Combining Independent Specifications

Joy N. Reed[1*] and Jane E. Sinclair[2]

[1] Oxford Brookes University, Oxford, UK
Email: JNReed@brookes.ac.uk Fax: +44 (0) 1865 483666
[2] University of Warwick, Coventry, UK
Email: jane@dcs.warwick.ac.uk Fax: +44 (0) 2476 573024

Abstract. We present a formal framework for characterising plug-in relationships between component specifications. A suitability requirement is defined based on the effect one component has on the other in terms of deadlock. Unlike monotonic operations such as parallel composition, not all such suitability requirements are preserved by refinement. Hence, we define the notion of a bicompositional relation between co-operating processes which is preserved by component-wise refinements. The approach is described in CSP using the 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 [Spi92] has been found to lead to the earlier discovery of design flaws. Formal modelling and verification of small security protocols such as that by Lowe and Roscoe [LR97], Lowe [Low96] and Meadows [Mea94] has revealed previously unsuspected flaws in the operation of these protocols.

Various formal methods with different theoretical bases have been proposed. 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 [RSG99] and general routing

[*] This work was supported in part by the US Office of Naval Research

protocols [RSR99]. 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 [RSG99] 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 long-term 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 identify relationships between loosely-coupled components which ensure the soundness of component-wise refinement. The notation used is CSP [Hoa85] which is particularly convenient for event-based specification and model checking. We indicate how this could be combined with a state-based notation such as Action Systems [BKS83], using their common failures divergences semantics.

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 characterising such assurances.

As an example, we consider a specification for a secure database which answers requests for information from its subscription 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.

Using Action Systems, which describe both system state and the interaction of events, we can specify a suitable top-level for the database. 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 such a notation. In fact, suitable protocols have already been described and verified in other ways, notably using CSP. We wish to "hand over" from the Action Systems to CSP for these. We consider below how this hand over can be achieved and why something more than the usual parallel composition is desirable.

3 Combining Notations in Practice

The trace/failures/divergences semantics models of CSP described in the next section provide a unifying semantics for Action Systems and CSP. Although a common semantics provides a theoretical link, it does not immediately solve the problems of using the notations together. The specifier is concerned with the effects of combining different parts of a specification and the constraints placed on each part by others. Translating the specifications to common ground is a possibility but, in general, prohibitively cumbersome. This has led to on-going research for specific areas where manageable proof conditions can be generated. Schneider and Treharne [TS99,TS] have worked with B and CSP defining a CSP driver to control operations described in B. Butler [But99] defines a general approach to combining B and CSP which merges CSP into B through translation. Both approaches provide useful insights into how the two notations can be brought together, yet neither are sufficient for our purpose. Like Schneider and Treharne, we wish to maintain the separation of notations. Like Butler, we wish to allow each to have a more general use. Our aim is to provide a framework in which "suitable" components can be "plugged" together and refined separately. In this paper, we explore a notion of "suitable plug-in".

The parallel composition operator can be used to combine CSP processes and/or Action Systems. So 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 (although possibly not easily understood), 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 2 below. If Q is a plug-in for P then at the very least we require that it should respond in a way suitable for P and not cause P to deadlock when they are run in parallel.

We formulate our suitability requirements using CSP semantics shared by both Action Systems and CSP. The issue of specialising to the different notations is returned to in later sections. We first give a brief introduction to CSP. An overview of CSP syntax and the failures model is given in the Appendix.

4 CSP and FDR

CSP [Hoa85] 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. 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 [BHR84] 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[BR85] and timed failures/divergences [RR99] have more structures allowing finer distinctions to support more powerful reasoning. Traces and failures are also defined for Action Systems [Mor90,WM90,But92] providing a semantic link to CSP. 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:

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

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 [For].

5 The Relationship between Components

We view our top-level specification as structured as a set of interoperable components which act in parallel. In particular, we are interested in components, such as the key exchange protocol which may be called upon to perform some task as part of a larger system. Although parallel composition can be used to combine any two processes, we would not regard every process as a *suitable* fulfilment of the requirements of the larger system. It is this additional concept of *suitability* that we wish to capture.

Example 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 $P \parallel Q$ conforms to one of the possibilities allowed by P. 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 an acceptable pattern of response, and the result is deadlock at the third step.

In general, Q will be suitable to plug in to P if it is prepared to co-operate with the pattern set out by P. If P allows an (external) choice then Q may resolve it. If P's actions are nondeterministic, Q must be prepared to deal with each possibility. This can be characterised in terms of deadlock: Q must not cause P to deadlock when they are run in parallel.

