DOI 10.1007/s00165-007-0032-9
BCS © 2007
Formal Aspects of Computing (2007) 19: 303–319

Formal Aspects of Computing

Responsiveness and stable revivals

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

[1]Department of Information Technology, Armstrong Atlantic State University, Savannah, GA, USA
[2]Computing Laboratory, University of Oxford, Oxford, UK
[3]Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK

Abstract. Individual components in an inter-operating system require assurance from other components both of appropriate functionality and of suitable responsiveness. We have developed properties which capture the notion of non-blocking responsive behaviour, together with machine-based checks implemented in the CSP model-checker, FDR. In this paper we illustrate the use of our responsiveness properties with a small example, and provide a detailed comparison to related work in CCS. This work has led to the discovery of a new semantic model for CSP with respect to which such properties are fully abstract. We present the new stable revivals model and discuss implications for responsiveness checking.

Keywords: Responsiveness; CSP; Semantic model; Model-checking

1. Introduction

The historical focus of formal verification of component-based systems has been to reason about the behaviour of a system based on the collective behaviour of its components. Typical inference rules allow the derivation of properties of an entire system from individual properties of its components. However in certain sorts of systems developed today, such as distributed services, it is more natural to reason from the point of view of an individual component. In this component-centred world each component may provide a particular service and may also depend on tasks performed and services provided by other components. A component requires assurance that other components used as sub-contractors really do provide the services it requires. It can also provide a specification of its own services to others. This picture of individual responsibility via inter-component contracts is very different from one of overall design and central control where a full "bird's eye" view of the overall system can be specified. We wish to reason about the effect on the behaviour of one component resulting from its interactions with other, probably independently developed, components. Our approach is to view verification from the perspective of each single component.

In order to make decisions about the suitability of available services, an autonomous component would need to be satisfied that appropriate functionality was offered. It would also need to know that it could interact suitably with the sub-contractor, that is, they can agree on behaviour to make progress towards the required goal.

Our work has initially focused on a behavioural aspect of component compatibility, using the process algebra CSP to investigate the concept of responsiveness. A fundamental requirement is that a requesting (or client) process, P, should not be blocked by a service process, Q, not responding to it when expected. The client, P, simply

Correspondence and offprint requests to: J. E. Sinclair, E-mail: jane@dcs.warwick.ac.uk

requires that Q never prevents it from making progress whenever P demands. This is not equivalent to demanding that the parallel combination of the two components be deadlock free. Characterising this interpretation of responsiveness gives rise to a property which is weaker than deadlock freedom and which has proved surprisingly difficult to capture formally. We illustrate this difference between the two behavioural concepts below.

In [RSR04], we defined *RespondsTo*, which captures the property of responsiveness for distributed components and introduced conditions suitable for model checking the property with FDR [FSE97]. The current paper reviews the idea of responsiveness and provides a case study to show how the property can be used in practice. We suggest how the event-based view might be linked to a more data-rich description of the components. The On-line Shopping Network example used here is a typical instance of a system where both aspects are important. It allows a direct comparison with some related work on a property known as stuck-freeness formulated in CCS by Fournet et al. [FHRR04]. This investigation has yielded a new model for CSP, and additionally, shed significant new light on the relationship between CCS and CSP.

In Sect. 2 we introduce relevant CSP concepts and motivate the responsiveness property with some small examples. Section 3 gives the definitions for our concept of responsiveness. The case study is presented in Sect. 4, showing how the responsiveness properties may be used and model-checked. We also indicate a possible way of linking to state-based process specifications, appropriate for describing functionality using the B method. Section 5 presents detailed discussion and comparison of the CSP responsiveness work and the CCS stuck-freeness property, leading to the presentation of the new stable revivals model for CSP. Section 6 presents conclusions and relations to other work.

2. An introduction to CSP and Responsiveness

CSP [Hoa85, Ros98] models a concurrent system as a collection of communicating processes, describing their patterns of interaction by means of atomic events. Communication is synchronous: an event happens when all participants agree to it. In the following sections, P and Q denote processes. The set Σ contains all possible events, that is, communications for processes in the universe of consideration. An overview of the relevant CSP syntax is given in Appendix A.

2.1. Introductory examples

Suppose process P makes a request to server Q after which P is happy to deal with either of two possible responses. This may be represented in CSP using the external (deterministic) choice operator, $\Box$

$$P = request \rightarrow (response1 \rightarrow P \,\Box\, response2 \rightarrow P)$$

Suppose, upon receiving a request, Q offers only the service indicated by $response1$.

$$Q = request \rightarrow response1 \rightarrow Q$$

P will regard Q as a suitable service because Q can supply one of the possible acceptable patterns. In this case, the parallel combination $P \parallel Q$ runs successfully, making progress without deadlocking on events they have in common from $\Sigma = \{request, response1, response2\}$. As Q is able to cooperate with one of the possible patterns of interaction set out by P, we regard Q as being responsive to P. In this particular case, P is also responsive to Q.

The situation would be very different if P made an internal (nondeterministic) choice between the replies. This can be represented as:

$$P = request \rightarrow (response1 \rightarrow P \;\sqcap\; response2 \rightarrow P)$$

In this case, Q should no longer be regarded as responsive to P, since if P chooses $response2$, it would be forever blocked. From Q's perspective, it is also the case that P is not responsive to it.

In both of these cases, checking whether $P \parallel Q$ deadlocks tells us what we need to know. However, deadlock may be introduced by P with Q still willing to be responsive. For instance, suppose P is a process which may either request a response or may decide to terminate (with *SKIP* representing clean termination).

$$P = (request \rightarrow response1 \rightarrow P) \;\sqcap\; SKIP$$

Process P may request Q's services one or more times but may also choose to terminate. Q is responsive to P because it is willing to cooperate whenever P requires it. However, $P \parallel Q$ may deadlock. It is also apparent that P is not responsive to Q, demonstrating that responsiveness is not symmetric.

In the previous example, deadlock in P did not prevent Q from being responsive to it. It is also possible that in some situations Q can deadlock whilst still remaining responsive. For example, P may be a process which simply requires a single service and then terminates, or perhaps continues with its own behaviour, never again requesting cooperation from Q. In this case, it is of no relevance to P what happens to Q after their joint interaction is concluded. Hence Q may be regarded as responsive to P regardless of possible deadlock introduced by Q after fulfilling its obligations to P. This is in keeping with the component-centred view in which P cares about its own progress but does not need assurance about the wider system beyond this.

These examples provide a general idea of the responsiveness concept. Following an overview of CSP, we present a formal definition of responsiveness in Sect. 3.

2.2. CSP semantic models

A number of related semantic models have been set out for CSP, each based on different observable process behaviours. The standard one is the failures/divergences model. A *failure* of a process is a pair of the form (s, X) where s is a finite trace (sequence of observable events the process can perform) and X is a set of events all of which may be refused by the process after it has performed s. The set X is called a refusal set. A *divergence* is a trace after which the process can perform an infinite sequence of internal actions.

