Calculated Secure Processes

Michael J. Banks and Jeremy L. Jacob

Department of Computer Science, University of York
{Michael.Banks,Jeremy.Jacob}@cs.york.ac.uk

Abstract. This paper introduces a versatile operator for modifying CSP processes to satisfy particular information flow security requirements. We present and justify an algebraic semantics for this operator, which allows us to derive secure processes from (potentially) insecure processes in a calculational style. Moreover, the operator simplifies the task of verifying the security of processes.
Keywords: computer security, information flow, formal software development, CSP, semantics, confidentiality properties, secure refinement.

1 Introduction

In the broadest sense, a formal specification of a system defines a set of *correctness properties*. For an implementation of a system to satisfy its specification, it must satisfy all the correctness properties present in the specification. Perhaps the most obvious examples of correctness properties are concerned with functionality, but other non-functional requirements of software that can be precisely articulated — such as performance, reliability and resource usage — may also be interpreted as correctness properties.

Confidentiality properties are a class of non-functional correctness properties of software systems that are related to information security. Informally, a confidentiality property codifies an upper bound on what information a user of the system can deduce about the system's behaviour from its local observation of that behaviour. A system's specification may include confidentiality properties to stipulate that an implementation of the system must not (directly or indirectly) leak secret information to untrusted users.

This paper describes an approach for extending the CSP process algebra [1,2] to assist in the construction of software systems that are guaranteed to uphold specified confidentiality properties. Our main contribution is an operator over CSP processes that is parametrised by an encoding of confidentiality properties. This operator can be applied to processes in two ways. First, it may be used to verify that a process satisfies a specified confidentiality property. Secondly, the operator may be applied to any process S to *calculate* a process S' that is guaranteed to satisfy a confidentiality property.

This paper is structured as follows. Section 2 gives an overview of CSP and information flow security and outlines how CSP processes can be analysed for information leaks. Section 3 defines the algebraic semantics of the operator and Section 4 details a worked example in applying the operator to a simple process.

In Section 5, we identify techniques for verifying processes against confidentiality properties. We examine how our approach relates to the wider field of information flow security in Section 6 and summarise our work in Section 7.

2 CSP and Confidentiality

CSP (*Communicating Sequential Processes*) is a formal notation for modelling the patterns of behaviour of concurrent and distributed systems [1,2].[1] In CSP, a process interacts with its environment by engaging in *events* over an *alphabet* (set) of named channels: for example, the process $a \rightarrow b \rightarrow S$ performs an event on channel 'a' followed by an event on 'b', and then behaves according to S.

CSP features a rich algebra of operators for giving structure to processes, whose semantics are formally defined by a denotational model. In the *traces* model, a trace records the history of a process as a sequence of events. The semantics of a process S in the traces model is characterised by the prefix-closed set of all traces that S may perform: for example, if $S = a \rightarrow b \rightarrow \mathit{Stop}$, then $\mathit{traces}\,[\![S]\!] = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$.

Consider the following recursive CSP process M and its associated trace set:

$$M \triangleq (h \rightarrow l \rightarrow M) \sqcap (k \rightarrow \mathit{Stop})$$

$$\mathit{traces}\,[\![M]\!] = \left\{ \begin{array}{l} \langle\rangle, \langle h\rangle, \langle h, l\rangle, \langle k\rangle, \langle h, l, h\rangle, \langle h, l, h, l\rangle, \langle h, l, k\rangle, \\ \langle h, l, h, l, h\rangle, \langle h, l, h, l, h, l\rangle, \langle h, l, h, l, k\rangle, \ldots \end{array} \right\}$$

M non-deterministically chooses either to perform a 'h' event followed by a 'l' event and then behave as M again, or to perform a 'k' event and then refuse (forever) to perform any events (Stop).

