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 bicompositional 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 bicompositional 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 [23] 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 [15] and Meadows [16] 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 [8,11], testifies to a variety of approach 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 implementations. 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 the different views can give a fuller picture, but the question remains as to how this integration can best be achieved. Incorporating all aspects into one unified approach based around a particular formalism and tool set is a possibility, but this could result in additional complexity and obscurity for loosely coupled components. In contrast, our approach for combining different formalisms is to relate their different system views in a way which allows meaningful and independent analysis, including stepwise development through separate refinement. Our aim is to provide a formal underpinning for this approach, so that certain state-based safety properties can be handled with state-based techniques and tools, and event-based, liveness properties can be handled with model checkers. In this paper we wish to motivate and describe our approach for specifying loosely-coupled systems using different notations. We focus on systems comprising a main component specified in one notation together with a "plug-in" component specified in another notation. We indicate how to characterise plug-in relationships so that the coupled system is under control of the main component. The notations chosen are CSP [13] for event-based specification and model checking, and Action Systems [1] which allow state-based description and deductive reasoning. We show by example 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 [17], Butler [4] and Treharne and Schneider [24,25].

2 Interoperating components: an example

In an environment of distributed systems it is increasingly the case that any transaction requires the interoperation of a chain of components and services which combine to produce (hopefully) the desired result. Various components may be selected as "off the shelf" products to plug-in to our particular application. Some of these services may be beyond our control, but we still have expectations of harmonious behaviour. The correct operation of an application depends not only on the integrity of its own functions, but also on the components with which it interacts and on the interaction itself. Whilst formal verification of the entire system is not practical, there may be certain crucial interactions which warrant the extra care provided by formalism. Some components, such as security services, may have been verified in their own right and come with assurance of their behaviour. We are interested in how these assurances can be given and used.

As an example, we consider a specification for a secure database which answers requests for information from its paying customers. The database and many of its operations can be described in a natural way using a state-based approach. The system must also take into account the need for appropriate security controls. A number of useful algorithms (and implementations of these) already exist for such things, so our top level specification states its basic requirements and relies on these being satisfied by a suitable plug-in component, which as a separate concern may range from providing only simple confidentiality through to providing additional authentication and integrity. The top level specification simply needs to know that a 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. We wish to treat the functional properties of the subscription database and the security protocol as separate units which can be further developed and verified separately in different ways as appropriate. 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 requests from valid customers. Conditions such as clearance to access data can be succinctly captured and verified in this way. 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 for any particular user u being met by the receipt of a key by both the database and u. Order of receipt of keys is immaterial. 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.

Actions of database obtaining key for communication with user u

for $u \in USR$ **action** $GetKey_u$: −
$status(u) = startsession \longrightarrow status(u) := needkey$

for $u \in USR$ **action** $DBGotKey_u$ **in** $k? : KEY$: −
$status(u) = needkey \longrightarrow status(u), key(u) := haskey, k?$

Action of user u obtaining key for communication with database

for $u \in USR$ **action** $UserGotKey_u$ **in** $k? : KEY$: −
$ustatus = needkey \longrightarrow key := k?$

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

A detailed understanding of Action Systems is not required here, and the interested reader is directed to the work of Morgan [17] and that of Butler [5,6]. It is worth noting that *status*, *ustatus* 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 trace/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. General mechanisms for establishing liveness properties of parallel combinations of Action Systems and CSP specifications have been given by Butler [7,6] and Traherne and Schneider [24,25]. 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 a notion that one process is as live as another (a desirable behaviour for a plug-in component to offer its controller), and show by example that unfortunately, more deterministic stepwise 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 is as live as P, and (2) is itself 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 CSP and FDR

CSP [13] 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 comprises a process-algebraic programming language (brief overview of CSP is presented in Appendix A). A related series of semantic models capture different aspects of observable behaviours of processes: traces, failures and divergences. The simplest semantic model is the *traces* model which characterises a process as an *alphabet* $\alpha(P)$ of events, together with the set of all finite traces, $traces(P)$, 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 $\alpha(P)$; $(s, X) \in failures(P)$ 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 [18] 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. A brief description of a simple failures model with finite alphabets is presented in Appendix B. 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:

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

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 [12].