In our work on responsiveness we have so far restricted our consideration to divergence-free processes. That processes do indeed meet this restriction can be ensured by checking for divergence using the FDR model checker. Under the assumption of divergence-freedom, the failures/divergence model, $\mathcal{N}$, is essentially the same as the stable failures model, $\mathcal{F}$. A *stable failure* is a failure from which no internal events are possible. Our properties and the theorems relating them [RSR04] have been developed in the stable failures model. In fact, we refer to $\mathcal{F}$ with the addition of the special clean termination symbol, $\checkmark$, which is described in detail by Roscoe [Ros98].

Whilst detailed consideration of semantics is not necessary in order either to use CSP or to apply our responsiveness condition, it is appropriate to note here the model with respect to which the property is defined and justified. Our work with responsiveness has also led to the discovery of a new semantic model, and we return to this in Sect. 5.

3. Defining responsiveness

In the following descriptions, P and Q are processes, with P regarded as the requesting (client) process which requires Q to respond in a non-blocking manner. J denotes the shared alphabet of P and Q. We assume that there is no other member of Σ which both P and Q are capable of communicating – the more general case can be reduced to this one by applying it to the *lazy abstractions* $\mathcal{L}_{\Sigma-J}(P)$ and $\mathcal{L}_{\Sigma-J}(Q)$ which form the views of P and Q relevant to each other —see [Ros98], Chap. 12. Thus $P \underset{J}{\parallel} Q$ will, in this paper, always be the same as Hoare's *alphabetised* parallel. We will refer to their alphabets as αP and αQ, so that $J = \alpha P \cap \alpha Q$.

For Q to be suitably responsive to P, whenever P requires co-operation from Q in an event $j \in J$, Q must be willing to participate. Q must not cause deadlock, but P may behave as it chooses. If P is happy to engage in any one of a set of joint events, Q must be willing to engage in at least one of these.

3.1. Failures-based formulation of responsiveness properties

Our formal definition of responsiveness is given in CSP over $\mathcal{F}$. It requires that, at any point in the joint execution of P and Q, if P demands participation in a set of joint events, Q complies for some non-empty subset of the events. In this definition:

- $\checkmark$ is the special termination event on which all CSP parallel operators effectively synchronise (distributed termination);
- $J^{\checkmark}$ is the joint alphabet with the $\checkmark$ added;
- S^* is the set of all finite sequences whose elements are members of set S;

- $initials(P)$ is the set of all initial events in which P may engage;
- P/s is the process which behaves as P would after execution of trace s;
- $s \upharpoonright A$ is the subsequence of s formed by restricting s to elements of set A.

Definition 1 For processes P and Q with joint alphabet J, we say that Q *RespondsTo* P iff for all $s \in (\alpha P \cup \alpha Q)^*$, $X \subseteq \alpha P$:

$$(s \upharpoonright \alpha P, X) \in failures(P) \wedge (J^{\checkmark} \cap initials(P/s)) - X \neq \{\} \Rightarrow$$
$$(s \upharpoonright \alpha Q, (J^{\checkmark} \cap initials(P/s)) - X) \notin failures(Q)$$

The definition of responsiveness is nontrivial due to the effect of nondeterminism, which may occur in both P and Q. The intuition is that, given trace t and failure (t, X) of P, the set $(J^{\checkmark} \cap initials(P/t)) - X$ describes an event set in which P may demand participation. Thus for Q to be responsive to P, Q must not be able to refuse all events in that set at the corresponding point in its execution trace, that is, after any trace u such that $u \upharpoonright J^{\checkmark} = t \upharpoonright J^{\checkmark}$. Further explanation and discussion of this and additional examples can be found in [RSR04]. For example, consider

$$P = request \rightarrow (response1 \rightarrow P \sqcap response2 \rightarrow P)$$
$$Q = request \rightarrow response1 \rightarrow Q$$

The pair $(\langle request \rangle, \{response1\})$ is a failure of P, since P may nondeterministically decide to insist upon *response*2 after the *request* event. The definition requires that Q should not refuse $\{response2\}$ at this point. However, $(\langle request \rangle, \{response2\})$ is a failure of Q. Hence Q does not respond to P.

If we instead define P as:

$$P = (request \rightarrow response1 \rightarrow P) \sqcap SKIP$$

then it is clear that P may choose to terminate and require no further participation from Q. However, if P does decide to interact with Q, the participation it can demand is:

$\{request\}$ after $\langle\rangle$
$\{response1\}$ after $\langle request \rangle$
andsoon

These are not failures of Q, and so Q is responsive to P.

3.2. Responsiveness for deterministic processes

If we are dealing with deterministic processes the property of responsiveness can be captured more simply. In [RSR04] this is formalised as the property *RespondsToLive*.

Definition 2 Q *RespondsToLive* P means that for every trace s

$$(s, J^{\checkmark}) \in failures(P \underset{J}{\parallel} Q) \Rightarrow (s \upharpoonright \alpha P, J^{\checkmark}) \in Failures(P)$$

This says that if $P \parallel Q$ can refuse the whole of the joint alphabet then P itself must be refusing it. For example, the definition can be applied to the deterministic processes:

$$P = request \rightarrow (response1 \rightarrow P \,\Box\, response2 \rightarrow P)$$
$$Q = request \rightarrow response1 \rightarrow Q$$

The parallel combination $P \parallel Q$ behaves as Q which never refuses the whole of their joint alphabet. Hence Definition 2 is trivially satisfied.

This definition is not in general suitable for nondeterministic processes and would not necessarily be preserved by refinement. Hence, *RespondsTo* should be used if processes may be nondeterministic.

In [RSR04] we examined the relationship between the two definitions and established that *RespondsTo* is the weakest refinement-closed strengthening of *RespondsToLive*. (A binary property, H, is refinement-closed if whenever $H(P, Q)$ holds, then $H(P', Q')$ also holds for all refinements $P \sqsubseteq P'$, $Q \sqsubseteq Q'$.)

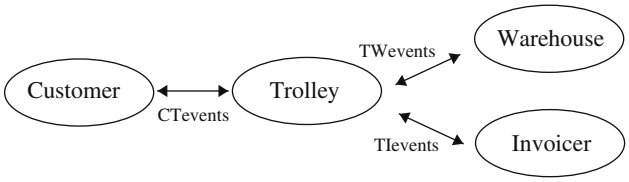

Fig. 1. On-line Shopping Network

3.3. Machine-based verification

An important aspect of our responsiveness properties is that they are both mechanically verifiable. Automatic refinement checking for CSP is provided by the FDR model-checker [FSE97]. A process P is a refinement of process S (written $S \sqsubseteq P$) if any possible behaviour of P is also a possible 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 freedom. A wide range of correctness conditions can be encoded as refinement checks between processes. We have shown [RSR04] that both properties *RespondsTo* and *RespondsToLive* may be formulated as machine-checkable assertions suitable for verification as refinements. These results have been generalised by Roscoe [Ros03] who has described techniques for translating more-or-less arbitrary predicates on a process into refinement checks. The FDR check for *RespondsTo* is used in the verification of the Shopping Network example in the following section and the approach is described further there.