We model a user of a process (representing a system) as an agent which can observe some — but not necessarily all — of the channels of the process. We define a user's *window* to be the subset of a process's alphabet that contains all events visible to that user. Hence, a user's observation of a trace tr is the *projection* of tr through the user's window w, denoted by $tr \upharpoonright w$. [2]

Suppose that M's environment consists of a "low-level" (untrusted) user $\mathcal{L}$ with window $L = \{l\}$. If $\mathcal{L}$ knows the structure of M then, given an observation ϕ of M viewed through L, $\mathcal{L}$ can deduce all traces of M that are consistent with ϕ. We capture this intuition by defining an *inference function* on processes: [3,4]

$$\mathsf{infer}(M, L, \phi) \triangleq \{tr \mid tr \in \mathit{traces}\,[\![M]\!] \wedge tr \upharpoonright L = \phi\} \tag{1}$$

The traces in $\mathsf{infer}(M, L, \phi)$ are *indistinguishable* from $\mathcal{L}$'s perspective: if this set contains more than one trace, then $\mathcal{L}$ cannot determine which of these traces describes the actual behaviour of M. Nevertheless, $\mathcal{L}$ may still be able to identify common features of all traces in this inference set and so deduce sensitive or valuable information about the actual behaviour of M.

We may insist that M satisfies the following security requirement:

[1] Readers unfamiliar with CSP notation may wish to consult the Appendix.

[2] $tr \upharpoonright A$ is the trace given by removing from tr all events that are absent from set A. For example, $\langle a, b, a, c, a, b\rangle \upharpoonright \{b, c\} = \langle b, c, b\rangle$.

"The occurrence of 'h' events should be kept secret from $\mathcal{L}$."

While $\mathcal{L}$ cannot observe 'h' events directly (since $h \notin L$), if $\mathcal{L}$ makes an observation $\langle l \rangle$ of M, it could deduce that a 'h' event must have occurred in M's execution by calculating the inference set associated with $\langle l \rangle$:

$$\mathsf{infer}(M, L, \langle l \rangle) = \{\langle h, l \rangle, \langle h, l, h \rangle, \langle h, l, k \rangle\} \tag{2}$$

Since all traces in $\mathsf{infer}(M, L, \langle l \rangle)$ feature an initial 'h' event, we conclude that M does not satisfy our security requirement. (Of course, M satisfies other requirements: for example, $\mathcal{L}$ cannot establish that 'k' has occurred, because for each $\mathcal{L}$ observation ϕ of M, $\mathsf{infer}(M, L, \phi)$ contains traces without a 'k' event.)

3 Securing Processes by Extension

In this section, we describe a systematic approach for extending processes to satisfy given security requirements. We write $<P, Q>(S)$ — where S, P and Q are CSP processes — to denote a process S' that behaves like S but, whenever S' can perform an activity that conforms to a behaviour of P, then S' may (at the environment's discretion) instead perform an alternative activity conforming to Q (instead of P) and then proceed to behave as S as if it had performed P.

We select the P and Q processes to express a confidentiality property. For example, we could encode the requirement that $\mathcal{L}$ cannot deduce the occurrence of 'h' events followed by 'l' events in M as a $<P, Q>$ pair by writing:

$$<h \rightarrow l \rightarrow \mathit{Skip}, l \rightarrow \mathit{Skip}>$$

Here, P specifies that an occurrence of a 'h' event immediately prior to an 'l' event is classed as confidential. Q specifies that the absence of the 'h' event represents a plausible non-confidential "cover story" for the confidential behaviour. When applied to a process, the $<P, Q>$ operator masks the occurrence of P behaviours from $\mathcal{L}$'s perspective, by inserting alternative Q behaviours that $\mathcal{L}$ cannot distinguish from P behaviours. Hence, it is essential that the projections of P and Q through $\mathcal{L}$'s window are identical. Formally:

$$\{tr \upharpoonright L \mid tr \in \mathit{traces}\, [\![P]\!]\} = \{tr \upharpoonright L \mid tr \in \mathit{traces}\, [\![Q]\!]\} \tag{3}$$

We define the semantics of $<P, Q>$ by giving a collection of algebraic laws formulated in terms of the standard CSP operators. Hence, the application of $<P, Q>$ to a process S is carried out by recursively applying these laws according to the structure of S. Since $<P, Q>$ is defined in this way, a denotational semantics for $<P, Q>$ can be calculated straightforwardly.[3]

We insist that P, Q and S are divergence-free[4] processes, to simplify our reasoning and to make certain that applying $<P, Q>$ to S can never introduce

[3] Due to space constraints, we do not provide the denotational semantics here.