Further problems are encountered when considering refinement. In general, we expect that whenever a specification is good enough for some purpose, then so is any refinement. However, Example 2 shows that if we are dealing with *plug-in 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 $P \rho Q$, $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$, then by monotonicity we know that $P' \parallel Q' \sqsubseteq P \parallel Q$ – but is not necessarily true that $P' \rho Q'$.

Example 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 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 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 in a way which does not increase the opportunities for deadlock. Furthermore, we regard Q to be a *suitable plug-in* to P if for any component-wise refinements Q' and P', Q' is a plugin 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 1 (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 3. 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 addition, it is neither reflexive nor transitive.

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 cooperating 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 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 $P = \sqcap_T x \to P$ and $Q = \sqcap_T x \to Q$. 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 4 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 4. 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:

$$DF = \sqcap_{\Sigma} a \to DF$$

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 cooperating 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 2. Q **is as live as** P, for $\alpha P \cap \alpha Q = J$ means

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

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 (which both behave chaotically), but for the refinements, Q' is not as live as P'.

Example 5. $P = (\Box_R\, r \to P) \sqcap \text{STOP} \quad Q = (\sqcap_R\, r \to Q) \sqcap \text{STOP}$
$P' = \Box_R\, r \to P \quad Q' = \text{STOP}$

We characterise a plug-in relationship:

Definition 3. Q **plugs-in** P, means

1. *Q is as live as P, and*
2. *for all P', Q', 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 4. Q ***listens on*** T ***to*** P, for $T \subseteq \alpha P \cap \alpha Q$ means

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

Theorem 1. *The relation* listens on T to *is bicompositional.*

Proof. Assume

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

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

We must show

$$x \in T \wedge s \frown \langle x \rangle \in \mathit{traces}(P') \upharpoonright J \wedge s \in \mathit{traces}(Q') \upharpoonright J$$
$$\Rightarrow (s, \{x\}) \notin \mathit{failures}(Q') \upharpoonright J$$

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

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 5. Q answers on R to P, for $R \subseteq \alpha P \cap \alpha Q = J$ means

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

Theorem 2. *The relation* answers on R to *is bicompositional.*

Proof. Assume

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

$$\begin{aligned} &(2)\ x \in R \wedge y \in R \wedge s \frown \langle x \rangle \in \mathit{traces}(P) \upharpoonright J \wedge s \in \mathit{traces}(Q) \upharpoonright J \\ &\quad \Rightarrow (s, R) \notin \mathit{failures}(Q) \upharpoonright J \wedge (s, \{y\}) \notin \mathit{failures}(P) \upharpoonright J \end{aligned}$$

Let

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

We must show

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

Assume $(s, R) \in \mathit{failures}(Q') \upharpoonright J$. Then by (1), $(s, R) \in \mathit{failures}(Q) \upharpoonright J$, and furthermore, $s \frown \langle x \rangle \in \mathit{traces}(P) \upharpoonright J$ and $s \in \mathit{traces}(Q) \upharpoonright J$. Thus by (2), $(s, R) \notin \mathit{failures}(Q) \upharpoonright J$, and this contradiction establishes that $(s, R) \notin \mathit{failures}(Q') \upharpoonright J$. Assume $(s, \{y\}) \in \mathit{failures}(P') \upharpoonright J$. Again by (1), $(s, \{y\}) \in \mathit{failures}(P) \upharpoonright J$, 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 3. *If Q listens on T to P and Q answers on R for $T \cup R = \alpha(P) \cap \alpha(Q) = J$, then $(s, J) \in \mathit{failures}(P \parallel Q) \upharpoonright J$ only if $(s, J) \in \mathit{failures}(P) \upharpoonright J$.*

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

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. P nondeterministically triggers 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 to P , and if $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$ then Q' plugs-in t P. Thus we can confidently refine P and Q separately, knowing that refinements cooperate as desired.

7 Using the Definitions

The following small examples illustrate the use of the properties defined above. Components are defined using CSP.

Example 6. In the first case, P is a process which places a request with either event $a1$ or event $a2$ (chosen nondeterministically). Q is willing to accept either and provide an acceptable response, hence Q acts as a suitable plug-in to P.

$$P = (a1 \rightarrow r- > P) \sqcap (a2 \rightarrow r \rightarrow P)$$
$$Q = (a1 \rightarrow r \rightarrow Q) \,\Box\, (a2 \rightarrow r \rightarrow Q)$$

Q **listens on** $\{a1, a2\}$ **to** P – because Q has the same traces as P and refuses neither $a1$ nor $a2$ at any point where P might engage in them.
Q **answers on** $\{r\}$ **to** P – because Q is willing to provide a response whenever P expects one.

