Electronic Notes in Theoretical Computer Science 40 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume40.html 19 pages

Refinement-Preserving Plug-In Components

J.N. Reed [1,2]

Oxford Brookes University, Oxford, UK

J.E. Sinclair [3]

University of Warwick, Coventry, UK

Abstract

We present a formal framework for characterising plug-in relationships between components whereby one does not cause the other to deadlock. We define the notion of a stable relation ϕ between co-operating processes such that whenever P and Q are related by ϕ, then any component-wise refinements P' and Q' are related by ϕ. We use stable relations to ensure that plug-in components can be separately refined whilst maintaining integrity of the original relational properties. We ground our notions in the CSP failures semantic model. The aim is to underpin a mixed-paradigm approach combining different specification methods, including state-based deductive formalisms such as Action Systems, and event-based model checking formalisms such as CSP/FDR. The objective is to play to the strengths and overcome limitations of each technique, by treating different system aspects with individual tools and notations which are most appropriate.

1 Introduction

A formal method is a mathematically-based theory which is used to describe and reason about the behaviour of a computer system. Application of a formal method encompasses specification of the system in a chosen formal notation, analysis and verification of key properties and stepwise refinement to a correct implementation. It is generally recognised that even partial use of these techniques during development can significantly increase the quality of software and hardware systems, with respect to correctness and maintainability. For example, the application of a general-purpose specification notation such as Z [24] has been found to lead to the earlier discovery of design flaws. Formal

[1] This work was supported in part by the US Office of Naval Research.
[2] Email: jnreed@brookes.ac.uk
[3] Email: jane@dcs.warwick.ac.uk

modelling and verification of small security protocols such as that by Lowe and Roscoe [14], Lowe [13] and Meadows [15] has revealed previously unsuspected flaws in the operation of these protocols.

Many different formal methods with differing theoretical bases have been proposed. An overview of formalisms, as given in [7,10], testifies to a variety of approaches and methods. No one formalism is fully suitable for all aspects of industrial-sized applications, as we have illustrated by directly comparing strengths and weaknesses of state-based, deductive reasoning approaches and event-based, model checking approaches applied to a distributed mail system [19] and general routing protocols [20]. With a deductive reasoning approach, a specification gives an abstract description of the significant behaviour of the required system. This behaviour can be verified for the defined implementation by proving the theorems which constitute the rules of refinement. With model checking, a specification corresponds to a formula or property which can be exhaustively evaluated on a specific finite domain representing an implementation. Deductive reasoning is more general, but only partially automatable. Model checking is more limited but fully automatable. Our previous work [19] shows that in addition to theoretical limitations of a notation, its form leads towards specification of a certain style and often with particular implicit assumptions.

Combining different formalisms potentially offers a fuller picture, but the question remains as to how this integration can best be achieved. A number of options are available. A unified notation may be applied throughout, but possibly at the expense of prohibitive complexity and lack of optimality for individual parts of the system. Alternatively, specifying different system aspects in different notations is attractive, particularly for greater flexibility in incorporating "off-the-shelf" components which ideally come with some guarantee of their behaviour. In a distributed system it can be the case that a transaction or service requires the interoperation of a chain of components and services which combine to produce (hopefully) 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 application depends not only on the integrity of its own functions, but also on its interactions with other components.

Our aim is to provide a framework in which components of a system can be specified and developed independently in different notations, with constraints on their interfaces ensuring appropriate cooperation so that the specification of the overall system can be satisfied. We develop a relational view of components, with one acting as a plug-in to the other if it does not increase the possibilities of deadlock. This view has the advantage of offering a mechanism to specify minimal interface properties required of a plug-in component, but in general it lacks stability, by which we mean being preserved by refinement.

The notations chosen are CSP [12,21] for event-based specification and model checking, and Action Systems [1] which allow state-based description and deductive reasoning. We show by examples that certain desirable relational properties of a system with arbitrary components are not necessarily preserved by component-wise refinements, and we describe a solution for avoiding this refinement paradox. We conclude by placing our work in the context of other research linking the two notations, notably, that by Morgan [16], Butler [4] and Treharne and Schneider [25,26].

The papers is organised as follows. In section 2, we motivate our approach with a simple example of a subscription database. In section 3, we give an overview of our work. Section 4 contains a brief introduction to CSP. In section 5, we provide examples illustrating desirable relationships between components, and that such relationships are not in general preserved by refinement, which we regard as unstable. In section 6, we provide CSP formulations of relationships which are stable, and define our notion of a plug-in relationship. Section 7 briefly revisits the database example. In section 8 we discuss comparisons with other work, and present conclusions.

2 Interoperating components: an example

Consider a secure database which answers requests for information from authorised customers. We wish to treat the functional properties of the subscription database and authorisation protocol as separate units which can be further developed and verified separately in different ways as appropriate. The information retrieval services are best treated with a state-based formalism such as Action Systems, while the key-exchange protocol is most effectively treated with a finite-automata, exhaustive state search technique such as CSP.