Once responsiveness has been verified at the specification level the components may be refined independently in the usual way. It is significant that responsiveness is preserved by refinement, and thus any refinements of the original specifications will be guaranteed to have the same relationship.

4. A Simple On-line Shopping Network

The On-line Shopping Network presents a situation in which components interact in a pairwise fashion to provide a service to the customer. Each component may be developed separately, but the interaction between the components must ensure that the network provides suitable functionality and that the customer is not blocked by the network.

This case study is useful not only in illustrating the use of our own property, but also in allowing a comparison between our work on responsiveness and related work. In particular, in Sect. 5, we discuss the connection between responsiveness and the idea of *stuck-freeness* proposed by Fournet et al. [FHRR04], who have also applied their work on contract-checking in component-based systems to an on-line shopping scenario. The comparison yields interesting insight into some fundamental differences between CSP and CCS.

Another aim of the case study is to set out some provisional thoughts on how behavioural interface descriptions can fit together with a more state-oriented approach for modelling data-rich aspects of component functionality. As this type of specification is not best achieved in a process algebra, we suggest by example a strategy for incorporating results from the area of integrated formal methods.

4.1. Specification of interactions in the network

Figure 1 shows the components of our system (a much-simplified version of the problem domain, showing basic interactions and considering the behaviour of a single customer and trolley only). The Customer interacts directly with the Trolley component via a set of events we have called *CTevents*. The trolley acts as an intermediary with the rest of the system, itself interacting with both the Warehouse and an Invoicer. The sets of events for the Trolley's interaction with these components are *TWevents* and *TIevent* respectively.

We provide stylised FDR scripts for this network consisting of the four processes: *Customer*, *Trolley*, *Warehouse*, and *Invoicer*. The event sets describe the shared channels between the components. For data type T

and channel c, $c.T$ is the set of all events associated with c. In order to describe these data-passing events we introduce the following abstract types.

ItemType	items that may be selected
TType	type for trolleys
InvType	type for invoices
Flag	for indicating success or failure

The intention is to keep such types as abstract as possible at this level. For example, the trolley data type is likely to be a structure built from *Items*, but this is not relevant here.

4.1.1. The Customer component

The events associated with the Customer are:

$$\begin{aligned} CTevents = \{&open, close, viewTrolley, checkOut, cancel\} \\ &\cup addItem.ItemType \cup removeItem.ItemType \cup getTrolley.TType \\ &\cup addAck.Flag \cup remAck.Flag \cup invoice.InvType \end{aligned}$$

After opening the transaction, the Customer may choose to add an item to the Trolley, or to remove an item, and receives an acknowledgement; it may ask to view the current contents of the Trolley; it may also choose either to check out and receive an invoice or to simply abandon the session. The specification captures this as a nondeterministic choice between the options.

$$Customer = open \rightarrow Customer1$$

$$\begin{aligned} Customer1 = \; & (\sqcap_{item:ItemType}\, addItem!item \rightarrow addAck?flag \rightarrow Customer1) \\ \sqcap\; & (\sqcap_{item:ItemType}\, removeItem!item \rightarrow remAck?flag \rightarrow Customer1) \\ \sqcap\; & checkOut \rightarrow invoice?xx \rightarrow close \rightarrow SKIP \\ \sqcap\; & cancel \rightarrow close \rightarrow SKIP \\ \sqcap\; & viewTrolley \rightarrow getTrolley?tt \rightarrow Customer1 \end{aligned}$$

We wish to make the interface at this level as abstract as possible, ignoring data-specific aspects which are not relevant to it. This can be accomplished by using nondeterminism to abstract away from nonessential details. It is possible that further elaboration could be provided at a more detailed level of development. As long as that description can be shown to be a refinement of the original, responsiveness will be maintained. However, generality in the specification of an interface may have consequences in other areas. For example, this version of the Customer specification allows the *removeItem* event to be available for any item, whether or not it is present in the Trolley. This works only in conjunction with a Trolley which deals sensibly with requests to remove nonexistent items. Presenting a more restricted choice of items to the Customer would be a refinement of this, but needs additional state information in the Customer, or further communication with the Trolley. This starts to touch on interesting aspects of the interplay between the interface and the state-based views of components.

4.1.2. The Trolley component

In addition to its interactions with the Customer, the Trolley invokes the services of a Warehouse and an Invoicer. It communicates with the Warehouse on the set *TWevents*.

$$\begin{aligned} TWevents = \{&open, close, viewTrolley, checkOut, commitReserve, cancelReserve, cancel\} \\ &\cup reserveItem.ItemType \cup cancelItem.ItemType \\ &\cup resAck.Flag \cup cancelAck.Flag \end{aligned}$$

$$TIevents = processOrder.TType \cup processInvoice.InvType$$

The Trolley specification allows external choice between the events the Customer may choose. It handles each of these appropriately, passing on requests to the Warehouse or the Invoicer as necessary. Replies from these two

components are then relayed to the Customer.

$$Trolley = open \rightarrow Trolley1$$

$$\begin{aligned}
Trolley1 = \;& addItem?item \rightarrow reserveItem!item \rightarrow resAck?flag \rightarrow addAck!flag \rightarrow Trolley1 \\
\Box\; & removeItem?item \rightarrow cancelItem!item \rightarrow cancelAck?flag \rightarrow remAck!flag \rightarrow Trolley1 \\
\Box\; & checkOut \rightarrow commitReserve \rightarrow \\
& (\sqcap_{tt:TType}\, processOrder!tt \rightarrow processInvoice?xx \rightarrow invoice!xx \rightarrow close \rightarrow SKIP) \\
\Box\; & cancel \rightarrow cancelReserve \rightarrow close \rightarrow SKIP \\
\Box\; & viewTrolley \rightarrow \\
& (\sqcap_{tt:TType}\, getTrolley!tt \rightarrow Trolley1)
\end{aligned}$$

Again, nondeterminism is used to abstract the details of the contents of the Trolley. In this specification an item of *TType* appears to be selected at random, for example, in response to a *viewTrolley* request. This is sufficient for the interface specification as in this case the flow of control is not affected by the actual value. A state-oriented description can elaborate these details by providing a different view of the Trolley (see Sect. 4.4).

4.1.3. The Warehouse component

The Warehouse reserves or releases items (as directed by the Trolley).

$$\begin{aligned}
Warehouse = \;& reserveItem?item \rightarrow \\
& (\sqcap_{flag:FLAG}\, resAck!flag \rightarrow Warehouse \\
\Box\; & cancelItem?item \rightarrow \\
& (\sqcap_{flag:FLAG}\, cancelAck!flag \rightarrow Warehouse \\
\Box\; & commitReserve \rightarrow Warehouse \\
\Box\; & cancelReserve \rightarrow Warehouse
\end{aligned}$$

The nondeterministic choice of reply value from the Warehouse can be used to indicate, for example, whether a requested item is available or not. Again, the actual value will be determined by examination of the actual state of the Warehouse. In contrast to the Customer and the Trolley, which act as threads terminating when their business is done, the Warehouse is modelled here as an ongoing process which recurses to await the possibility of future requests.