[4] A process is said to *diverge* if it can perform *"an infinite sequence of internal actions"* (without interacting with its environment) [2].

divergence into S. Furthermore, we require that P and Q always terminate by performing *Skip* as their final action. This condition ensures that applying $<P, Q>$ to a terminating process produces a terminating process.

In what follows, we require a notation for describing the behaviour of a process S after having performed an activity modelled by process P. The predicate $S \,\mathsf{app}\, P$ is satisfied if and only if S features a trace prefixed by a complete trace of P (with the final '✓' generated by the *Skip* in P omitted):

$$S \,\mathsf{app}\, P \;\triangleq\; \exists\, tr \in traces\, [\![S]\!] \bullet tr \frown \langle ✓ \rangle \in traces\, [\![P]\!] \tag{4}$$

We write $S\#P$ to denote the process that behaves as S after executing any P trace that is a trace of S. Of course, $S\#P$ is well-defined only if $S \,\mathsf{app}\, P$ holds.

$S\#P$ is defined in terms of the CSP "after" operator — where S/tr represents all possible behaviours of S after behaving as tr [2, §1.3.4] — as follows:[5]

$$S\#P \;\triangleq\; \bigsqcap \{S/tr \mid tr \frown \langle ✓ \rangle \in traces\, [\![P]\!] \wedge tr \in traces\, [\![S]\!]\} \tag{5}$$

3.1 Distribution through Choice

As we saw in Section 2, $S \sqcap T$ is a process that chooses non-determistically to behave either as S or as T. Likewise, the process $S \,\Box\, T$ can behave as S or as T, but allows the environment to select between them.

The semantics of $<P, Q>$ applied to processes that offer a choice of multiple initial events is given by distributing $<P, Q>$ across each branch of the choice:

Law 1

$$<P, Q>(S \sqcap T) \;\triangleq\; <P, Q>(S) \sqcap <P, Q>(T) \tag{6}$$

$$<P, Q>(S \,\Box\, T) \;\triangleq\; <P, Q>(S) \,\Box\, <P, Q>(T) \tag{7}$$

3.2 *Stop* and *Skip*

Two fundamental processes in CSP are *Stop* (deadlock) and *Skip* (termination). Applying $<P, Q>$ to *Stop* or *Skip* has no effect, because *Stop* can never perform any P activity. Likewise, since *Skip* can only ever extend the trace with the (invisible) signal event ✓, *Skip* cannot perform any P activity.

Law 2 *Stop and Skip are unaffected by applying* $<P, Q>$.

[5] Notation: $\bigsqcap\{S_1, \ldots, S_n\}$ may be read as $S_1 \sqcap \ldots \sqcap S_n$.

3.3 Prefixing

We now consider the semantics of $<P, Q>(a \rightarrow S)$, where 'a' is any event.

If P cannot perform any events — that is, if $P = \mathit{Skip}$ — the semantics of $<P, Q>$ are not defined by the laws below. We consider such a choice for P improper, because it does not represent any confidential activity.

Law 3 states that if $a \rightarrow S$ cannot behave as P (as when 'a' is not an initial event of P), then we can safely move the $<P, Q>$ operator to S.

Law 3 *Provided that* $\neg\,(a \rightarrow S)\,\mathsf{app}\,P$*:*

$$<P, Q>(a \rightarrow S) \triangleq a \rightarrow <P, Q>(S) \tag{8}$$

We need a prefix law to handle cases where $a \rightarrow S$ may perform an activity encoded by P (i.e. $(a \rightarrow S)\,\mathsf{app}\,P$ holds). In such cases, we require that $a \rightarrow S$ is extended with the behaviours of Q to ensure that $\mathcal{L}$ cannot determine that a P behaviour has occurred. When P and Q do not share the same initial events, the operator should extend $a \rightarrow S$ to offer the environment the choice of performing Q instead of P. This effect is codified by Law 4.