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, each component satisfying 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.

$$P = (a \rightarrow k1 \rightarrow k2 \rightarrow P) \,\Box\, (a \rightarrow k2 \rightarrow k1 \rightarrow P)$$

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 $k2$.

$$Q = a \rightarrow k1 \rightarrow k2 \rightarrow Q$$

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:

$$R = a \rightarrow k1 \rightarrow k1 \rightarrow R$$

This time R does not interact with Q in a desirable way since it does not have a acceptable pattern of response, and the result is deadlock at the third step.

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:

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

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:

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

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 (Bicompositional Relations) *Let ϕ and ρ each be a relation on X. We say that ϕ is bicompositional 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 bicompositional with S. This example also shows that the property is not symmetric since S is not bicompositional with R.

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

6 Bicompositional Refinements

We can capture the notion that a given relationship ϕ between co-operating specifications is inherited by refinements by requiring ϕ to be bicompositional with $\sqsubseteq$. We say that ϕ is bicompositional 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 bicompositional 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 defeats the purpose of refinement since all refinements of P must then in fact be equal to P. 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 $P = \bigsqcap_T x \rightarrow P$ 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 also 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 bicompositional.

Proof. A process is deadlock free iff it refines the process DF which is always willing to do something in its 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 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 bicompositional, 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 bicompositional relation between P and Q which implies that their parallel combination deadlocks on the intersection of their

joint alphabet 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 the intersection of their joint alphabet only where P' chooses.

We begin with some notation, then define a relationship which we regard as characterising the notion that one process is as live as another. We illustrate that this relationship is not in general preserved by refinement, and require that such a relation qualify as a plug-in only if it is preserved by refinement. We finally define some stronger bicompositional relations which qualify as plug-ins.

Notation For process P and set J of events, $\alpha(P)$ represents the alphabet of P, and $traces(P) \upharpoonright J$ represents the set of traces of P stripped of all events not in J, and $failures(P) \upharpoonright J$ has both traces and refusals of P stripped of any events not in J. We regard Q as a plug-in to P iff Q is allowed to refuse to do all of J after trace S (thus causing deadlock) only if P can as well:

Definition 6.2 *Q* **is as live as** *P*, for $\alpha P \cap \alpha Q = J$ means

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

This relation constrains Q to deadlock only when P might, and in particular, if P is deadlock free, then $P \parallel Q$ is deadlock free. But it is not bicompositional. For the processes defined below, Q is as live as P (both behave chaotically), but for the refinements, Q' is not as live as P'.

Example 6.3 $P = (\Box_R r \rightarrow P) \sqcap \text{STOP}$ $\quad Q = (\sqcap_R r \rightarrow Q) \sqcap \text{STOP}$
$P' = \Box_R r \rightarrow P$ $\quad Q' = \text{STOP}$

We characterise a plug-in relationship:

Definition 6.4 *Q* **plugs-in to** *P means*

(i) *Q is as live as P, and*

(ii) *if $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$, then Q' is as live as P'.*

We now identify bicompositional relations between P and Q which imply that Q plugs-in to P. 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 6.5 *Q* ***listens on*** *T* ***to*** *P*, for $T \subseteq \alpha P \cap \alpha Q$ means

$$x \in T \wedge s \frown \langle x \rangle \in traces(P) \upharpoonright J \wedge s \in traces(Q) \upharpoonright J$$

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

Theorem 6.6 *The relation* listens on T to *is bicompositional.*

Proof. Assume

(1) $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$

(2) $x \in T \wedge s \frown \langle x \rangle \in traces(P) \upharpoonright J \wedge s \in traces(Q) \upharpoonright J$

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

We must show

$x \in T \wedge s \frown \langle x \rangle \in traces(P') \upharpoonright J \wedge s \in traces(Q') \upharpoonright J$

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

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

We next define a bicompositional 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 their joint alphabet 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 bicompositional. 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 6.7 Q answers on R to P, for $R \subseteq \alpha P \cap \alpha Q = J$ means

$x \in R \wedge y \in R \wedge s \frown \langle x \rangle \in traces(P) \upharpoonright J \wedge s \in traces(Q) \upharpoonright J$

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

Theorem 6.8 *The relation* answers on R to *is bicompositional.*

Proof. Assume

(1) $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$

(2) $x \in R \wedge y \in R \wedge s \frown \langle x \rangle \in traces(P) \upharpoonright J \wedge s \in traces(Q) \upharpoonright J$

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

Let

$$x \in R \wedge y \in R \wedge s \frown \langle x \rangle \in traces(P') \upharpoonright J \wedge s \in traces(Q') \upharpoonright J$$

We must show

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

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

The next theorem establishes that for processes P and Q synchronising on the intersection of their alphabets, if Q listens to P, and Q answers P, then Q plugs-in to P.