Example 7. We have a different situation if Q is able to deal with one of the possible requests only, rendering it an unsuitable plug-in to P.

$$P = (a1 \rightarrow r \rightarrow P) \sqcap (a2 \rightarrow r \rightarrow P) \qquad Q = (a1 \rightarrow r \rightarrow Q)$$

Q **does not listen on** $\{a1, a2\}$ **to** P – because $\langle a2 \rangle$ is a trace of P but $(\langle\rangle, \{a2\})$ is a failure of Q.

Example 8. The listening and answering relations are not inverses:

$$P = (a \rightarrow P) \sqcap \mathit{STOP} \qquad Q = (a \rightarrow Q)$$

Q **listens on** $\{a\}$ **to** P – because Q is willing to provide a response whenever P might expect one.
P **does not answer on** $\{a\}$ **to** Q – because $\langle a \rangle$ is a trace of Q but $(\langle\rangle, \{a\})$ is a failure of P.

Example 9. Here P requests a key which is subsequently supplied by Q. Q selects the value for the key which it outputs to P on channel *SupplyKey*. P is prepared to input any value on *SupplyKey*, hence, Q acts as a suitable plug-in to P.

$$P = \mathit{RequestKey} \rightarrow \mathit{SupplyKey}?k : \mathit{KEY} \rightarrow P$$
$$Q = \mathit{RequestKey} \rightarrow \sqcap_{k:KEY} (\mathit{SupplyKey}!k \rightarrow Q)$$

Q **listens on** $\{\mathit{RequestKey}\}$ **to** P – because Q is always ready to accept a key request whenever P issues one.
Q **answers on** $\{k : \mathit{KEY} \mid \mathit{SupplyKey}.k\}$ **to** P – because whenever P wants a key as input, it accepts any key offered, and Q is ready to offer one.

8 Using CSP Plug-Ins with Action Systems

The listening and answering relations we have identified are not completely general, but do typify a number of client-server behaviours. Their formulations in CSP failures models makes them meaningful for both Action Systems and CSP, providing a means of ensuring that two processes can be separately developed, whilst maintaining integrity of combined behaviour.

What remains is to to use the formal connection between the two notations to verify the properties for individually specified components. We are investigating practical approaches for verifying these properties for certain classes of "loosely-coupled" components illustrated by our secure database example.

For example, suppose a CSP specification for a key exchange protocol describes a process which is always willing to accept a request as input, and subsequently distribute a common key to the server and user, in nondeterministic order. It should be deliberately abstract to allow for a variety of specific key distribution protocols. This is to be viewed as a "minimum specification" of the key exchange to be refined and verified as a separate unit. We want 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.

One of the advantages of relating components using bicompositional properties is that different components can be refined independently and the properties are guaranteed to hold between all 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, with additional events hidden, it 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 [CR99]) and data independence techniques (for example [Ros98,RL96]) 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, for example, robustness against deliberate or inadvertent attacks by intruders. For example, Lowe and Roscoe [LR97] 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.

9 Conclusion

The work described in this paper is the first step towards a general framework for describing and verifying the suitability of separate components working together to form an integrated system. Components are likely to be developed separately, using different notations and techniques. It is also likely that previously-developed components would be incorporated, in which case the component should provide an abstract specification (or specifications) as a minimum guarantee of its behaviour. In this way, a formal framework can be provided for larger systems created from a variety of loosely coupled subsystems, using off-the-shelf components where possible.

We have used relational constraints to characterise our notion of minimum requirements for a plug-in component: it must operate under the control of the main component in that the combination deadlocks only at the behest of the main. The approach also applies to independent processes which are not in a master-slave relationship but which are co-operating to achieve a one-off task. There may be other properties which would be useful to incorporate, perhaps tailored to the purposes of a specific system, but the notion of plug-in components which operate in accordance with the demands of a requesting module seems to be a generally useful one. Identifying bicompositionality between specifications underlines the fact that not all desirable relationships between components would be preserved by refinement. Working with a property which is not bicompositional makes verification extremely difficult: either the property is proved at an early stage and continued effort is needed to check that the property is maintained as the system is refined, or the property is not checked till a later stage when proof is often more difficult and may reveal errors stemming from an early stage of development. The “listening” and “answering” properties provide sufficient conditions ensuring a bicompositional plug-in relationship. Another feature of these properties is that, unlike the formulation of the ”as live as” relation, there is no need to directly deal with the parallel system of the two components. However, proving trace requirements is still not completely straightforward and efficient proof techniques are a topic of on-going research.