4.1.4. The Invoicer component

The Invoicer takes a request to process an order, and responds with an invoice. The way in which the invoice is calculated from the order is another area of abstraction.

$$\begin{aligned}
Invoicer = \;& processOrder?xx \rightarrow \\
& (\sqcap_{yy:InvType}\, processInvoice!yy \rightarrow Invoicer
\end{aligned}$$

As with the Warehouse, the Invoicer recurses repeatedly, acting as a server always ready to accept requests.

4.1.5. The Shopping Network

The Shopping Network is made up of the Trolley, Warehouse, and Invoicer, with pairwise communication on their respective shared channels:

$$ShopNet = (Trolley \underset{TWevents}{\parallel} Warehouse) \underset{TIevents}{\parallel} Invoicer$$

4.2. Verifying responsiveness

The FDR model-checker [FSE97] verifies process refinement in the failures-divergences model. In general, we can discover whether some behavioural property holds for some process P by checking the refinement $SPEC \sqsubseteq P$ where $SPEC$ captures the desired property. However, *RespondsTo* is not a simple behavioural property of this kind (it is not distributive) and so a different approach is needed. We make use of our previous work on developing machine-checkable assertions corresponding to *RespondsTo* [RSR04, Ros03] to generate the necessary proof obligations in the form of refinement statements.

Appendix B contains a summary of the FDR check which will establish whether Q *RespondsTo* P. There, H refers to the alphabet of P and J is the set of shared events for P and Q.

To show that the Shopping Network is responsive to the Customer we check the *RespondsTo* property to see whether it can make progress on *CTevents*. Other communications in the system are abstracted using hiding (we discuss this further in Section 6). That is, with

$$\begin{aligned} P &= Customer \\ Q &= ShopNet \backslash (TWevents \cup TIevents) \\ J &= CTevents \\ H &= CTevents \end{aligned}$$

the FDR check shows that Q *RespondsTo* P. Relationships between subcomponents can also be investigated. For example, with

$$\begin{aligned} P &= Trolley \\ Q &= Warehouse \\ J &= TWevents \\ H &= CTevents \cup TWevents \cup TIevents \end{aligned}$$

we can again verify that Q *RespondsTo* P.

Examples from this case study also underline the fact that responsiveness is not symmetric and that it is not equivalent to deadlock-freedom. For instance, although the Warehouse is responsive to the Trolley, the Trolley is not responsive to the Warehouse. Investigating their behaviour in parallel shows that

$$Trolley \underset{TWevents}{\parallel} Warehouse$$

can deadlock.

Refinement-closure of *RespondsTo* allows developers of *Trolley*, *Warehouse*, and *Invoicer* to refine their implementation without worry that a modified component (satisfying standard refinement rules) would cause the overall system to be non-responsive to customers. Significantly, this offers component-side development which preserves responsiveness.

The Shopping Network might use distributed services, for example, the trolley might search dynamically for the best provider of individual items. If specifications of behaviour for warehouse services are published, the trolley could validate them on-the-fly in order to determine if their behaviours were responsive. Indeed, the warehouse and trolley could exchange behavioural specifications and negotiate before committing to interaction. Importantly it is *not* necessary to check the whole network to verify responsiveness, only the relevant pairwise interactions (as discussed in [RSR04]).

4.3. Detecting a fault

If the interface specification of a component does not meet the requirements of the client, the responsiveness check will fail. Suppose the Warehouse does not provide an acknowledgement when an item is removed from the trolley. This is the case in FaultyWarehouse.

MACHINE *Type*
SETS $ItemType;\ \ Flag = \{success, failure\};\ \ \ldots$
CONSTANTS $TType\ \ldots$
PROPERTIES $TType = \mathrm{bag}\ ItemType\ \ldots$

MACHINE *Trolley*
SEES *Type*
VARIABLES $trolley,\ citem,\ cflag,\ \ldots$
INVARIANT $trolley \in TType\ \wedge\ citem \in ItemType\ \wedge\ cflag \in Flag\ \wedge\ \ldots$
INITIALISATION $trolley = [\![]\!]\ \ ||\ \ citem :\in ItemType\ \ ||\ \ \ldots$
OPERATIONS

$open\ =\ skip;$

$addItem(item : ItemType)\ =\ citem := item;$

$ii \longleftarrow reserveItem\ =\ ii := citem$

$resAck(flag : Flag)\ =$
IF $flag = success$
THEN $trolley := trolley \uplus [\![citem]\!]\ \ ||\ \ cflag := flag$
ELSE $cflag := flag;$

$ff \longleftarrow addAck\ =\ ff := cflag$

$tt \longleftarrow processOrder\ =\ tt := trolley;$

$\vdots$

Fig. 2. Part of a B specification for the Shopping Network

$$
\begin{aligned}
&FaultyWarehouse = \\
&\quad reserveItem?item \rightarrow \\
&\quad (\sqcap_{flag:FLAG}\ resAck!flag \rightarrow FaultyWarehouse \\
&\ \Box\ cancelItem?item \rightarrow FaultyWarehouse \\
&\ \Box\ commitReserve \rightarrow FaultyWarehouse \\
&\ \Box\ cancelReserve \rightarrow FaultyWarehouse
\end{aligned}
$$

This component blocks the Trolley which expects an acknowledgement. Checking with FDR shows that FaultyWarehouse does not respond to Trolley, and counterexamples are generated.

4.4. A state-based view of the Shopping Network

In this section we outline a way in which our CSP interface specification focusing on behavioural requirements might be aligned with a state-based specification of a component focusing on functional requirements. Our work at this stage is preliminary. However it suggests how integration between these two views of a component could be soundly achieved, pointing to existing work in the area of integrated formal methods.

We choose B Abstract Machine Notation [Abr96] to describe the state-based view of a component. This decision is based partly on the availability of existing, well-developed tool support for the notation, and partly on the fact that a number of very useful results for B/CSP integration have already been obtained (we refer to these later).

Our view at this stage is that each CSP component can be associated with a B abstract machine (the machine is the basic component of a B specification, maintaining state components and defining operations on the state). For example, the Trolley process description is augmented by the Trolley machine shown in Fig. 2. The VARIABLES section of the machine introduces the state components which the Customer machine maintains. The INVARIANT part states important properties relating to the variables (such as their types) and the INITIALI-

SATION section describes initial values for the state variables. The OPERATIONS part of the machine includes a description for each of the events offered by the Trolley interface. These echo the input/output structure from the interface description and provide further detail which may (wholly or partially) resolve nondeterminism in the interface. An event such as *open* which has no effect on the state has been represented by *skip* (in this context, *skip* is the operation which leaves all variables unchanged).