Theorem 6.9 *If Q listens on T to P and Q answers on R for $T \cup R = \alpha(P) \cap \alpha(Q) = J$, then $(s, J) \in failures(P \parallel Q) \upharpoonright J$ only if $(s, J) \in failures(P) \upharpoonright J$.*

Proof. Assume $(s, J) \in failures(P \parallel Q) \upharpoonright J$. By *failures* model definition for the parallel operator (see Appendix), for some refusal sets X, Y, $X \cup Y = J$, $s \in traces(P) \upharpoonright J$ and $s \in traces(Q) \upharpoonright J$ and $(s, X) \in failures(P) \upharpoonright J$, and $(s, Y) \in failures(Q) \upharpoonright J$. Assume $(s, J) \notin failures(P) \upharpoonright J$. Then $X \neq J$, and since P cannot refuse all of J there must exist an $x \notin X$ such that $s \frown \langle x \rangle \in traces(P) \upharpoonright J$. There are two cases : $x \in T$ or $x \in R$. *Case 1.* Assume $x \in T$. Since Q listens on T to P, $(s, \{x\}) \notin failures(Q) \upharpoonright J$. Since $(s, Y) \in failures(Q) \upharpoonright J$, 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, $(s, R) \notin failures(Q) \upharpoonright J$. Furthermore, $(s, \{y\}) \notin failures(P) \upharpoonright J$ for any $y \in R$. By (M3), it follows that $X \cap R = \{\}$. Hence $R \subseteq Y$ and again by (M3), $(s, R) \in failures(Q) \upharpoonright J$. This contradiction proves the case and the theorem. □

Remark 6.10 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 bicompositional listening and answering relations between P and Q defined above ensure that Q acts as a suitable 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 $\alpha(P) \cap \alpha(Q) = T \cup R$ then Q plugs-in

channel $GetKey.USR$

channel $DBGotKey.KEY$

channel $UserGotKey.USR.KEY$

$$KeyX(u) = GetKey.u \rightarrow \prod_{k:KEY} (UserGotKey.u!k \rightarrow DBGotKey!k \rightarrow KeyX(u) \prod (DBGotKey!k \rightarrow UserGotKey.u!k \rightarrow KeyX(u))$$

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

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

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: an example

The relations that we have defined provide a means of ensuring that two processes can be separately developed, whilst maintaining integrity of combined behaviour. Though they may not be appropriate for all cases (see discussion below) they seem to capture the suitability of plug-in specifications for applications such as our key server.

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 separate unit. 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. If the database were described in CSP, our bicompositional requirements (or any other CSP formulated ones) could be checked directly. 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 $GetKey.USR$ to P, and Q answers on $DBGotKey.KEY$ to P, and for each $u \in USR$, Q answers on $UserGotKey.u.KEY$ to P.

The next step is to make the formal connection between the two notations to verify the properties in this case. Section 8 discusses work in this area.

7.1 CSP Refinements

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 [9,10]) 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 [15,14]. 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 Formally relating Actions Systems and CSP

Our objective for formally relating Action System and CSP specifications is to identify bicompositional relations among them ensuring suitability, and then automatically check that each component specification separately satisfy the relation. For our secure database example consisting of Action Systems specification P and CSP specification Q:

A sufficient condition for component P: P may (or may not) choose to invoke action *GetKey* for a given user u, and then be willing to receive any key k input via actions $UserGotKey.u$ and *DBGotKey*.

A sufficient condition for component Q: Q must be always willing to communicate on channel *GetKey* for any value u, and subsequently communicate on channels $UserGotKey.u$ and *DBGotKey* returning a specific key k.

The semantic links between Action Systems and CSP alluded to in the next section provide the mechanisms to integrate separate views using our relational conditions. How most effectively to check that component specifications satisfy these conditions is a challenging problem under research.

9 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 [17] 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 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 [24,25] 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 bicomposition 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.