Law 4 *Provided that* $S\,\mathsf{app}\,X$ *holds:*

$$<a \rightarrow X, b \rightarrow Y>(a \rightarrow S) \triangleq \begin{pmatrix} a \rightarrow <a \rightarrow X, b \rightarrow Y>(S) \\ \Box\ b \rightarrow Y \fatsemi <a \rightarrow X, b \rightarrow Y>(S \# X) \end{pmatrix} \tag{9}$$

Law 4 allows $S' = <a \rightarrow X, b \rightarrow Y>(a \rightarrow S)$ to behave as $b \rightarrow Y$ whenever it can behave as $a \rightarrow X$. After behaving as $b \rightarrow Y$, S' reverts to behaving as it would have done *after* performing as $a \rightarrow X$. Law 4 employs deterministic choice between $a \rightarrow X$ and $b \rightarrow Y$ to ensure that S' must always offer $b \rightarrow Y$ to the environment whenever it can offer $a \rightarrow X$.

Up to now, we have considered cases where confidential activities are masked by adding appropriate cover story activities. Alternatively, a cover story may take the form of the *absence* of a confidential activity by setting $Q = \mathit{Skip}$. The semantics of $<P, \mathit{Skip}>(a \rightarrow S)$ are not given by Law 4. Hence, we introduce a new law to accommodate cases where $Q = \mathit{Skip}$.

Law 5 *Provided that* $S\,\mathsf{app}\,X$ *holds:*

$$<a \rightarrow X, \mathit{Skip}>(a \rightarrow S) \triangleq \begin{pmatrix} a \rightarrow <a \rightarrow X, \mathit{Skip}>(S) \\ \Box <a \rightarrow X, \mathit{Skip}>(S \# X) \end{pmatrix} \tag{10}$$

Assuming that $\mathcal{L}$ cannot distinguish $a \rightarrow X$ from Skip, this law generates a process that masks the occurrence of $a \rightarrow X$ from $\mathcal{L}$'s perspective.

3.4 Operator Disposal

If we can prove that S never behaves according to P (i.e. S never performs a confidential activity at any point in its execution), we can be sure that it is unnecessary to introduce the cover story Q within S. If this condition holds, then we can safely remove the $<P, Q>$ operator from S.

Law 6 *When* $\forall\, s \bullet \neg\, (S/s)\,\mathsf{app}\,P$ *holds,* $<P, Q>(S)$ *is equal to* S.

This law can be justified by appealing to the repeated application of Law 3 to the expansion of S, in the context of the other laws.

4 Worked Example

Together, these laws are sufficient for applying the $<P, Q>$ operator to simple CSP processes. Let $M' = <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>(M)$. We derive the CSP process equal to M' — using the laws of $<P, Q>$ and CSP — as follows:

$$
\begin{array}{ll}
& <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>(M) \\
= & \quad \{\ \text{definition of } M\ \} \\
& <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>((h \to l \to M) \sqcap (k \to \mathit{Stop})) \\
= & \quad \{\ \text{distribute operator through internal choice (Law 1)}\ \} \\
& \left(\begin{array}{l} <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>(h \to l \to M) \\ \sqcap\ <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>(k \to \mathit{Stop}) \end{array}\right) \\
= & \quad \{\ k \to \mathit{Stop} \text{ cannot behave as } h \to l \to \mathit{Skip}\ \text{(Law 3)}\ \} \\
& \left(\begin{array}{l} <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>(h \to l \to M) \\ \sqcap\ k \to <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>(\mathit{Stop}) \end{array}\right) \\
= & \quad \{\ \text{applying to } \mathit{Stop} \text{ has no effect (Law 2)}\ \} \\
& <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>(h \to l \to M) \sqcap k \to \mathit{Stop} \\
= & \quad \{\ h \to l \to M \text{ is confidential (Law 4)}\ \} \\
& \left(\begin{array}{l} h \to <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>(l \to M) \\ \Box\ l \to \mathit{Skip} \fatsemi <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>((l \to M) \# (l \to \mathit{Skip})) \end{array}\right) \\
& \sqcap\ k \to \mathit{Stop} \\
= & \quad \{\ \text{carry prefix (Law 3); unfold } S \# P\ \text{(Equation 5)}\ \} \\
& \left(\begin{array}{l} h \to l \to <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>(M) \\ \Box\ l \to \mathit{Skip} \fatsemi <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>(\bigsqcap\{M\}) \end{array}\right) \sqcap k \to \mathit{Stop} \\
= & \quad \{\ \text{simplify}\ \} \\
& \left(\begin{array}{l} h \to l \to <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>(M) \\ \Box\ l \to <h \to l \to \mathit{Skip}, l \to \mathit{Skip}>(M) \end{array}\right) \sqcap k \to \mathit{Stop} \\
= & \quad \{\ \text{fold } M' \text{ (twice)}\ \} \\
& (h \to l \to M' \ \Box\ l \to M') \sqcap k \to \mathit{Stop}
\end{array}
$$

The inference set associated with the $\mathcal{L}$ observation $\langle l \rangle$ of M' is:

$$\mathsf{infer}(M', L, \langle l \rangle) = \{\langle l \rangle, \langle l, k \rangle, \langle h, l \rangle, \langle h, l, h \rangle, \langle h, l, k \rangle\}$$

Observe that $\mathsf{infer}(M, L, \langle l \rangle) \subseteq \mathsf{infer}(M', L, \langle l \rangle)$ (see Equation 2). Hence, if $\mathcal{L}$ observes $\langle l \rangle$, it cannot deduce any more about the behaviour of M' as it could

about the behaviour of M. More importantly, $\mathsf{infer}(M, L, \langle l \rangle)$ features traces free of 'h' events, which implies that $\mathcal{L}$ cannot establish that M' has performed 'h', in keeping with our security requirement. In the next section, we present theorems for proving that this requirement is upheld for *all* of $\mathcal{L}$'s observations of M'.

5 Verification and Refinement

We claim that $<P, Q>(S)$ must satisfy the confidentiality property encoded by P and Q by appealing to the following theorem.

Theorem 1 ($\mathcal{L}$ cannot deduce P). *Given any L-observation ϕ of $<P, Q>(S)$, $\mathcal{L}$ can never establish (with certainty) that a P activity has taken place.*

This theorem is justified from the semantics of $<P, Q>$ by establishing that the process $<P, Q>(S)$ always permits a Q activity to be performed in place of a P activity. Since we insist that P and Q yield the same observations to $\mathcal{L}$ (Equation 3) — and that after performing Q, $<P, Q>(S)$ behaves as it would after performing P (Law 4) — it must be the case that if $\mathcal{L}$'s inference set contains a trace $s \frown p \frown s'$ (where $p \frown \langle \checkmark \rangle \in \mathit{traces}\, [\![P]\!]$), then it also contains a trace $s \frown q \frown s'$ (where $q \frown \langle \checkmark \rangle \in \mathit{traces}\, [\![Q]\!]$). It follows that $\mathcal{L}$ can never deduce from its observation of S that a confidential P activity has been performed.

$<P, Q>$ is an idempotent operator. (The proof follows by applying structural induction to each of the laws presented in Section 3.) It is therefore necessary to apply $<P, Q>$ to a process just once, to obtain a process that respects Theorem 1. We can appeal to this result to characterise secure processes algebraically.