Correct operation of the state-based information retrieval services would rely on a crypto-protocol specification ultimately implemented with a suitable plug-in component, which as a separate concern may range from providing only simple key exchange through to providing additional authorization and integrity. The specification simply needs to state that a necessary task will be performed (such as, a common session key being distributed to both database and client), with no need to place constraints on the values it requires, nor on the steps required to execute the crypto-protocol. We wish to outline the required task in the main specification for the system and hand over to a more concrete protocol for the details. The top level specification of the protocol should give us sufficient guarantees of its behaviour in order that it, and any more concrete refinement of it, be considered compatible as a plug-in to our system specification. In this paper we explore what constitutes a *compatible plug-in* which can be verified as such.

Figure 1 gives part of a top level specification for the database written as an Action System. An Action System incorporates both a description of the state variables and the effects of each action upon them, and the order in which execution may occur. The full specification of which this is a small part defines the state of the database and the mechanisms required to serve

Actions of database obtaining key for communication with user u

$$\begin{array}{l} \textbf{for } u \in \mathit{USR} \textbf{ action } \mathit{GetKey}_u \ :- \\ \quad \mathit{status}(u) = \mathit{startsession} \ \longrightarrow \ \mathit{status}(u) := \mathit{needkey} \end{array}$$

$$\begin{array}{l} \textbf{for } u \in \mathit{USR} \textbf{ action } \mathit{GotKey}_u \textbf{ in } k? : \mathit{KEY} \ :- \\ \quad \mathit{status}(u) = \mathit{needkey} \ \longrightarrow \ \mathit{status}(u), \mathit{key}(u) := \mathit{haskey}, k? \end{array}$$

Fig. 1. Action System specification for part of secure database system

requests from valid customers. Conditions such as clearance to access data can be succinctly captured and verified with this formalism. In contrast, other tasks such as developing and verifying a suitable key exchange protocol between the parties is not best-suited to the Action System notation. The Action System specification states the requirement for this in general terms (these are the actions in Figure 1) with a key request and expected response for any particular user u. Each action is of the form $g \longrightarrow c$ where g is a guard determining when the action may be executed and c is the command which is executed when the action is selected.

A detailed understanding of Action Systems is not required here, and the interested reader is directed to the work of Morgan [16] and that of Butler [5,6]. It is worth noting that *status* and *key* are examples of state variables characteristic of this style of formalism. At this point, we wish to "hand over" to a suitable protocol, very possibly developed using a different notation, in this case, CSP.

3 Combining specifications – an overview

The traces/failures/divergences models, described in the next section, provide a unifying formal semantics for Action Systems and CSP so that we can combine specifications in a meaningful way. That is, if P is an Action Systems specification for some aspect or component of a system, and Q is a CSP specification for another aspect or component, then $P \parallel Q$ represents their parallel combination with behaviour well-defined, and any safety (that is, trace) property of either P or Q with respect to their common actions/events is preserved by $P \parallel Q$.

In contrast to safety properties, liveness properties are not preserved by the $\parallel$ operator, as illustrated by Example 5.1 below. Our focus is on plug-in relationships among components, whereby we mean that component Q plugs in to component P iff Q does not cause P to deadlock when they are run in parallel. We formalise this notion (a desirable behaviour for a plug-in component to offer its controller), and show by example that unfortunately, more deterministic refinements of processes satisfying such a relationship do not themselves have to satisfy the relationship.

Our solution for avoiding this potential pitfall ("refinement paradox") is to characterise P and Q with a stronger plug-in relationship which (1) implies Q does not deadlock P, and (2) is preserved by refinement. This is analogous to establishing a non-invariant post condition for a loop using a stronger loop invariant; the desired post condition on its own is not necessarily sufficiently strong to be invariant, but can be deduced from a stronger predicate which is invariant.

4 An introduction to CSP

CSP [12] 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. CSP is a process algebra.

A related series of semantic models capture different aspects of observable behaviours of processes: traces, failures and divergences. The simplest semantic model is based on the concept of a *trace*: a finite sequence of events drawn from an alphabet set Σ of all events. A trace for a process represents a sequence of events which a process could be observed to perform. The *traces* model characterises a process as the set of all traces, $\mathit{traces}(P)$, it can perform. The traces model is sufficient for reasoning about safety properties. In the failures model [2] a process P is modelled as a set of *failures*. A *failure* is a pair (s, X) for s a finite trace of events of $\alpha(P)$, and X a subset of events of $\alpha(P)$; $(s, X) \in \mathit{failures}(P)$ 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 failures model allows reasoning about certain liveness properties. More complex models such as failures/divergences [3] and timed failures/divergences [17] have more structures allowing finer distinctions to support more powerful reasoning. For the rest of this paper, we restrict our discussion to the *failures* model.

We say that 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:

$$\mathit{failures}(P) \;\subseteq\; \mathit{failures}(S)$$

which tells us that any trace of P is a trace of S, and P can refuse an event 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 Formal Systems' model checker, FDR [11]. An overview of CSP syntax and a failures model is given in the Appendices.