Most events match operations which are associated with a change in state, for instance, the *remAck* operation has an input which indicates whether or not it is possible to reserve an item: if it is, then that item can be added to the Trolley. Values which are passed along in the CSP (such as the item input to *addItem*, output by *reserveItem*) are coded in a routine way by introducing a B variable (in this case, *citem*) to store the value. Some type information has to be shared, as in the channel types. This information has been gathered into a separate B machine which is then made available to the relevant component machines by means of the SEES section. A machine which SEES another may refer to the state components of the seen machine, but does not change their values.

Simply writing down "views" in two different notations begs many questions about the semantic connection between them and the compatibility of the two. It is beyond the scope of this paper to discuss the issues involved in detail, but there are some interesting results which can provide a sound foundation for our approach. The basic semantic link between the two notations can be made via the work of Morgan [Mor90] and Butler [But92] who provide a weakest precondition characterisation of traces, failures and divergences of action systems. This has been applied in the context of B machines by Schneider and Treharne [TS00]. The conditions they set out for establishing compatibility of a CSP controller for a B machine are basically what are required here: we too need to know that B operations are not called outside their preconditions (which would introduce divergence). We also need to consider the possibility of deadlock introduced by "unsuitable" outputs from the B description. Schneider and Treharne [ST05] have developed a number of techniques for verifying divergence-freedom and deadlock-freedom between a CSP process and a B machine, and they present a number of very useful theorems towards the verification of these properties. Their technique of attaching conditions to CSP channels in a rely/guarantee fashion could be very useful in the context of an interface specification, allowing the specification to include information in the form of predicates about what it guarantees to provide or about restrictions it must place on the inputs it accepts.

Although there are many similarities between the way in which we want to reconcile different views of a system and the approach to integration taken in Schneider and Treharne's B||CSP work, there are also some differences both in context and detail. However, the general aspects of integration are similar enough that we should be able to apply their results to this aspect of our work.

The CSP and B specifications provide two views of the same components, providing complementary (sometimes overlapping) information. Together they provide the blueprint for the development of the component. The interface specification can be made available as the external contract which the component guarantees to fulfil. If the compatibility of each B machine/CSP interface pair is established, and if the required responsiveness property between interfaces is also demonstrated, then independent, component-wise refinement can be undertaken. If I_1 and I_2 are interface specifications relating to machines M_1 and M_2 respectively, then if:

$$I_2 \ \mathit{RespondsTo} \ I_1$$

and

$$I_1 \sqsubseteq I_1 \parallel M_1$$

and

$$I_2 \sqsubseteq I_2 \parallel M_2$$

it follows that, since *RespondsTo* is refinement-closed:

$$(I_2 \parallel M_2) \ \mathit{RespondsTo} \ (I_1 \parallel M_1)$$

Again, the motivation is to capture the different aspects in of the specification in an appropriate notation whilst generating proof obligations that can be discharged with existing tool support.

5. Comparison of models

Since doing our original work on the topic of responsiveness, some similar work by Fournet et al. has appeared [FHRR04]. That work was based on a slightly different motivation, namely ensuring that a network of processes does not reach a state from which no further progress can be made while one of them is still requesting something from another. Comparing our work and theirs provides a fascinating insight into the relative qualities of CCS and CSP for specification, as well as illustrating their similarity.

The intention in [FHRR04] is that a combination does not *terminate* leaving one partner hanging. They call the absence of such behaviour *stuck-freeness*. It is noteworthy that this is not really an issue in CSP thanks to the termination signal ✓: the distributed termination condition of CSP means that the network can only seem to have terminated when they both actually have. This simply results from Hoare's decision to separate semantically between deadlock and successful termination: a stuck combination will appear as deadlock, whereas a pair that has terminated normally will have signalled ✓.

It follows that the absence of the type of behaviour identified as bad in [FHRR04] follows from a standard check for deadlock freedom (naturally, permitting processes to do nothing further after ✓).[1] This is at the expense of signalling termination via ✓, but that seems to us to be a distinction worth making.

The reason why simple termination-based reasoning will not work for *RespondsTo* is that we forbid some behaviours that are not final. We forbid one process from refusing another even when one, other, or even both processes have other things they can do (although internal events may occur). So we mind even if the refusing process has the potential, via other actions external to the binary parallel we are considering, to do more things and then reach a position where it can now satisfy its partner's request.

Nevertheless the way [FHRR04] chooses to address their issue is remarkably similar to the way we have addressed ours. They specify that the network N never reaches a state in which no further action can happen (i.e., it is deadlocked) but some $P \in N$ is still offering communications to another $Q \in N$. Over a pair of processes $P \mid Q$ this is conceptually equivalent to saying that any failure of our *RespondsTo* condition (in either direction) only occurs when either P or Q has some alternative action to the interactions in this parallel.

Just as we, in [RSR04], observed that *RespondsToLive* is not refinement closed, they observe that their condition cannot be specified in a refinement-closed manner over the failures model. While our reaction was to strengthen the condition to the weakest refinement-closed one which implied the original, namely *RespondsTo*, theirs was to devise a special equivalence over processes to support it. They call this *conformance* and in it two processes are equivalent if they have identical behaviours of the form (s, X, Y), in which (s, X) is a failure where, in the same stable state which witnesses the failure, every event from Y is available. They restrict Y to be of size 0 or 1. Necessarily, of course, $X \cap Y = \emptyset$.

We make two observations about the conformance equivalence.

- Firstly, if the restriction to $\mid Y \mid \leqslant 1$ were removed, one gets a different congruence equivalent to the *Ready-Sets* model of Olderog and Hoare [OH83]. In that, processes are associated with sets of pairs (s, A) in which s is a trace and A is the set of events which are on offer in some stable state reachable on s. In the absence of the $\mid Y \mid \leqslant 1$ assumption, every triple (s, X, Y) extends to a maximal one in which $X \cup Y = \Sigma$, and it is clear that the two models will then be the same identifying Y with the ready set.
- Secondly, conformance can be developed into a model which is fully abstract with respect to properties like stuck-freeness and precise operational characterisations of *RespondsTo*.

The stable revivals model

In order to turn the idea of conformance into a CSP model we separate the two cases of $Y = \emptyset$ (only necessary for deadlock traces) and $Y = \{a\}$. The latter can be represented as a triple (s, X, a) for $a \notin X$. Since the a represents *revival* from the stable failure represented by (s, X), that is what we shall call the triple.

On the basis (already adopted in [Ros98] relating to the stable failures model) that it is always a good idea to know a process's traces[2] for reasons of safety specification, our new model $\mathcal{R}$ equates a process with three components, respectively

[1] There would be one difference: the deadlock check would regard a state in which every single component process has individually deadlocked without terminating as incorrect even though there is no stuck-ness.

[2] It is possible to get a compositional version of either this model or the stable failures model (see [Ros98]) without the trace component provided one omits the CSP interrupt operator $\triangle$. For this reason, the full abstraction result quoted below is only true for the language including this operator.

