is appropriate to use a model which distinguishes these behaviours.

For simplicity, we have assumed that processes synchronise on the entire set of events Σ, though these relations could be generalised for processes which do not synchronise on their entire alphabets.

8. Relations to other work and conclusions

In this paper we have taken an intuitive idea of one component being responsive to another and have shown how a formal definition of such a property and machine-checkable tests may be defined corresponding to this intuition. From the beginning, deadlock and termination are concepts which have been dealt with by process algebraic formalisms. The focus has historically been on analysing behaviour of the combined system resulting from individual behaviours of its components. In contrast, we wish to reason about the behaviour of an individual component resulting from its interaction with other components operating in parallel.

It is generally easier to verify system properties at an abstract level rather than at more concrete implementation ones, but this is useful only if properties verified early on are preserved by refinement. We faced a refinement situation somewhat similar to that of the so-called "refinement paradox" for security properties, which refers to the fact that many security properties are not preserved by refinement. The root cause is that nondeterminism, intended as unbiased specification at the top level, can be refined in an insecure manner. Strong security properties, such as those proposed in [Ros98], eliminate all possible influence through the resolution of nondeterminism. In our case, refinement preservation is fundamental not only from a perspective of retaining correctness during subsequent system development, but also to ensure that a given component operating as part of a component-based system behaves responsively no matter what its behaviour. These considerations caused us to move on from the first definition, which can only guarantee the requirements in certain cases, to the *RespondsTo* definition which covers the general case.

There is an intriguing parallel between our two definitions of responsiveness and Roscoe's formulations of security and fault tolerance properties [Ros98]. Within each pair, the properties have similar but subtly (and importantly) different meanings; one property is refinement-closed but not distributive (security and *RespondsTo*) while the other is distributive but not refinement-closed (fault tolerance and *RespondsToLive*); similar models of FDR checks may be used to verify them. It would be interesting to know whether some wider classification could be made based on such similar pairs of properties.

In previous work [RS01b, RS01a], we investigated an earlier version of our property *RespondsToLive*, where we also gave sufficient (though not necessary) constraints for it to be refinement-closed and mechanically verifiable. As we described in Section 6, existing results dealing with distributive and refinement closure ensure that the notions of responsiveness described here are mechanically verifiable with FDR. Bolton and Lowe [BL03] investigate a class of non-standard refinements, each of which may be expressed in the form:

$$\{(tr, X) \in \mathit{failures}(\mathit{IMPL}) \mid F(X)\} \subseteq \{(tr, X) \in \mathit{failures}(\mathit{SPEC})$$

where $SPEC$ is the specification and $IMPL$ the implementation. When $SPEC$ is taken as P, $IMPL$ as $P \mathbin{||} Q$ and $F(X)$ is $X = J$, this is equivalent to our *RespondsToLive* property and its precursor [RS01b]. Bolton and Lowe's check removes all non-relevant failures (those not meeting $F(X)$)) by interleaving, and adds additional pairs for X satisfying $F(X)$. They judge their approach to be more efficient than ours for larger X, but less efficient for smaller X.

Treharne and Schneider [TS99, TS00] have developed an approach to using CSP in parallel with the event-based B notation [Abr96]. This work is specifically focused on describing control executives which gives it a different emphasis to ours, and makes the important step of integrating two notations. The B specification describes the operations of the system, while the CSP provides a control loop to capture the order of execution for the operations. The individual components can then be developed separately, exploiting the respective strengths of each notation. The basic suitability criterion (provided via a weakest precondition [Dij76] correspondence similar to that of Morgan [Mor90]) is that the loop should not call any B operation outside its precondition.

Another approach for combining different notations based on failures-based semantics is Butler's combined use of B and CSP [But99]. This is similar in many respects to that of Treharne and Schneider, but with event information incorporated in the B. An interesting application of this combined approach is verification of an implementation (deployment) of a security protocol [But02], which goes beyond the more abstract, algorithmic analysis of the protocol typically performed using a single formalism. A different route is taken in the CIRCUS [CSW02] work, where the two specification approaches are combined within a single language.

We note that the work presented here grew out of an earlier initiative to use a combined failures-based formalism to specify a secure database intended to serve remote clients. For reasons of economy of expression, we chose to specify the functional services with a state-based notation, and to specify the security protocol with an event-based notation. Our approach was to treat one component as distinguished in that it required other components to comply with its actions, while at times allowing them some flexibility of provision. Our contribution here has been to define the property of interaction between the components and to show how it could be verified automatically solely using FDR model-checking. Our intention is to incorporate this within a mixed formalism approach, linking our work to existing research on methods integration and theorem proving.