5 Combining Components

We view our top-level specification as being structured as a set of interoperable specifications whose parallel combination describes desirable properties of the system. As noted in section 3, that each component satisfies a desirable liveness condition does not guarantee that the parallel combination satisfies the condition. Example 5.1 sets the scene for characterising the suitability of one specification "plugging in" to another.

Example 5.1 Suppose process P makes a request (event a) to receive two keys (events $k1$ and $k2$). P does not care in which order the keys are obtained. Here $\Box$ represents external choice.

$$P = (a \to k1 \to k2 \to P) \Box (a \to k2 \to k1 \to P)$$

Process Q is a component which is chosen to distribute keys, and this happens to be specified as responding first with $k1$ and then with $k2$.

$$Q = a \to k1 \to k2 \to Q$$

It is expected that establishing the keys as specified by Q will be achieved by some more detailed algorithm which, with internal details hidden, refines Q. We regard Q as a plug-in to P, since their joint behaviour expressed by $P \parallel Q$ behaves desirably. Clearly not every process which returns keys should be regarded as a plug-in to P. For example, consider R:

$$R = a \to k1 \to k1 \to R$$

This time R does not interact with P in a desirable way since it does not have a acceptable pattern of behaviour, and the result is deadlock at the third step. $\Box$

Refinement brings additional difficulties. Refinement has various desirable properties, such as transitivity and monotonicity which are very useful for compositional development. These properties allow us to know that whenever a specification is good enough for some purpose, then so is any refinement. However, Example 5.2 below shows that if we are dealing with *relationships* between component processes which ensure that their parallel combination describes desirable properties of our system, it does not follow that component-wise refinements are suitable according to the same criteria. That is, for a relation ρ on processes, if

$$P \rho\ Q,\ P \sqsubseteq P' \text{ and } Q \sqsubseteq Q'$$

then by monotonicity we know that

$$P \parallel Q \sqsubseteq P' \parallel Q'$$

but it is not necessarily true that

$$P' \rho \; Q'$$

Example 5.2 Suppose P and Q are both defined as follows:

$$\begin{aligned} P &= (x \to P) \sqcap STOP \\ Q &= (x \to Q) \sqcap STOP \end{aligned}$$

Here $\sqcap$ represents internal (nondeterministic) choice. We observe that P and Q satisfy the relationship:

$$P \sqsubseteq P \parallel Q$$

(this relationship might appear desirable since it ensures that Q cannot cause any deadlock not also allowed by P). P and Q satisfy this property, but not refinements:

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

Clearly $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$ and yet $P' \not\sqsubseteq P' \parallel Q'$.

□

Example 5.2 shows that potential candidates for describing suitable plug-in relationships between components may not be preserved by refinement. This would be disastrous from the point of view of building systems with independently developed components. In this paper we concentrate on patterns of interaction between two processes P and Q which, as in Example 5.1 above, behave like a simple remote procedure call from P to Q. We regard Q as a *plug-in* to P if Q responds to P in a way which does not increase the opportunities for deadlock, and for any component-wise refinements Q' and P', Q' responds to P' in a way which does not increase the opportunities for deadlock to P'. In the rest of the paper, we investigate how to formalise these notions. We first define a general property of relations which is useful for capturing the notion that refinements of processes inherit their parents' relationship.

Definition 5.3 (Stable Relations) *Let ϕ and ρ each be a relation on X. We say that ϕ is stable with ρ iff for $x\phi y$, $x\rho x'$, $y\rho y'$, then $x'\phi y'$.*

Example 5.4 Let R be the relation $<$ and S the relation which holds between x and y iff $y = x + 1$. Then R is stable with S. This example also shows that the property is not symmetric since S is not stable with R.

□

In general, relations are not stable with themselves, for example, the relation $<$ is not stable with $<$. Equivalence relations are stable with themselves, though not in general stable with arbitrary other relations.

6 Stability with Refinement

We can capture the notion that a given relationship ϕ between co-operating specifications is inherited by refinements by requiring ϕ to be stable with $\sqsubseteq$. We say that ϕ is stable whenever

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

We can make the relationship given in Example 5.2 stable by requiring not only that $P \sqsubseteq P \parallel Q$ but also that P is deterministic. However this does not offer a general solution to this "refinement paradox"; it may defeat the purpose of refinement since it disallows the use of nondeterminism as a mechanism for abstraction. We might instead insist that P and Q always operate together in a deadlock free fashion. We cannot ensure this by simply requiring that each of P and Q is deadlock free, as illustrated by

$$P = \bigsqcap_{T} x \rightarrow P \;\text{ and }\; Q = \bigsqcap_{T} x \rightarrow Q.$$

P and Q are each deadlock free since each is willing to do some event of T, but $P \parallel Q$ can deadlock whenever they do not agree on their chosen events. However, if we require $P \parallel Q$ to be deadlock free as well, Example 6.1 below ensures that any refinements P' and Q' inherit their parents' good behaviour and so cannot deadlock when they themselves are run in parallel.