Our notion of bicomposition 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 a significant reduction of effort. In formulating bicompositional properties we encounter 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. Other variations may be suitable in different circumstances. Further, there may be other useful paradigms characterising processes which could place constraints on input, such as insisting on receiving fresh keys. These are areas of ongoing research. 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 Mike Reed for his valuable discussions about various finer points of CSP semantics.

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 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 ; 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.)

$P \,|||\, Q$ 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 Failures Model

This model, taken from [2], 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, but it cannot distinguish deadlock from livelock nor can it handle unbounded nondeterminism. The intuition for the 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 set $\alpha(P)$ of events which the process may engage in, and X a subset of $\alpha(P)$. 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 not do an event $x \in \alpha(P)$ after trace s, both $(s, \{x\})$ and $(s \frown \langle x \rangle, \{\})$ are refusals for it. The set of *failures* $\mathcal{F}$ satisfy the

following axioms.

M1. $(\langle\rangle, \{\}) \in \mathcal{F}$

M2. $(s \frown t) \in \mathcal{F} \Rightarrow (s, \{\}) \in \mathcal{F}$

M3. $(s, X) \in \mathcal{F} \wedge Y \subseteq X \Rightarrow (s, Y) \in \mathcal{F}$

M4. $(s, X) \in \mathcal{F} \wedge (\forall\, c \in Y \subseteq \alpha(P) \bullet ((s \frown \langle c\rangle, \{\}) \notin \mathcal{F} \Rightarrow (s, X \cup Y) \in \mathcal{F}$

(M1) and (M2) state that the traces of a process form a non-empty, prefix-closed set. (M3) states that if a process can refuse all events in finite set X then it can also refuse all subsets of Y. (M4) states that 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. 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. Note that the semantic view of having individual alphabets for each process is taken in [13] primarily to simplify the use of the parallel operator $\|$. Hence in defining $a \rightarrow P$, $(P \,\Box\, Q)$, and $(P \sqcap Q)$, we assume $a \in \alpha(P)$, and $\alpha(P) = \alpha(Q)$. Thus we define $\alpha(a \rightarrow P) = \alpha(P)$, and $\alpha(P \sqcap Q) = \alpha(P \,\Box\, Q) = \alpha(P)$. However, we define $\alpha(P \parallel Q) = \alpha(P) \cup \alpha(Q)$, and our definition requires that P and Q synchronise on $\alpha(P) \cap \alpha(Q)$. Alphabets are treated differently in other CSP models, but here this simplified view suffices.

$$
\begin{aligned}
\mathcal{F}[\![a \rightarrow P]\!] &= \{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a\rangle \frown s, X) \mid (s, X) \in \mathcal{F}[\![P]\!]\} \\
\mathcal{F}[\![P \,\Box\, Q]\!] &= \{(\langle\rangle, X \mid (\langle\rangle, X) \in \mathcal{F}[\![P]\!] \cap \mathcal{F}[\![Q]\!]\} \\
&\quad \cup \{(s, X) \mid s \notin \langle\rangle \wedge (s, X) \in \mathcal{F}[\![P]\!] \cup \mathcal{F}[\![Q]\!]\} \\
\mathcal{F}[\![P \sqcap Q]\!] &= \mathcal{F}[\![P]\!] \cup \mathcal{F}[\![Q]\!] \\
\mathcal{F}[\![P \parallel Q]\!] &= \{(s, X \cup Y) \mid x \in s \Rightarrow x \in \alpha(P) \cup \alpha(Q) \wedge \\
&\quad (s \upharpoonright \alpha(P), X) \in \mathcal{F}[\![P]\!] \ \wedge (s \upharpoonright \alpha(Q), X) \in \mathcal{F}[\![Q]\!]\}
\end{aligned}
$$

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] M.J. Butler and M. Waldén. Distributed system development in B. In H. Habrias, editor, *First B Conference*, 1996.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Formal Systems (Europe) Ltd. *Failures Divergence Refinement.* User Manual and Tutorial*, version 2.11.*

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

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

[15] Gavin 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.

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

[17] 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.

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

[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] A.W. Roscoe and R.S. Lazic. Using logical relations for automated verification of data-independent CSP. In *Oxford Workshop on Automated Formal Methods ENTCS*, Oxford, UK, 1996.

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

[24] 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.

[25] H. Treharne and S. Schneider. How to drive a B machine. To appear.