Theorem 2 (Verifying Security). *Given a process S such that $<P, Q>(S) = S$, S must satisfy the confidentiality property encoded by $<P, Q>$.*

We now discuss the relationship between the $<P, Q>$ operator and refinement. It has long been known that refining a secure process may result in an insecure process [4,5]. This problem arises because non-determinism may serve two purposes in process specifications: to avoid describing "don't-care" implementation details (*underspecification*) and to ensure the behaviour of the process is *unpredictable* from $\mathcal{L}$'s perspective. It follows that naïvely resolving non-determinism within a process S (by refinement) may result in the removal of cover story behaviours from S. In turn, this may introduce new sources of information flow to $\mathcal{L}$ — that were absent from S — that allow $\mathcal{L}$ to deduce the presence of confidential activity in the refined process.

The standard CSP refinement relation is failures-divergences refinement [2], which corresponds to the resolution of (finite) non-determinism in processes: $S \sqcap T$ is refined by S (and T).

The process M' (see Section 4) can be refined to:

$$M'_0 \triangleq (h \rightarrow l \rightarrow M'_0 \,\Box\, l \rightarrow k \rightarrow \mathit{Stop}) \qquad \text{or} \qquad M'_1 \triangleq k \rightarrow \mathit{Stop}$$

Both of these processes satisfy our security requirement, since $\mathcal{L}$ cannot deduce the occurrence of a 'h' event by observing either of these processes through its window. It is perhaps tempting to claim that all refinements of a process $<P, Q>(S)$ satisfy the confidentiality property encoded by $<P, Q>$, but this is not so. Consider the process:

$$M_2' \triangleq (h \rightarrow l \rightarrow k \rightarrow \mathit{Stop}) \,\Box\, l \rightarrow (h \rightarrow l \rightarrow k \rightarrow \mathit{Stop} \,\Box\, l \rightarrow k \rightarrow \mathit{Stop})$$

M_2' is a refinement of M'. However, if $\mathcal{L}$ makes the observation $\langle l, k \rangle$ of M_2', then it can deduce that 'h' must have occurred, since the inference set associated with that observation contains only the trace $\langle h, l, k \rangle$. This result indicates that it may be necessary to re-apply the $<P, Q>$ operator after performing refinement.

6 Related Work

The canonical notion of information flow security is *noninterference*, which characterises the absence of any information flow about a high-level user's interactions with a system to low-level users [6]. While approaches for verifying that systems satisfy noninterference-like properties have been studied extensively, they are not widely used in practice, because it is often necessary to allow users to exchange data. Our confidentiality properties are weaker than noninterference, but are better suited for capturing practical security requirements than noninterference, since they permit non-secret information to flow to low-level users.

Our formalisation of confidentiality properties is loosely based on work by Mantel [7], who devised a "schema" condition for verifying that a system does not leak confidential information, expressed in terms of a set of confidential traces and a mapping from confidential traces to cover story traces. We specialise this approach by directly encoding confidential and cover story activities as processes. While the $<P, Q>$ operator is less general than Mantel's schema, it enables us to manipulate a CSP process using CSP itself and thereby allows us to provide an algebraic characterisation of whether a process is secure in Theorem 2.

Our method of verifying that a process satisfies a confidentiality property is related to Roscoe et al.'s *low-level determinism* test for CSP processes [8]. This test identifies whether a low-level user's observations of a process S can be encoded as a deterministic process: if so, the actions of other users cannot influence the observations of S made by $\mathcal{L}$, which implies that S satisfies noninterference. In contrast, we insist only that $\mathcal{L}$'s observations cannot be perturbed by the occurrence of confidential activities.

Finally, and more generally, there is a connection between our operator and *aspect-oriented programming* (AOP). AOP encourages software developers to implement the secondary *aspects* of a program (such as logging or user authorisation checks) separately from its core functionality [9]. A variant of our operator could potentially be used to support the construction of systems in an AOP style: for instance, the first argument of the operator would specify the "join points" in a program at which an aspect should be triggered, while the second argument would specify an aspect's behaviour.

7 Conclusion

In this paper, we have defined an algebraic operator for rewriting CSP processes to ensure they uphold particular security requirements. Applying this operator to a process guarantees that the resulting process satisfies the confidentiality property encoded by the operator. This suggests that the operator holds promise for constructing software that is "secure by design".