Example 6.1 Processes P and Q are mutually deadlock free iff each of P and Q is deadlock free, and their parallel composition is deadlock free. Mutual deadlock freedom is stable.

A process is deadlock free iff it refines the process DF which is always willing to do something in the alphabet Σ:

$$DF = \bigsqcap_{\Sigma} a \rightarrow DF$$

Let $DF \sqsubseteq P$, $DF \sqsubseteq Q$, and $DF \sqsubseteq P \parallel Q$. Also let $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$. Then it follows by monotonicity and transitivity that $DF \sqsubseteq P'$, $DF \sqsubseteq Q'$, and $P \parallel Q \sqsubseteq P' \parallel Q'$, and $DF \sqsubseteq P' \parallel Q'$.

□

Mutual deadlock freedom, though stable, is too strong a property to require of co-operating applications Q and P for which Q responds to a one-time invocation from P. For example, Q may be a set-up process which P calls once and only once. Thus, $P \parallel Q$ will properly deadlock on their joint alphabet after Q finishes its work, whilst P carries on with other events not requiring any participation from Q. Or Q behaves as a remote procedure call to P, which may acceptably never invoke any services provided by Q. Indeed, we may wish to allow P itself to deadlock.

Let us imagine that we want P to trigger Q by handing over some parameters, which Q processes, subsequently returning results back to P. P is in control, and may invoke Q arbitrarily, including never; Q is always required to be ready, and is willing to be invoked forever. What is required is a stable relation between P and Q which implies that their parallel combination deadlocks on their set of common events, J, only where P chooses. Then, if we refine P and Q with P' and Q', the parallel combination of P' and Q' deadlocks on J only where P' chooses.

We now identify stable relations between P and Q which imply that Q does not deadlock P. For the rest of the paper, we assume J represents the set of common events requiring participation by P and Q, that is, the events requiring synchronization by both processes. We assume that P may engage in events outside J, but Q does not (other events may be regarded as internal to Q). The first definition characterises interactions between processes P and Q whereby whenever P is ready to output a value x on the channel T to Q, then Q is ready to receive it.

Definition 6.2 Q ***listens on*** T ***to*** P, for $T \subseteq J$ means

$$x \in T \wedge s \frown \langle x \rangle \in \mathit{traces}(P) \wedge s \in \mathit{traces}(P \parallel_J Q)$$

$$\Rightarrow (s \upharpoonright J, \{x\}) \notin \mathit{failures}(Q)$$

Theorem 6.3 *The relation* listens on T to *is stable.*

Proof. Assume

$$(1)\, P \sqsubseteq P' \text{ and } Q \sqsubseteq Q'$$

$$(2)\ x \in T \wedge s \frown \langle x \rangle \in \mathit{traces}(P) \wedge s \in \mathit{traces}(P \parallel_J Q)$$

$$\Rightarrow (s \upharpoonright J, \{x\}) \notin \mathit{failures}(Q)$$

We must show

$$x \in T \wedge s \frown \langle x \rangle \in \mathit{traces}(P') \wedge s \in \mathit{traces}(P' \parallel_J Q')$$

$$\Rightarrow (s \upharpoonright J, \{x\}) \notin \mathit{failures}(Q')$$

Assume the hypothesis of the implication. By (1) $s \frown \langle x \rangle \in \mathit{traces}(P)$ and $s \in \mathit{traces}(P \parallel_J Q)$. Thus by (2), $(s \upharpoonright J, \{x\}) \notin \mathit{failures}(Q)$, and again by (1) $(s \upharpoonright J, \{x\}) \notin \mathit{failures}(Q')$. □

We next define a stable property which characterises a process Q outputting along channel R whenever P wants. We do not want to overly constrain Q, that is, Q should be allowed to deadlock on R whenever P is willing to do so. If we require that P either must be prepared to accept any answer communicated by Q, or possibly deadlock – but not both simultaneously – then the relation is stable. The following definition characterises processes P and Q whereby whenever P is ready to receive input from Q, Q is ready to send it. It says that whenever P is ready to receive any value of R – there is an obligation on P to be ready to receive any other value as well, and there is an obligation on Q be ready to output something.

Definition 6.4 Q answers on R to P, for $R \subseteq J$ means

$$x \in R \wedge y \in R \wedge s \frown \langle x \rangle \in \mathit{traces}(P) \wedge s \in \mathit{traces}(Q) \upharpoonright J$$
$$\Rightarrow (s \upharpoonright J, R) \notin \mathit{failures}(Q) \upharpoonright J \wedge (s, \{y\}) \notin \mathit{failures}(P)$$

Theorem 6.5 *The relation* answers on R to *is stable.*

Proof. Assume

$$(1)\; P \sqsubseteq P' \text{ and } Q \sqsubseteq Q'$$

$$(2)\; x \in R \wedge y \in R \wedge s \frown \langle x \rangle \in \mathit{traces}(P) \wedge s \in \mathit{traces}(P \parallel_J Q)$$
$$\Rightarrow (s \upharpoonright J, R) \notin \mathit{failures}(Q) \wedge (s, \{y\}) \notin \mathit{failures}(P)$$