As sufficient conditions, we are aware that the properties given here rule out some cases which might otherwise be considered acceptable and that variations on the conditions might be devised to improve the situation. 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. This is another area which we are continuing to develop. It is perhaps worth noting that in formulating bicompositional properties we encounter a difficulty similar to that of noninterference properties in security. Many different variants are possible and the effects are not always apparent until pathological examples are examined.

Another guiding principle has been the need to support different notations. The case study from which this work arose describes a secure database described using Action Systems which uses plug-in components for security services such as key exchange. The properties we have described are suitable for dealing with combinations of both CSP processes and Action Systems since traces and failures are defined for both. This allows both state-based and event-oriented specifications to be used for parts of the system to which they are best-suited and combined in a loosely-coupled manner. The approach is also applicable to other notations for which a traces/failures interpretation can be given. Again, the fact that the parallel system does not have to be calculated is an advantage. However, the identification of sub-traces within each component is still a significant task which forms part of the ongoing investigations referred to above.

As referred to in Section 3, work combining CSP with B is being undertaken by Butler [But99] and Treharne and Schneider [TS99,TS]. The main focus of these approaches has been to use CSP as a convenient way to specify constraints on the sequencing of, i.e., controlling the state-based actions. We take a different perspective, wishing to allow combinations of state-based and event-based components with either being designated as being “in control” as appropriate. However, the techniques developed for these approaches in which loop control requirements are “unwound” to give a set of verification conditions may be useful in establishing more easily provable conditions for our approach.

In common with Butler, Treharne and Schneider, our aim 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 specifications early in the development process, so that independent analysis can be effectively performed. Finally we note that our techniques are formulated using a traces/failures approach, but the 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 Taste 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. We give a selection of CSP algebraic operators for constructing processes.

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

$a \rightarrow P$ 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 synchronise on their events.

Failures Model. The set of *failures* $\mathcal{F}$ satisfies the following axioms [BHR84].

$$(\langle\rangle, \{\}) \in \mathcal{F} \tag{M1}$$
$$(s \frown t) \in \mathcal{F} \Rightarrow (s, \{\}) \in \mathcal{F} \tag{M2}$$
$$(s, X) \in \mathcal{F} \wedge Y \subseteq X \Rightarrow (s, Y) \in \mathcal{F} \tag{M3}$$
$$(s, X) \in \mathcal{F} \wedge (\forall\, c \in Y \subseteq \alpha(P) \bullet ((s \frown \langle c\rangle, \{\}) \notin \mathcal{F} \Rightarrow (s, X \cup Y) \in \mathcal{F} \tag{M4}$$

To illustrate how CSP operations are defined with failures semantics, we give the failures for the parallel operator. We take the semantic view of having individual alphabets for each process [Hoa85] here to simplify the use of the parallel operator.

$$\begin{aligned}\mathcal{F}[\![P \parallel Q]\!] = \{(s, X \cup Y) \mid x \in s \Rightarrow x \in \alpha(P) \cup \alpha(Q) \wedge \\ (s \upharpoonright \alpha(P), X) \in \mathcal{F}[\![P]\!] \ \wedge \\ (s \upharpoonright \alpha(Q), X) \in \mathcal{F}[\![Q]\!]\}\end{aligned}$$

References

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *JACM*, 31:560–599, 1984.

[BKS83] 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.

[BR85] S.D. Brookes and A.W. Roscoe. An improved failures model for communicating sequential processes. In *Proc. Pittsburgh Seminar on Concurrency.* Springer, 1985.

[But92] M.J. Butler. *A CSP Approach to Action Systems.* DPhil thesis, University of Oxford, 1992.

[But99] 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.

[CR99] 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.

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

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

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. of TACAS*, volume 1055 of *LNCS*. Springer, 1996.

[LR97] G. Lowe and A.W. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Trans. Soft. Eng.*, 23(10), 1997.

[Mea94] C. Meadows. The NRL protocol analyzer: An overview. *J. Logic Programming*, 19,20, 1994.

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

[RL96] 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.

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

[RR99] G.M. Reed and A.W. Roscoe. The timed failures-stability model for CSP. *Theoretical Computer Science*, 211:85–127, 1999.

[RSG99] 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.

[RSR99] 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.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd ed., 1992.

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

[TS99] 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.

[WM90] J.C.P. Woodcock and C.C. Morgan. Refinement of state-based concurrent systems. In *Proc. of VDM Symposium*, LNCS 428. Springer-Verlag, 1990.