The idea of non-blocking processes arises in a number of other contexts. For example, in extending existing correctness rules for composition in the assume-guarantee setting, Amla et al. [AENT01] define a process as a tuple consisting of variables, initial condition, transition relation and fairness relation. In this context, a rule both sound and complete (for safety and liveness properties) is developed for reasoning about component decomposition. The idea of nondeterministic blocking is not at issue here, and blocking behaviour can trivially be removed by adding transitions to a special state with a self loop. In contrast, we treat blocking as fundamental and undesirable.

Appendix A: Introduction to CSP

We use the syntax and semantics from [Ros98]. The CSP language describes interacting components of systems: *processes* whose external actions are the communication or refusal of instantaneous atomic *events*. All the participants in an event must agree on its performance. The following CSP algebraic operators are used for constructing processes.

$STOP$ is the CSP process which never engages in any event, never terminates (deadlock).

$SKIP$ similarly never performs any action, but instead terminates

$a \rightarrow P$ performs event a and then behaves as process P. The same notation is used for outputs ($c!v \rightarrow P$) and inputs ($c?x \rightarrow P(x)$) of typed values on named channels, with $c.T = \{c.x \mid x \in T\}$.

$P \sqcap Q$ is *nondeterministic* or internal choice. It may behave as P or Q arbitrarily.

$P \,\Box\, Q$ is external or *deterministic* choice. It first offers the initial events of both P and Q to its environment. Its subsequent behaviour is like P if the initial action chosen was possible only for P, and similarly for Q. If P and Q have common initial actions, its subsequent behaviour is nondeterministic (like $\sqcap$).

$\bigsqcap_{x:X} P(x)$ **and** $\Box_{x:X} P(x)$ represent generalised forms of the choice operators allowing indexing over a finite set of indices where $P(x)$ is defined for each x in X. $c?x \rightarrow P$ is shorthand for $\Box_{x:T}\, c.x \rightarrow P$.

$P \parallel_X Q$ is parallel (concurrent) composition. P and Q evolve separately, but events in X occur only when P and Q agree (i.e. *synchronise*) to perform them.

$P \parallel Q$ is parallel composition, with P and Q synchronising on all events, that is, on all of Σ.

$P \mid\mid\mid Q$ represents the interleaved parallel composition. P and Q evolve separately, and do not synchronise on their events.

$P \setminus A$ is the CSP abstraction or hiding operator. This process behaves as P except that events in set A are hidden from the environment and are solely determined by P; the environment can neither observe nor influence them.

$P[[x := y]]$ is the process formed by renaming x to y in P. Whenever P would offer x, this process instead offers y.

Failures/Divergences Model. A process P is modelled as a set of *failures* and *divergences*. The set Σ contains all possible communications events of processes. The set $\Sigma^{\checkmark} = \Sigma \cup \{\checkmark\}$, contains a special event $\checkmark$ that signals that a process has terminated cleanly. A *failure* is a pair (s, X) for s a finite trace of events of $\Sigma^{\checkmark}$, and X a subset of events of $\Sigma^{\checkmark}$. It is understood that whenever $\checkmark$ appears in a trace, it is the last event in the trace. The pair $(s, X) \in \mathit{failures}(P)$ means that P may engage in the sequence s and then refuse all of the events in X. The set X is called a *refusal*. The set $\mathit{divergences}(P)$ is the set of traces on which P can diverge, meaning perform an infinite unbroken sequence of internal events.

Failures and divergences must satisfy certain axioms ensuring well-formedness. The ones for the $\mathit{failures}(P)$ are given below [Ros98]. It is in general true that $\mathit{traces}(P) \cup \mathit{divergences}(P) = \{s \mid (s, \{\}) \in \mathit{failures}(P)\}$. Let $F = \mathit{failures}(P)$.