Let

$$x \in R \wedge y \in R \wedge s \frown \langle x \rangle \in \mathit{traces}(P') \wedge s \in \mathit{traces}(P' \parallel_J Q')$$

We must show

$$(s \upharpoonright J, R) \notin \mathit{failures}(Q') \wedge (s, \{y\}) \notin \mathit{failures}(P')$$

Assume $(s \upharpoonright J, R) \in \mathit{failures}(Q')$. By (1), $(s \upharpoonright J, R) \in \mathit{failures}(Q)$, and furthermore, $s \frown \langle x \rangle \in \mathit{traces}(P)$ and $s \in \mathit{traces}(P \parallel_J Q)$. Thus by (2), $(s \upharpoonright J, R) \notin \mathit{failures}(Q)$, and this contradiction establishes that $(s \upharpoonright J, R) \notin \mathit{failures}(Q')$. Assume $(s, \{y\}) \in \mathit{failures}(P')$. Again by (1), $(s, \{y\}) \in \mathit{failures}(P)$, but this contradicts (2) thereby establishing the theorem. □

The next theorem establishes that if Q listens on T to P, and Q answers on R to P, then Q does not deadlock P on their common set of events, $J = T \cup R$.

Theorem 6.6 *If Q listens on T to P and Q answers on R for $T \cup R = J$, then $(s, J) \in \mathit{failures}(P \parallel_J Q) \Rightarrow (s, J) \in \mathit{failures}(P)$.*

Proof. Assume $(s, J) \in \mathit{failures}(P \parallel_J Q)$. By *failures* model definition for the parallel operator (see Appendix), for some refusal sets X, Y, $X \cup Y = J$, $s \in \mathit{traces}(P)$ and $s \upharpoonright J \in \mathit{traces}(Q)$ and $(s, X) \in \mathit{failures}(P)$, and $(s \upharpoonright J, Y) \in \mathit{failures}(Q)$. Assume there exists and $x \in J$ such that $s \frown \langle x\rangle \in \mathit{traces}(P)$. There are two cases : $x \in T$ or $x \in R$.
Case 1. Assume $x \in T$. Since Q listens on T to P, $(s \upharpoonright J, \{x\}) \notin \mathit{failures}(Q)$. Since $(s \upharpoonright J, Y) \in \mathit{failures}(Q)$, then by *failures* axiom (M3) which says that any subset of a refusal set is itself a refusal set, it follows that $x \notin Y$. This contradicts that $X \cup Y = J$, and case is proved.
Case 2. Assume $x \in R$. Since Q answers on R to P, $(s \upharpoonright J, R) \notin \mathit{failures}(Q)$. Furthermore, $(s, \{y\}) \notin \mathit{failures}(P)$ for any $y \in R$. By (M3), it follows that $X \cap R = \{\}$. Hence $R \subseteq Y$ and again by (M3), $(s \upharpoonright J, R) \in \mathit{failures}(Q)$. This contradiction proves the case and the theorem. □

The above theorem ensures that whenever the parallel system could refuse all of the fixed set J, then the specification for P alone allows it to refuse J. It follows that if at any point several possible events are acceptable to P, Q will be compatible as long as it permits at least one of these events.

We use this notion to give a definition of a plug-in relationship:

Definition 6.7 Q **plugs into** P **on** A *for* $A \subseteq J$ *means*

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

This relation constrains Q to deadlock only when P might. But it is not stable. For the processes defined below, Q plugs into P on R (both behave chaotically), but for the refinements, Q' does not plug into P' on R.

Example 6.8 $P \;=\; (\Box_R\, r \to P) \sqcap \mathrm{STOP} \qquad Q \;=\; (\sqcap_R\, r \;\to Q) \sqcap \mathrm{STOP}$
$P' = \;\Box_R\, r \;\to P \qquad\qquad\qquad Q' = \;\mathrm{STOP}$

□

If Q is a plug-in to P on channels T and R, it would be tempting to generalize theorem 6.6 that Q is a plug-in to P on $T \cup R$. However, this is not in general true, as illustrated by the following example.

Example 6.9

$$P \;= a \to P \sqcap b \to P$$

$$Q \;= a \to Q \sqcap b \to Q$$

Since for any $s \in \mathit{traces}(P)$, $(s, \{a\}) \in \mathit{failures}(P)$ and $(s, \{b\}) \in \mathit{failures}(P)$

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

channel $GetKey.USR$

channel $GotKey.KEY$

$$KeyX(u) = GetKey.u \rightarrow \sqcap_{k:KEY} GotKey.u!k \rightarrow KeyX(u)$$

$$Q = |||_{u:USR}\, KeyX(u)$$

Fig. 2. CSP specification for key-exchange plug-in to secure database system

but $(s, \{a, b\}) \in failures(P \parallel_J Q)$ and $(s, \{a, b\}) \notin failures(P)$. Thus, Q plugs in to P on $\{a\}$ and also on $\{b\}$, but not on $\{a, b\}$. $\square$

Summary We have identified a notion of one process Q being as live as another process P, whereby we mean that Q introduces no more possibilities for deadlock than P. The examples show that there is a conflict between allowing nondeterminism in specifications for P and Q, and preserving this liveness relationship under refinement. We have identified a notion of a plug-in relationship between P and Q, which requires that this liveness relationship is preserved by component-wise refinements.