- The finite traces T (a prefix-closed nonempty subset of Σ^*).
- The deadlock traces D (a subset of T).
- The revivals R, namely triples of the form (s, X, a) where $s \in \Sigma^*$, $X \subseteq \Sigma$ and $a \in \Sigma - X$, such that R1: $s \frown \langle a \rangle \in T$, R2: $(s, X, a) \in R$ and $Y \subseteq X$ implies $(s, Y, a) \in R$, and R3: $(s, X, a) \in R$ and $b \in \Sigma$ implies that either $(s, X, b) \in R$ or $(s, X \cup \{b\}, a) \in R$.

This yields a model which is a congruence for CSP and which yields the natural fixed point under subset-least fixed points, like $\mathcal{F}$. It is straightforward to recover the $\mathcal{F}$ representation of any process from the $\mathcal{R}$ one: the new one is strictly less abstract.

The most interesting point in it being a congruence arises in hiding: the triple $(s, X, a) \in revivals(P)$ only gives rise to a revival of $P \setminus Y$ if $Y \subseteq X$ (because, analogously with the usual CSP hiding operator, $P \setminus Y$ is not stable unless P refuses Y). It follows that $a \notin Y$ and therefore is not hidden – something which would have caused a problem as we would have lost our next step.[3]

That this equivalence is weaker than ready sets is demonstrated by the following example. Let $\Sigma = \{a, b\}$ and let

$$P = (a \rightarrow Stop) \sqcap (b \rightarrow Stop)$$
$$Q = P \sqcap (a \rightarrow Stop \,\Box\, b \rightarrow Stop)$$

These two processes are equivalent under conformance/stable revivals semantics, since both can refuse any subset of $\{a\}$ and offer b, or vice-versa. They are not equivalent under ready sets since Q can refuse $\emptyset$ and offer both a and b at the same time.

Just as the concept of $\checkmark$ in CSP gives a convenient solution not available in CCS, the nature of the parallel and restriction operators in CCS makes stuck-freeness rather more natural to specify there. As stated in [FHRR04], it is that no unsynchronised label of a sort local to the network can be available when nothing else is in a stable state: it is thus definable as a property of the process representing the network (unrestricted) rather than of the individual network components.

The following definition captures this CCS style in language which is also appropriate to CSP.

Definition 3 The process N is $\mathcal{R}$-stuck-free with respect to the set of actions A provided it has no revival of the form $(s, \Sigma - A, a)$ with $s \in (\Sigma - A)^*$ and $a \in A$.

Most interestingly, the mechanisation of the *RespondsToLive* specification we presented in [RSR04] used a modified parallel composition of the pair, with much in common with the ordinary CCS one: parallel processes are enabled to perform an unsynchronised event as an alternative to parallel ones.

Exactly the same thing could be done in CSP to test stuck-freeness for networks: simply rename all synchronised events to both themselves and a special event *stuck* as an alternative, which is not synchronised. The network is then stuck-free if it has no revival $(s, \Sigma - \{stuck\}, stuck)$ for any trace s not containing *stuck*.

In order to formulate *RespondsTo* for $\mathcal{R}$ we need to extend the latter to include the termination signal $\checkmark$. The traces component T is extended to include members of the form $s \frown \langle \checkmark \rangle$, where $s \in \Sigma^*$ (recall that $\checkmark \notin \Sigma$). The deadlock component D is unchanged: still members of Σ^* (for a terminated process is not deadlocked). A revival is of the form (s, X, a), where $s \in \Sigma^*$, $X \subseteq \Sigma$ and $a \in \Sigma^{\checkmark}$. In other words, $\checkmark$ is not recorded in the refusal set, but can be the successor event a. This comes from the philosophy, described in detail in [Ros98] that termination is a "signal" event: not one the environment can refuse or which can meaningfully be offered as an alternative to another visible event. If $s \frown \langle \checkmark \rangle \in T$ then we specify $(s, \Sigma, \checkmark) \in R$: this states that a process which can terminate does not have to offer any other alternative (even τ implicitly).

The structure expressed here allows us to decide whether a process which can terminate after trace s can refuse to do so. For then $s \in D$ or $(s, X, a) \in R$ for some $a \neq \checkmark$: implicitly every revival with $a \neq \checkmark$ implies the refusal of $\checkmark$.

Note that if $P = (T, D, R)$ is a process represented in $\mathcal{R}$ we can easily calculate (bearing in mind the conventions set out in [Ros98] for $\mathcal{F}$):

[3] This problem means that one cannot, for example, modify this congruence so that it records traces of length two or less after a refusal: the result would not be compositional under hiding.

$$\begin{aligned} failures(P) &= \{(s, X) \mid X \subseteq \Sigma^{\checkmark} \wedge s \in D\} \\ &\cup \{(s \frown \langle\checkmark\rangle, X) \mid X \subseteq \Sigma^{\checkmark} \wedge s \frown \langle\checkmark\rangle \in T\} \\ &\cup \{(s, X) \mid (s, X, a) \in R\} \\ &\cup \{(s, X \cup \{\checkmark\}) \mid (s, X, a) \in R \wedge a \neq \checkmark\} \end{aligned}$$

The representation in the Stable Failures Model $\mathcal{F}$ of P is then $(T, failures(P))$.

Now we have extended our model, it is capable of giving a completely precise definition of *RespondsTo*.

Definition 4 We say that Q $\mathcal{R}$-*RespondsTo* P if for every trace s, there do not exist $(s \restriction \alpha P, X, a) \in R_P$ and $(s \restriction \alpha Q, Y) \in failures(Q)$ such that $a \in J^{\checkmark}$ and $J^{\checkmark} \subseteq X^{+} \cup Y$. Here, $X^{+} = X$ if $a = \checkmark$, and $X \cup \{\checkmark\}$ otherwise.

This precisely captures the concept of P having a communication it wants to make with Q, but them being unable to agree on any.

This *implies RespondsToLive* over $\mathcal{F}$ since if $(s, J^{\checkmark}) \in failures(P \underset{J}{\parallel} Q)$ is created by the maximal failures $(s \restriction \alpha P, X)$ of P and $(s \restriction \alpha Q, Y)$ of Q, then if P, Q satisfy the condition above, $(s \restriction \alpha P, X)$ either comes from a deadlock trace $s \restriction \alpha P$ or a revival $(s \restriction \alpha P, X, b)$ with $b \notin J$. In the second case, by the healthiness condition R2 above, and $Q\mathcal{R}$-*RespondsTo* P, we get that $J \subseteq X$. In either case $(s \restriction \alpha P, J^{\checkmark}) \in failures(P)$.

Our new definition is very close to the original definition of *RespondsTo* over $\mathcal{F}$. The old definition says that if P can refuse X and do other things in $J^{\checkmark}$ besides X, then Q cannot refuse them. Our new definition, in fact, says precisely the same except that it is now able to couple the "do other things" more closely to X: they are necessarily from the same state. With this in mind it is straightforward to see that the definition over $\mathcal{F}$ implies the one over $\mathcal{R}$.