F1. The set of traces is non-empty and prefix closed, and

$$\mathit{traces}(P) = \{t \mid (t, X) \in F\}$$

F2. If a process can refuse the set X it can refuse any subset of X:

$$(s, X) \in F \wedge Y \subseteq X \Rightarrow (s, Y) \in F$$

F3. Whenever process P can refuse the set X of events in some state then the same state must also refuse the set Y of events that the process can never perform after S:

$$(s, X) \in F \wedge \forall\, a \in Y \bullet (s \frown \langle a \rangle \notin \mathit{traces}(P) \Rightarrow (s, X \cup Y) \in F$$

F4. If a process can terminate, then it can refuse to do anything but terminate:

$$s \frown \langle \checkmark \rangle \in traces(P) \Rightarrow (s, \Sigma) \in F$$

The operators appearing in this paper satisfy semantics-defining equations given below. We omit requirements for divergences.

$$\begin{aligned}
failures(STOP) &= \{(\langle\rangle, X \mid X \subseteq \Sigma^{\checkmark}\} \\
failures(SKIP) &= \{(\langle\rangle, X) \mid X \subseteq \Sigma\} \cup \{(\langle\checkmark\rangle, X \mid X \subseteq \Sigma^{\checkmark}\} \\
failures(a \to P) &= \{(\langle\rangle, X) \mid a \notin X\} \\
&\quad \cup \{(\langle a\rangle \frown s, X \mid (s, X) \in failures(P)\} \\
failures(?x : A \to P) &= \{(\langle\rangle, X) \mid X \cap A = \{\}\} \\
&\quad \cup \{(\langle a\rangle \frown s, X \mid a \in A \wedge (s, X) \in failures(P[a/x])\} \\
failures(P \sqcap Q) &= failures(P) \cup failures(Q) \\
failures(P \,\square\, Q)^{*} &= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in failures(P) \cap failures(Q)\} \\
&\quad \cup \{(s, X) \mid (s, X) \in failures(P) \wedge s \neq \{\}\} \\
&\quad \cup \{(\langle\rangle, X) \mid X \subseteq \Sigma \wedge \langle\checkmark\rangle \in traces(P) \cup Traces(Q)\} \\
failures(P \parallel_X Q)^{\dagger} &= \{(u, Y \cup Z) \mid Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \\
&\quad \wedge \exists\, s, t.(s, Y) \in failures(P) \\
&\quad \wedge (t, Z) \in failures(Q) \\
&\quad \wedge u \in s \parallel_X t\} \\
failures(P \setminus X)^{\ddagger} &= \{(s \setminus X, Y) \mid (s, Y \cup X) \in failures(P)\}
\end{aligned}$$

$*$ $P \,\square\, Q$ may refuse event a only when both may choose to do so. The special event $\checkmark$ must be given special status: if it appears in a trace of P or Q, then everything may be refused.

$\dagger$ $P \parallel_X Q$ can refuse an event in X when either P or Q can because they both have to participate in it. They can independently perform events outside of X, these can only be refused when both P and Q do, much as the empty trace in $P \,\square\, Q$. The special event $\checkmark$ is treated so that termination is at the behest of either. The notation $s \parallel_X t$ for traces s and t represents the set of all interleavings that could arise if P and Q respectively communicate s and t.

$\ddagger$ $failures(P \setminus X)$ has traces which are formed for P with all elements of X stripped out. The refusals are those of P, and at any point, all of X may be refused as well.

Appendix B: Mechanical verification

B1 Mechanically verifying *RespondsTo*

Mechanically checking *RespondsTo* requires comparing the initial events of P and its refusals on the same trace. Consider

$$P^{\sim} = (P \,|||\, P^{*}) \parallel_{H \cup H^{*}} Reg$$

where H is the alphabet of P, and

$$\begin{aligned}
H^{*} &= \{a^{*} \mid a \in H\} \text{ disjoint from } H, {}^{*} \text{ injective} \\
P^{*} &= P[[a := a^{*} \mid a \in H]] \\
Reg &= \square_{a:H}\, a^{*} \to (\square_{b:H}\, b \to (a = b \& Reg))
\end{aligned}$$

This allows P^{*} a step and then P, and then only allows them to proceed when they have performed the same (modulo *) event. That is, $P^{\sim}$ alternates between performing events from a given trace of P and the corresponding trace of the copy P^{*}, starting with the latter.