The listening and answering relations between P and Q defined above ensure that Q acts as a stable plug-in to P, with P nondeterministically triggering Q, which returns results back to P. If Q listens on T to P, and Q answers on R to P, for $J = T \cup R$ then Q plugs-in to P , and if $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$ then Q' plugs-in to P. Thus we can confidently refine P and Q separately, without having to verify that the refinements cooperate as desired.

7 CSP plug-in to an Action System specification

The relations that we have defined provide a means of ensuring that two processes can be separately developed, whilst maintaining integrity of combined behaviour.

Suppose we have a CSP specification for a key exchange protocol Q as given in Figure 2. It simply describes a process which is always willing to communicate on the *GetKey* channel and accept any user u as input, and subsequently distribute a common key to the server and user, in nondeterministic order. It is deliberately abstract to allow for a variety of specific key distribution protocols. This can be viewed as a “minimum specification” of the key exchange to be refined and verified as a separately. Our ultimate goal is to plug this in to the Action System and show that the behaviour allowed by the CSP plug-in is acceptable to the database specification. Intuitively, we see that Q – specified in CSP – behaves as a suitable plug-in characterised in the previous section, to P – specified in Action Systems in Figure 1:

> Q listens on *GetKey* to P, and for each user u, Q answers on *GotKey*.u to P.

The semantic links between Action Systems and CSP (see Section 8) provide the mechanisms to integrate these separate views given in the two notations. How most effectively to check that component specifications above satisfy these relational properties is a challenging problem under research.

There are various aspects of the CSP specification which we might want to further develop through refinement. One is to unfold specific details of a chosen key exchange protocol by describing an intended implementation, or an off-the-shelf component. This might introduce a trusted key server together with a prescribed sequence of events required by the protocol between user clients and the database server. We can verify that the implementation is valid by checking that it with the new events hidden is a refinement of Q, thus establishing that the chosen protocol behaves as expected.

A significant advantage of treating the security protocol as a suitable CSP plug-in is that we can naturally specify event-based behaviour and check relevant properties automatically using the FDR model checker, perhaps with various induction techniques (for example [8,9]) and data independence techniques (for example [21,22]) for transcending bounded state. A significant disadvantage of using Action Systems for such aspects is that properly specifying allowable sequences of actions is very awkward.

Another reason for refining the CSP specification is that we might want to analyse behaviour of a chosen protocol with respect to security, e.g., robustness against deliberate or inadvertent attacks by intruders. Demonstrating security or (lack of it) might involve modelling attackers as processes with certain constrained behaviour, such as not having the ability to decipher encrypted messages, whilst having the ability to intercept and replay communications containing cipher text [13,14]. For example, Lowe and Roscoe [14] discover potential security flaws with the TMN key exchange protocol, revealed by counter examples provided by FDR showing that attackers could perform operations specifically disallowed by the CSP specification. Again, adding detail involves introducing internal actions, and checking that the new system is a valid refinement. A disadvantage of Action Systems for this sort of analysis is that automated deductive reasoning tools cannot generally provide counter examples for flawed conjectures.

8 Relation to other work and conclusions

Butler has developed a tool csp2b [5,6] which provides a means of combining CSP with standard B specifications. The technique builds on weakest-precondition formulations for Action Systems given by Morgan [16] and Butler [4]. CSP-like descriptions are translated into machine readable B specifications, which can then be verified by a deductive tool. Event-based CSP descriptions and state-based Action System-like ones are combined into one B machine, with appropriate proof obligations to ensure liveness properties of the system.

In contrast, our desire is to modularise both specification and analysis from the beginning in order to reduce effort and space explosion for those systems where it is possible. Treharne and Schneider [25,26] provide techniques for using CSP and the B-method. They define CSP control executives for state-changing operations based on the B-method. They identify wp-formulated proof obligations on the CSP specifications to ensure that appropriate preconditions and guards (which they distinguish) are not violated. They do not generate an encompassing B machine, and do not allow shared state. Thus independent analysis of the separate specifications is possible.

The main focus of Butler, Treharne and Schneider approaches has been to use CSP as a convenient way to specify constraints on the sequencing of, i.e., controlling the state-based actions. Here we have taken the opposite perspective: we want the state-based actions to control the CSP. In order to accomplish this we require relational constraints to characterise a notion of minimum requirements for a plug-in component: the plug-in must operate under the control of the main component in that the combination deadlocks only at the behest of the main.