Both these implications are what we might have hoped for. Furthermore, if P is deterministic in the usual CSP sense (with each process fully characterised by its traces), all three conditions are equivalent. Note that in practical networks, parallel components are nearly always deterministic.

RespondsTo is both refinement-closed and distributive over $\mathcal{R}$.

In this section we have shown that the concept of responsiveness can be captured more precisely in a model we have created specially for this purpose. Indeed, this model is *fully abstract* with respect to both the natural operational characterisation of this or alternatively that of stuck-freeness.

The question then arises of which model we should generally choose to reason about *RespondsTo*. The obvious disadvantage of creating an ad hoc model to capture a condition is that it requires new theoretical work, new tool support, and places a substantial burden in ensuring that the rest of one's development is consistent with it. It also requires significant extra understanding on the part of anyone using it. Since we believe that in the vast majority of practical cases it will be possible to use the $\mathcal{F}$ version of *RespondsTo*, we think that pragmatically it is best to use that as the first line of attack, holding more sophisticated models for the rare cases where it is inadequate. As and when there is proper tool support for the refusal testing model of CSP [Muk93], based on Phillips's work [Phi87], it will make sense to reformulate our conditions in that ([FHRR04] observe that refusal testing can capture stuck-freeness[4]). For $\mathcal{R}$ is a weaker equivalence than refusal testing, so the latter can express our properties precisely.

This section has described how the development of a new fully abstract model for CSP has arisen from our work on responsiveness and its comparison to stuck-freeness. Full details of the CSP semantics for this model are beyond the scope of this paper but can be found in [Ros06] together with justification of the claim of full abstraction.

6. Further remarks

We have developed a general property characterising responsiveness of interacting components formulated in CSP which can be verified using the techniques of FDR. As shown in [RSR04], adding components which are responsive in our sense never introduces deadlock. These results have application both for component-side system development and for on-the-fly conformance checking and/or selection of distributed services.

This paper has developed our work on responsiveness both in terms of its application and in the theory underpinning its use. The case study illustrates how the property can be used in practice and points to work in the

[4] Note that $\mathcal{R}$ is the strongest congruence which is weaker than both ready sets and refusal testing.

area of integrated formal methods to provide a route for linking interface specifications with state-based views of components. The comparison with CCS has yielded interesting results, illuminating further the relationship between CSP and CCS. This work has also led to the development of the new CSP stable revivals model.

In terms of the responsiveness property, the work most closely related is that of Fournet et al. [FHRR04] and a detailed comparison of the two approaches has been provided in the previous section. Our work on responsiveness also relates to work on non-standard refinements carried out by Bolton and Lowe [BL03]. They investigate refinements expressible 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, $IMPL$ is the implementation and F provides a constraint on the refusals under consideration. With P as $SPEC$, $P \parallel Q$ as $IMPL$ and $X = J$ this is equivalent to our weaker property, *RespondsToLive*. Bolton and Lowe also provide a machine-verifiable check which they judge to be more efficient than ours for larger X, less efficient for smaller X.

The concept of non-blocking processes is also important in many other contexts. For example, in the assume-guarantee setting adopted by Amla et al. [AANT01] a rule is developed which is both sound and complete (for safety and liveness properties) for reasoning about component decomposition. The idea of nondeterministic blocking is not at issue here. In contrast, we treat blocking as fundamental and undesirable.

As discussed in Sect. 4, other aspects of our work are related to the CSP||B work on integrating CSP and B carried out by Treharne and Schneider [TS00, ST05]. They develop CSP controllers to drive B machines. The controllers may be integrated in a wider network, communicating with other controlled components. The proof obligations supporting this and theorems justifying the approach are set out in [ST05]. An alternative approach to verifying divergence-freedom of the composed system using uniform properties is proposed by Evans and Treharne [ET05]. This has the advantage of being amenable to more direct verification in PVS. The connection with our work arises both through the idea of responsiveness itself (the B is required to respond appropriately when the controller calls an operation) and through the concern to separate component specifications into an event-based view and a state-based view. The CSP||B method tackles issues of compatibility when composing two components specified in the two separate notations. It addresses fundamental issues that arise in this context which we believe will be applicable to our development. The approach is characterised by the requirement to make use of existing support tools for the methods used, and again this is a key feature of our work. However, our set-up is somewhat different: our interface specifications provide the contract for the outside world and we do not have a separate layer of communication between the B machine and the CSP process. We are working with independent components which must fend for themselves in a wider context, cooperating for their own ends rather than to achieve a wider network goal.

Other work on integration between state descriptions and behavioural specifications is also relevant here. Butler's csp2B approach [But00] and the more recent ProB method [BL05] bear much similarity to the CSP||B work, using a similar semantic link between the two notations. The more compositional approach of CSP||B appears to be better suited to our component-based view with passive B specifications, however the ProB approach to checking that B operations are not called outside their preconditions provides an alternative way which may be very useful in practice.

Further related work in this area includes Circus [WC02] which combines CSP with the state-based notation Z, and Event B [AM98] which introduces event ordering to B specifications. The tool support in these areas is not yet as developed as for the notations we have used.

The work mentioned in the preceding paragraphs indicates one area of further development for our responsiveness work: that is, how best to make use of existing results to integrate the different component views. A guiding principle is to maintain as far as possible the separation of concerns which allows us to make use of the strengths of each notation and the tool support that is currently provided. In addition to this, the question of scalability is very important, as is the independence of the composed components.

As component-based systems have become more wide-spread and more complex in nature, interest in the formal specification and development of such systems has increased. A common feature of this work is its concern to separate out the contract specification, which can be provided to the outside world, from the functional specification, which elaborates on details for internal component development. For example, Liu et al. [LHL04] provide a framework for contract specification based on Hoare and He's Unified Theory of Programming [HH98]. So far, our work has concentrated mainly on developing the condition of acceptability between components which we have called *RespondsTo*. It is also distinguished by the concern to use existing notations and technologies which can provide mature tool support for verification within the proposed development framework.

Further work is also needed to investigate responsiveness in other settings (such as in the presence of divergence and for infinite traces) and to set out properties of responsiveness which are useful for compositionality. We need to investigate further the way in which non-interface events are abstracted. Hiding has been used so far, but this may not always be appropriate.

The theoretical aspect of this paper which led to the discovery of the stable revivals model has already been taken further. Roscoe [Ros06] provides the necessary detail with proofs of full abstraction for the properties of the current paper.

Acknowledgment

The work of Joy Reed and Bill Roscoe was supported in part by the US Office of Naval Research.

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 process which never engages in any event nor terminates (deadlock).

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

$CHAOS(A)$ is the most non-deterministic, divergence-free process with alphabet A.

$a \rightarrow P$ 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.

$P \Box Q$ is external or *deterministic* choice.

$\sqcap_{x:X} P(x)$ 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 $\Box_{x:T} c.x \rightarrow P$.