Now consider $R = P^{\sim} \parallel_J \mathcal{L}_{\Sigma-J}(Q)$, where $\mathcal{L}_{\Sigma-J}(Q)$ is the lazy abstraction of Q with respect to $\Sigma - J$ see [Ros98]. We use this operator to remove from consideration events in the alphabet of Q which are not in J. This abstraction is the process which behaves like Q except that whenever Q can perform an abstracted event the new process has the choice of either not doing it or making it invisible. outside of J that Q can, but which can also refuse any of these events. For the purposes of verification we may (see [Ros98]) identify $\mathcal{L}_{\Sigma-J}(Q)$ with

$$LQ = (Q \parallel_{\Sigma-J} CHAOS(\Sigma - J)) \setminus (\Sigma - J)$$

$P^{\sim}$ performs events in twos, the first of which is an event from P^*, and the second of which is from P , synchronised if in J with the one from LQ. When it deadlocks after an even number of steps this is because P has deadlocked or because P and P^* performed different events: neither of these is interesting to us.

But suppose it refuses the whole of J after an odd number of steps where the immediately preceding event was in $J^* = \{a^* \mid a \in J\}$. Then after some trace of $P \parallel_J LQ$ (doubled into appropriate twos in R) we can reach a state where P^* can perform some $a^* \in J^*$ but P can conspire with LQ to refuse the whole of J. This does not quite characterise the failure of our condition, since for example, P might be able to refuse a while LQ refuses everything else.

What we need to ensure is that in the case where P^* has demonstrated that there is something in $initialsInJ^{\checkmark}(P)$, say a, then $G(P)$ only comes up with refusal sets X not containing a so that $initialsInJ^{\checkmark}(P) - X$ is nonempty: these being the one of interest for the condition.

This is impossible since we cannot get inside P . However, we can achieve the same effect by modifying the *Reg* process above.

$$\begin{aligned} Reg^* = \Box_{a:H}\, a^* \to ((&\Box_{b:H}\, b \to (a == b \& Reg^*)) \\ &\Box\, (a \in J) \& a^{\diamond} \to STOP \end{aligned}$$

where $a^{\diamond}$ is a further separate version of a.

$$P^{\dagger} = ((P \parallel\!| P^*) \parallel_{H \cup H^*} Reg^*)[[a^{\diamond} := a \mid a \in J]]$$

This process again runs P and P^* in the delayed lock-step way that $P^{\sim}$ does, only this time the action of *Reg*, after an odd length trace $s \frown \langle a^* \rangle$, when $a^* \in J$, removes a from the refusal set created by P . Hence if P is refusing X, then $P^{\dagger}$ refuses $X - \{a\}$. Since a is by construction in $initials(P/even(s))$, for $even(s)$ being the trace of the second, fourth, etc, of s, it follows that $(even(s), X - \{a\})$ is a failure of P such that

$$initials(P/even(s)) - (X - \{a\}) \neq \emptyset$$

And, every maximal refusal of $P/even(s)$ with this property appears in this way as P and P^* go through different behaviours within $P^{\dagger}$.

It follows that Q *RespondsTo* P if and only if $P^{\dagger} \parallel_J LQ$ has no deadlock after an odd-length trace whose last member is in J^*, or in other words if and only if it refines

$$\begin{aligned} Spec \quad = \quad & (\Box_{a:J}\, a^* \quad \to \quad (&& (1) \\ & \qquad\qquad (\sqcap_{b:J}\, b \to Spec) \\ & \qquad\qquad \Box \\ & \qquad\qquad (STOP \sqcap (\Box_{b:H-J}\, b \to Spec))) \\ & \Box \\ & (\Box_{a:H-J}\, a^* \to (STOP \sqcap (\Box_{a:H}\, a \to Spec)))) && (2) \\ & \sqcap\, STOP && (3) \end{aligned}$$

The above specification provides three cases: (1) after odd length traces, if the last element is in J, then something in J (the a from $P^{\dagger}$) must be offered, and it does not care whether anything outside of J is offered or refused, (2) after odd length traces, if the last element is not in J, then the specification does not care what events are offered or refused, and (3) after even length traces, deadlock is acceptable since it means that P has reached a state for which its set of initial events is empty.

Notice that since the construction of $P^\dagger$ from P is (like all other combinations of CSP operators) monotonic, and *Spec* is a fixed process, the check $Spec \sqsubseteq P^\dagger$ is refinement closed.

B2 Mechanically verifying *RespondsToLive*

The condition we want to check is that whenever $P \parallel_J Q$ reaches a state in which all of J can be refused, then P has performed a trace s after which it can refuse all of J.