We identify stability between specifications, which both simplifies proof obligations for top-level components and removes the need for re-establishing proof obligations for refinements, which would otherwise be required. We are concerned with characterising suitable, possibly off-the-shelf, plug-ins; our techniques allow us to view the Action System and CSP specification techniques as symmetric – either can describe plug-ins to the other. However our definitions so far are formulated in CSP and the best way to exploit the semantic link for verification between different notations is the subject of ongoing research.

Our notion of stability is a very strong requirement for component specifications, but it brings commensurate advantages in capturing the essence of loosely coupled components and reducing verification effort. Our approach is to do the hard work of proof for general paradigms of loosely coupled components, such as our theorems about listening and answering. We then reap the benefit when using any Action System and CSP specifications which fit the patterns. Since many plug-in relationships fit this paradigm, we can have a significant reduction of effort.

In formulating stable properties we have a difficulty similar to that of non-interference properties in security. Many different variants are possible and the effects are not always apparent until pathological examples are examined. Our notions of listening and answering are not fully general and variations may be required for different patterns of communication between components.

Automated checking of stable relational properties is desirable. We give a formulation of the listening and answering properties which can be model-checked by FDR [18]. This involves transforming the original system of components into a modified system, which can be checked against specialised specifications.

Our driving motivation is to contain inherent problems of scale in applying formal techniques to large applications. The goal of a great deal of current research is to combine different formal approaches in order to treat different aspects of a given system. There is a danger that combining techniques for a particular system creates prohibitive complexity. Our aim is to divide and conquer potential complexity by structuring separation-of-concerns specifications early in the development process, so that independent analysis can be effectively performed. Finally we note that our techniques are formulated using Action Systems and CSP, but we feel that concepts which we have identified are generally applicable to other formal and semi-formal specification techniques.

Acknowledgement The authors would like to thank the referees for valuable comments and suggestions.

Appendix A. A brief overview of CSP

The CSP language is a means of describing 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.

$STOP$ is the simplest CSP process; it never engages in any action, never terminates.

$SKIP$ similarly never performs any action, but instead terminates successfully, passing control to the next process in sequence (see $P; \ Q$ below).

$a \rightarrow P$ is the most basic program constructor. It waits to perform the event a and after this has occurred it behaves as process P. The same notation is used for outputs ($c!v \rightarrow P$) and inputs ($c?x \rightarrow P(x)$) of values on named channels.

$P \sqcap Q$ 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 actions 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 $\sqcap$). A deterministic choice between $STOP$ and another process, $STOP \,\Box\, P$ is identical to P.

$P \parallel Q$ is parallel (concurrent) composition. P and Q evolve separately, but events in the intersection of their alphabets occur only when P and Q agree (i.e. *synchronise*) to perform them. (We use this restricted form of the parallel operator. The more general form allows processes to selectively synchronise on events.)

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

$P;\ Q$ is a sequential, rather than parallel, composition. It behaves as P until and unless P terminates successfully: its subsequent behaviour is that of Q.

$P \setminus A$ 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.

Appendix B. A taste of a CSP Stable Failures Model

This model, described more fully in books [21,23], is an extension of the traces model which can represent nondeterministic behaviour in an elegant way. It is a simplified model in that it deals only with refusal sets rather than divergences and is restricted to finite alphabets. It supports reasoning about liveness through the use of refusal sets, it distinguishes deterministic (external) from nondeterministic (internal) choice, and it distinguishes deadlock from livelock, but it does not handle unbounded nondeterminism. The intuition for this failures model is that a process P is characterised by a set of *failures*. A *failure* is a pair (s, X) with s a finite sequence drawn from the universal set Σ of events which the process may engage in, and X a subset of Σ. The sequence s is called a *trace* and the set X is called a *refusal*. The pair model the notion that the process may engage in the trace s, after which it may refuse any event in X. If a process may nondeterministically do or refuse an event $x \in \Sigma$ after trace s, both $(s, \{x\})$ and $(s \frown \langle x \rangle, \{\})$ are failures for it. The set of *failures* $\mathcal{F}$ satisfy the following axioms.

F1. The set of traces, $\mathcal{T}$, is non-empty and prefix closed. Any failure (s, X) must have its trace s recorded in $\mathcal{T}$.

F2. $(s, X) \in \mathcal{F} \wedge Y \subseteq X \Rightarrow (s, Y) \in \mathcal{F}$
If a process can refuse all events in finite set X then it can also refuse all subsets of Y.

F3. $(s, X) \in \mathcal{F} \wedge (\forall\, c \in Y \subseteq \alpha(P) \bullet ((s \frown \langle c \rangle, \{\}) \notin \mathcal{F} \Rightarrow (s, X \cup Y) \in \mathcal{F}$
An event which is impossible as a next step can be included in a refusal set; it follows that after s, an event x must appear as a next step or in a refusal.

F4. $s \frown \langle \surd \rangle \in \mathcal{T} \Rightarrow (s \frown \langle \surd \rangle, X) \in \mathcal{F}$
If a process can terminate, it can refuse to do anything, that is, any set is refused.