A potential drawback of using the operator is that adding cover stories to a process may violate functional requirements on the process's behaviour (such as safety properties). This difficulty may be mitigated by carefully selecting the cover stories associated with a confidential activity. However, it may be impossible to avoid this difficulty altogether when developing systems to satisfy both functionality and confidentiality requirements, since these requirements place opposing constraints on the information flow from a system to its users [4]. Indeed, there is a trade-off between functionality and confidentiality requirements: if the customer specifies a strong confidentiality property (such as noninterference) for a system, then it may be necessary to weaken the functional requirements on the system's design (and vice versa).

We believe that the application of the operator to CSP models exhibiting finite behaviour can be (partially) automated, with the assistance of model-checking tools (such as FDR [10]) to determine the points in a process's execution where it can perform confidential activities. However, the problem of automatically analysing a larger CSP model in this way may be intractable or undecidable, especially in the presence of state variables or unbounded nondeterminism. Furthermore, we have left the semantics of the operator undefined for processes expressed using the full algebra of CSP. An important topic for future work is to identify laws for applying the operator to concurrent processes.

On a final note, taking a formal approach to security engineering can help us to gain confidence that a computer system does not leak secret information to low-level users, but it is unwise to assume that any system implementation is secure in all circumstances. In particular, we have not addressed potential sources of information leakage within a system's implementation (such as its responsiveness or power consumption) that are not modelled by its specification.

Acknowledgements. Michael Banks is supported by an EPSRC DTA studentship. Thanks to the anonymous referees for their helpful comments and suggestions for making the paper more accessible to the YDS audience.

References

1. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science. Prentice Hall, Inc. (1985)
2. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)
3. Jacob, J.L.: Security specifications. In: Proceedings of the 1988 IEEE Symposium on Security and Privacy. (1988) 14–23

4. Jacob, J.L.: On the derivation of secure components. In: Proceedings of the 1989 IEEE Symposium on Security and Privacy, IEEE Computer Society (1989) 242–247
5. Roscoe, A.W.: CSP and determinism in security modelling. In: Proceedings of the 1995 IEEE Symposium on Security and Privacy, IEEE Computer Society (1995) 114–127
6. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proceedings of the 1982 IEEE Symposium on Security and Privacy, IEEE Computer Society (April 1982) 11–20
7. Mantel, H.: A Uniform Framework for the Formal Specification and Verification of Information Flow Security. PhD thesis, Universität Saarbrücken (July 2003)
8. Roscoe, A.W., Woodcock, J.C.P., Wulf, L.: Non-interference through determinism. In: ESORICS '94: Proceedings of the Third European Symposium on Research in Computer Security. Volume 875 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (1994) 33–53
9. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP'97 Object-Oriented Programming. Volume 1241 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1997) 220–242
10. FSE: FDR2 User Manual. Formal Systems (Europe) Ltd. (June 2005)
11. Davies, J.: Using CSP. In: Refinement Techniques in Software Engineering. Volume 3167 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2006) 64–122

A CSP Notation

This appendix describes some of the fundamental CSP operators informally. (For a more comprehensive description, the reader is directed to the various introductory tutorials on CSP that are available, such as Davies' tutorial [11].)

Prefix The process $a \rightarrow S$ waits until the environment is ready to accept an 'a' event, then performs an 'a' event and subsequently behaves as S.

Non-deterministic choice The process $S \sqcap T$ non-deterministically chooses to behave either as S or as T. The environment cannot influence how the non-determinism is resolved.

Deterministic choice The process $S \ \Box \ T$ offers the environment a choice between S and T, wherever possible. For instance, $a \rightarrow S \ \Box \ b \rightarrow S$ accepts either an 'a' or a 'b' event, before behaving as S.

Sequence The process $S \fatsemi T$ behaves as S until reaching *Skip* (Section 3.2), whereupon the process continues to behave as T.

In addition, CSP features a variety of operators for composing processes in parallel, hiding the occurrence of internal events from the environment, conditional choice (if-then-else) and iteration.