$P \underset{X}{\parallel} Q$ 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. If X is omitted, it is taken to be Σ.

$P \,|||\, Q$ 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.

$P[[y/x]]$ is the process formed by renaming x to y in P.

Appendix B: Mechanical verification of *RespondsTo*

The checks described here are derived from those published in our earlier paper [RSR04]. We work in the CSP failures model and assume that all processes are divergence-free (which can be mechanically checked). P is a process with alphabet H and Q is a process which synchronises on set J of events. We define functions $G(P, Q)$ and $SPEC$ such that Q *RespondsTo* P if and only if the FDR-checkable assertion succeeds:

assert $SPEC \sqsubseteq G(P, Q)$

Let $H^{\bullet}$ and $H^{\diamond}$ be distinct, disjoint copies of H. Define the lazy abstraction [Ros98] of Q to be 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:

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

$P^\bullet$ is a copy of P which can engage in $a^\bullet \in H^\bullet$ whenever P can engage in a:

$$P^\bullet = P[[a^\bullet / a \mid a \in H]]$$

$P^\dagger$ is a process which runs P and $P^\bullet$ in parallel, with a regulator process $Reg^\bullet$. This runs P and $P^\bullet$ in a delayed lock-step manner, also ensuring that whenever $P^\bullet$ has demonstrated that there is something in $(J^\checkmark \cap initials(P))$, say a, then $G(P, Q)$ only comes up with refusal sets X not containing a so that $(J^\checkmark \cap initials(P)) - X$ is nonempty (the ones of interest for the condition).

$$Reg^\bullet = \Box_{a:H} a^\bullet \rightarrow ((\Box_{b:H} b \rightarrow (a == b \& Reg^\bullet)) \; \Box \; (a \in J) \& a^\diamond \rightarrow Stop$$

where $a^\diamond \in H^\diamond$ is a further separate version of a.

$$P^\dagger = ((P \;|||\; P^\bullet) \underset{H \cup H^\bullet}{\parallel} Reg^\bullet)[[a / a^\diamond \mid a \in J]]$$

Q *RespondsTo* P if and only if $G(P, Q) = P^\dagger \underset{J}{\parallel} LQ$ has no deadlock after an odd-length trace whose last member is in J^*. That is, if and only if it refines

$$\begin{aligned} Spec = (\Box_{a:J} a^\bullet \rightarrow (& \quad (1) \\ & (\sqcap_{b:J} b \rightarrow Spec) \\ & \Box \\ & (Stop \sqcap (\Box_{b:H-J} b \rightarrow Spec))) \\ \Box & \\ (\Box_{a:H-J} a^\bullet \rightarrow (Stop \sqcap (\Box_{a:H} a \rightarrow Spec)))) & \quad (2) \\ \sqcap Stop & \quad (3) \end{aligned}$$

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.

References

[Abr96] Abrial J-R (1996) The B-Book: assigning programs to meanings. Cambridge University Press, London

[AM98] Abrial J-R, Mussat L (1998) Introducing dynamic constraints in B. In: Bert D (ed) Proceedings of the 2nd International B Conference. Lecture Notes in Computer Science, Springer LNCS, 1393, pp 83–128

[AANT01] Amla N, Allen E, Namjoshi K, Trefler R (2001) Assume-guarantee based compositional reasoning for synchronous timing diagrams. Lecture Notes in Computer Science, Springer LNCS, 2031, pp 465–479

[BL03] Bolton C, Lowe G (2003) On the automatic verification of non-standard measures of consistency. In: 6th International workshop on formal methods, Dublin

[But92] Butler MJ (1992) A CSP approach to action systems. Oxford University Computing Laboratory, DPhil thesis

[But00] Butler MJ (2000) A practical approach to combining CSP and B. Formal Asp Comput 12(3):182–198

[BL05] Butler M, Leuschel M (2005) Combining CSP and B for specification and property verification. In: FM05, Springer LNCS, 3582, pp 221–236

[ET05] Evans N, Treharne H (2005) Linking semantic models to support CSP||B consistency checking. In: Proceedings of the AVoCS05, ENTCS, 145. Available at URL: http://www.sciencedirect.com/science/journal/15710661

[FHRR04] Fournet C, Hoare CAR, Rajamani SK, Rehof J (2004) Stuck-free conformance. In: Proceedings of the 16th international conference on computer aided verification (CAV'04), Springer LNCS, 3114, pp 242–254

[FSE97] Formal systems (Europe) Ltd. (1997) Failures divergence refinement: user manual and tutorial. Available at URL: http://www.fsel.com/documentation/fdr2/html

[Hoa85] Hoare CAR (1985) Communicating sequential processes. Prentice Hall, Englewood Cliffs

[HH98] Hoare CAR, He J (1998) Unifying theories of programming. Prentice Hall, Englewood Cliffs

[LHL04] Liu Z, He J, Li X (2004) Contract-oriented development of component systems. In: Proceedings of the 3rd IFIP international conference on theoretical computer science, Kluwer, Dordrecht, pp 349–366

[Mor90] Morgan CC (1990) Of wp and CSP. In: Gries D, Feijen WHJ, van Gasteren AGM, Misra J (eds) Beauty is our business: a birthday salute to Edsger W. Dijkstra. Springer, Berlin

[Muk93] Mukkaram A (1993) A refusal testing model for CSP. Oxford University Computing Laboratory, DPhil Thesis

[OH83] Olderog ER, Hoare CAR (1986) Specific ation-oriented semantics for communicating processes. Acta Informatica 23:9–66

[Phi87] Phillips I (1987) Refusal testing. Theor Comput Sci 50:241–284
[Ros98] Roscoe AW (1998) The theory and practice of concurrency. Prentice Hall Series in Computer Science
[Ros03] Roscoe AW (2003) On the expressive power of CSP refinement. In: Proceedings of the 3rd International workshop on automated verification of critical systems (AVoCS03)
[Ros06] Roscoe AW (2006) Revivals, stuckness and responsiveness. In preparation
[RS01] Reed JN, Sinclair JE (2001) Combining independent specifications. In: Proceedings of the ETAPS-FASE2001, Springer LNCS 2029, pp 45–59
[RSR04] Reed JN, Sinclair JE, Roscoe AW (2004) Responsiveness of Interacting Components. Formal Asp Comput 16(4):394–411
[ST05] Schneider S, Treharne H (2005) CSP theorems for communicating B machines. Formal Asp Comput 17:390–422
[TS00] Treharne H, Schneider S (2000) How to drive a B machine. In: Proceedings of the ZB2000, Springer LNCS, 1878, pp 188–209
[WC02] Woodcock J, Cavalcanti A (2002) The semantics of circus. In: Proceedings of the ZB2002, Springer LNCS, 2272, pp 184–203

Received 22 March 2006
Revised 1 August 2006
Accepted 12 February 2007 by R. Lazic, R. Nagarajan and J. C. P. Woodcock
Published online 5 April 2007