To illustrate how CSP operations are defined with failures semantics, the failures for the prefixing, internal choice, external choice and parallel operations are given below.

$$\begin{aligned}
\mathcal{F}[\![a \rightarrow P]\!] &= \{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a\rangle \frown s, X) \mid (s, X) \in \mathcal{F}[\![P]\!]\} \\
\mathcal{F}[\![P \,\Box\, Q]\!] &= \{(\langle\rangle, X \mid (\langle\rangle, X) \in \mathcal{F}[\![P]\!] \cap \mathcal{F}[\![Q]\!]\} \\
&\quad \cup \{(s, X) \mid s \notin \langle\rangle \wedge (s, X) \in \mathcal{F}[\![P]\!] \cup \mathcal{F}[\![Q]\!]\} \\
\mathcal{F}[\![P \sqcap Q]\!] &= \mathcal{F}[\![P]\!] \cup \mathcal{F}[\![Q]\!] \\
\mathcal{F}[\![P \parallel_A Q]\!] &= \{(u, X \cup Y) \mid X \setminus (A \cup \{\surd\}) = Y \setminus (A \cup \{\surd\}) \\
&\quad \wedge \exists\, s, t.(s, t) \in \mathcal{F}(P) \\
&\quad \wedge (t, Y) \in \mathcal{F}(Q) \\
&\quad \wedge u \in s \parallel_A t\}
\end{aligned}$$

The parallel operator is defined so that P and Q can independently perform events outside of A, but have to cooperate on A; that is, the combination can refuse to do any event in A whenever either process can refuse it. The trace-level parallel operator $s \parallel_A t$ produces the set of all traces that could arise if P and Q respectively communicate s and t.

References

[1] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142, 1983.

[2] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *JACM*, 31:560–599, 1984.

[3] S.D. Brookes and A.W. Roscoe. An improved failures model for communicating sequential processes. In *Proc. Pittsburgh Seminar on Concurrency*. Springer, 1985.

[4] M.J. Butler. *A CSP Approach to Action Systems*. DPhil thesis, University of Oxford, 1992.

[5] M.J. Butler. An approach to the design of distributed systems with B AMN. In D. Till J. Bowen, M. Hinchey, editor, *ZUM'97*, pages 223–241. Springer, 1998.

[6] M.J. Butler. csp2b: A practical approach to combining CSP and B. In J. Woodcock J. Davies, J.M. Wing, editor, *FM99 World Congress*. Springer Verlag, 1999.

[7] R. Covington et al. Formal methods specification and verification guidebook for the verification of software and computer systems: planning and technology insertion. Tech. Report NASA-GB-002-95, vol I, NASA, 1995.

[8] Sadie Creese and Joy Reed. Verifying end-to-end protocols using induction with CSP/FDR. In *Proc. of IPPS/SPDP Workshop on Parallel and Distributed Processing*, LNCS 1586, Lisbon, Portugal, 1999. Springer.

[9] S.J. Creese and A.W. Roscoe. Verifying an infinite family of inductions simultaneously using data independence and FDR. In *Proc. of Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification, FORTE/PSTV'99 Formal Methods for Protocol Engineering and Distributed Systems*, Beijing, China, 1999. Kluwer Academic Publishers.

[10] J. Crow et al. Formal methods specification and analysis guidebook for the verification of software and computer systems. Tech. Report NASA-GB-001-97, vol II, NASA, 1997.

[11] Formal Systems (Europe) Ltd. *Failures Divergence Refinement.* User Manual and Tutorial*, version 2.11.* http://www.formal.demon.co.uk/.

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

[13] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*. Springer, 1996.

[14] G. Lowe and A.W. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Trans. Soft. Eng.*, 23(10), 1997.

[15] C. Meadows. The NRL protocol analyzer: An overview. *J. Logic Programming*, 19,20, 1994.

[16] 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.

[17] G.M. Reed and A.W. Roscoe. The timed failures-stability model for CSP. *Theoretical Computer Science*, 211:85–127, 1999.

[18] J.N. Reed and J.E. Sinclair. Compatibility conditions for combining independent specifications. in preparation.

[19] J.N. Reed, J.E. Sinclair, and F. Guigand. Deductive reasoning versus model checking: two formal approaches for system development. In K. Taguchi K. Araki, A. Galloway, editor, *Integrated Formal Methods 1999*, York, UK, June 1999. Springer Verlag.

[20] J.N. Reed, J.E. Sinclair, and G.M. Reed. Routing - a challenge to formal methods. In *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, 1999. CSREA Press.

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

[22] R.S.Lazic. *A Semantic Study of Data Independence with Applications to Model Checking.* DPhil thesis, University of Oxford, 1999.

[23] S. Schneider. *Concurrent and Real-time Systems.* John Wiley and Sons, 2000.

[24] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd ed., 1992.

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

[26] H. Treharne and S. Schneider. How to drive a B Machine. In J. Bowen, S. Dunne, and A. Galloway, editors, *ZB2000: Formal Specification and Development in Z and B*, number 1878 in LNCS, pages 188–208. Springer-Verlag, August 2000.