This condition holds of P and Q precisely when it holds of the lazy abstraction $LQ = \mathcal{L}_{\Sigma - J}(Q)$ (see Appendix B1). Notice that $P \sqsubseteq_T (P \parallel_J LQ)$, since LQ restricts P's traces but never contributes any individual events. P *ResponsToLive* Q is true precisely when there is no trace s such that (s, J) is a failure of $P \parallel_J LQ$ but not of P. For a process R, define

$$Tref_J(R) = (R[[a := a, a := e \mid a \in J]] \parallel_{\Sigma - \{e\}} CHAOS(\Sigma - \{e\})) \parallel_\Sigma Onee$$

for e an event outside the alphabets of P and Q, and *Onee* defined:

$$\begin{aligned} Onee = {} & e \to STOP \\ & \quad \Box\ (\Box_{x:\Sigma-\{e\}}\ x \to Onee \end{aligned}$$

$Tref(R)$ can perform any trace of R with the additional possibility of performing an extra e when R can do something in J, after which it must stop. It can refuse anything anytime except for e, which it can only refuse when R can refuse the whole of J. Q *RespondsTo* P on J if and only if

$$Tref_J(P) \sqsubseteq_F Tref_J(P \parallel_J LQ)$$

The above predicate asserts that whenever the right-hand side can refuse e (hence exactly when $P \parallel_J Q$ can refuse all of J), then the left-hand side must be able to refuse e (exactly when P can refuse all of J). Note that P appears on both sides of the refinement, reflecting the fact that *RespondsToLive* is not refinement closed.

References

[Abr96] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[AENT01] Nina Amla, E. Allen Emerson, Kedar Namjoshi, and Richard Trefler. Assume-guarantee based compositional reasoning for synchronous timing diagrams. *Lecture Notes in Computer Science*, 2031:465+, 2001.

[Ars02] Ali Arsanjani. Developing and integrating enterprise components and services. *Communications of the ACM*, 45(10), Oct 2002.

[BL03] C. Bolton and G. Lowe. On the automatic verification of non-standard measures of consistency. In *6th International Workshop in Formal Methods*, Dublin, July 2003.

[BR85] S.D. Brookes and A.W. Roscoe. Deadlock analysis in networks of communicating processes. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series F*, pages 305–324. Springer, 1985.

[But99] M.J. Butler. csp2b: A practical approach to combining CSP and B. In J.M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM99: World Congress on Formal Methods in the Development of Computing Systems*, volume 1, pages 490–508. Springer Verlag, 1999.

[But02] M. Butler. On the use of data refinement in the development of secure communications systems. *Formal Aspects Computer Science*, 14:2–34, 2002.

[CHJK02] I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kisiltan. Specification, implementation, and deployment of components. *Communications of the ACM*, 45(10), Oct 2002.

[CSW02] A. Cavalcanti, A. Sampaio, and J. Woodcock. Refinement of actions in Circus. In *Electronic Notes in Theoretical Computer Science*, volume 70:3. Elsevier, 2002. REFINE 2002.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[For] Formal Systems (Europe) Ltd. *Failures Divergence Refinement*. User Manual and Tutorial. http://www.formal.demon.co.uk/fdr2manual/index.html.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Int] Internet. MasterCraft Integrated Development Framework for Distributed Applications. Tata Consultancy Services. http://www.tcs.com/0_products/mastercraft/index.htm.

[Laz99] R.S. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. DPhil Thesis, Oxford University Computing Laboratory, 1999.

[Mor90] C.C. Morgan. Of wp and CSP. In D. Gries W.H.J. Feijen, A.G.M. van Gasteren and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra*. Springer-Verlag, 1990.

[RD87] A.W. Roscoe and N. Dathi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, December 1987.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[Ros03] A.W. Roscoe. On the expressiveness of CSP refinement checking. Draft, 2003.

[RS01a] J.N. Reed and J.E. Sinclair. Combining Action Systems and CSP: Plug-in specifications view. *Electronic Notes in Theoretical Computer Science*, 40, Dec 2001.

[RS01b] J.N. Reed and J.E. Sinclair. Compatibility conditions for combining independent specifications. In *FASE 2001, Fundamental Approaches to Software Engineering*, pages 45–59, Genoa, Italy, 2001. Springer.

[TS99] H. Treharne and Schneider S. Using a process algebra to control B operations. In K. Araki, A. Galloway, and K. Taguchi, editors, *Integated Formal Methods 1999*, pages 437–456, York, UK, June 1999. Springer Verlag.

[TS00] H. Treharne and S. Schneider. How to drive a B machine. In J.P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *Proceedings of ZB2000*, LNCS 1878, pages 188–209, York,UK, September 2000. Springer Verlag.