Using transition traces to model a security protocol.

Stephen Brookes
Carnegie Mellon University

March 12, 2003

1 Introduction

Security protocols are often difficult to specify formally and hard to prove correct, because of the potentially complex patterns of interaction between processes executing in parallel. Many people have proposed the use of formal methods in such applications (cf. [13, 14, 15, 16, 18, 23, 24]). For example, Roscoe and his colleagues have used the model checker FDR [10], based on the failures-divergences semantics of CSP, to discover bugs in various key-exchange protocols [21]. Schneider and Sidiropoulos [25] used FDR to specify and verify an "anonymity" property of a security protocol known as the "dining cryptographers" [8].

Semantically-based tools and methodologies such as these have tended to be *paradigm-specific*, based on a particular choice of programming or specification language. It is difficult to adapt tools embedded in one parallel paradigm (like CSP) to problems concerning other parallel paradigms (such as shared-variable programs); at a more abstract level, semantic models for shared-variable programming languages have so far had little in common with semantic models for CSP-like languages. Yet we might want to prove the correctness of a security protocol concerning protection of private data, couched in terms of shared variable parallelism, with respect to a specification phrased in terms of communication patterns and written in CSP. It would be difficult even to say precisely what such a correctness criterion means if the underlying semantic models for the two paradigms differ. This motivates the

use of a *unifying semantic framework* capable of modelling parallel systems in a variety of paradigms.

In our recent work we have introduced a semantic framework based on transition traces, suitable for modelling both shared-variable parallel programs and asynchronous networks of communicating processes [2, 3, 4, 5]. This semantics supports compositional analysis of safety and liveness properties, and incorporates fairness, which is vital in establishing liveness properties. Our semantics validates a number of useful laws of program equivalence, including laws which reflect the underlying fairness assumption.

Because we employ such a simple semantic infrastructure we believe that our approach can lead to simpler proofs – even comprehensible manual proofs in reasonably sized examples – and better understanding of the principles behind correct protocol design. In particular one does not always need to rely solely on a " box" such as a model checker. This is not intended to belittle the role played by model checking: obviously many complex systems cannot easily be analyzed "manually" and the support of automated tools is generally beneficial. However there is still a role for semantics to play in identifying general laws of program equivalence that can be used to justify protocol design or network design. Indeed, we expect that such laws may prove useful in improving the efficiency of model checkers; note that (for instance) the FDR model checker relies on a collection of CSP laws in deriving a normal form for finite-state CSP programs [1].

To illustrate our ideas we outline our specification and proof of correctness for the dining cryptographers protocol, which differs significantly from the specification and automated analysis in [25].

2 Outline of paper

Notation

We use an amalgam of CSP and a parallel Algol-like language, as in Idealized CSP [4]. We assume that communication is *asynchronous*, so that an output command is always enabled but an input command is blocked if the relevant channel is empty. One can model synchronized communication within this framework by means of an obvious request-acknowledge protocol. We also allow processes to share variables. The programming language is (implicitly)

strongly typed; for example, a variable of type **chan**[**int**] represents a channel capable of transmitting integer values.

The notation **local** h **in** P denotes a process P equipped with a local variable h; processes outside the scope of this declaration for h do not affect this variable. When h represents a *channel* we write **local** $h = \rho$ **in** P to indicate that the contents of h is initialized to the (finite sequence) ρ. When ρ is the empty sequence this coincides with the previous notation.

2.1 Transition traces

A process P denotes a set of *transition traces* [2, 3, 4, 5]. A finite trace has the form $(s_0, s_0')(s_1, s_1') \ldots (s_n, s_n')$ and represents an interactive computation of P starting in state s_0, ending in s_n'. Each step (s_i, s_i') represents the effect of a finite sequence of atomic actions by P, and each interference step from s_{i-1}' to s_i represents a sequence of atomic actions by the process's environment. An infinite trace represents an infinite interactive computation, assuming (weakly) fair interaction between the process and environment.

A denotational description of the trace semantics of processes is given in [2, 4]. The traces of a network are *fair merges* of traces of its component processes [19]. The traces of **local** $h = \rho$ **in** P are obtained by projection (ignoring the h-component) from traces of P in which the initial value of h is ρ and the contents of h is never altered across step boundaries, since the environment has no access to h.

The trace set of P is closed under two natural conditions that reflect our use of a step to stand for an arbitrary finite sequence of atomic actions: *stuttering* and *mumbling*. For instance, whenever $\alpha\beta$ is a trace of P and s is a state, the trace $\alpha(s, s)\beta$ obtained by inserting an extra "stuttering" step is also possible for P; and whenever P has a trace $\alpha(s, s')(s', s'')\beta$ containing adjacent steps with the same intermediate state, it also has the trace $\alpha(s, s'')\beta$ in which these steps are "mumbled" together. These closure conditions enable our model to validate a number of laws of program equivalence that fail in more finely grained semantics [19].

We incorporate channels directly into the state, essentially treating a channel as a variable capable of holding a finite sequence of data values; an input or output action is modelled as a state change. The potential for deadlock is modelled by a form of infinite stuttering.

2.2 Laws of equivalence

Rather than relying on semantic definitions directly we list a number of useful laws of program equivalence validated by our semantics [5]. Each law, written as an equation of form $P_1 = P_2$, should be interpreted as asserting that the traces of P_1 coincide with the traces of P_2.

Scope contraction

- **local** h **in** $(P \| Q) \;=\; (\textbf{local}\ h\ \textbf{in}\ P) \| Q$
  if h does not occur free in Q.

Local input

- **local** $h = v\rho$ **in** $P \| (h?x; Q) \;=\; \textbf{local}\ h = \rho\ \textbf{in}\ P \| (x{:=}v; Q)$
  provided $h?$ does not occur free in P.

We write $v\rho$ for the sequence with head v and tail ρ.

Local output

- **local** $h = \rho$ **in** $P \| (h!v; Q) \;=\; \textbf{local}\ h = \rho v\ \textbf{in}\ P \| Q$
  provided $h!$ does not occur free in P.

We write ρv for the sequence obtained by appending the item v onto ρ.

These *local laws* show that under certain circumstances we lose no generality in assuming that a suitably enabled communication involving a local channel occurs immediately, regardless of the enabledness of other activity.

Global promotion

- **local** $h = \epsilon$ **in** $(h?x; P) \| (Q_1; Q_2) \;=\; Q_1; \textbf{local}\ h = \epsilon\ \textbf{in}\ (h?x; P) \| Q_2$
  if h does not occur free in Q_1.

The soundness of this law relies crucially on fairness. It can be used to simplify reasoning in cases where local channels are used to enforce synchronization. The significance of this law, and of its obvious generalization to the case when there are several components waiting on local channels, is that it allows one to "promote" an initial segment of code performable by a parallel

component, provided that code is "global" in that it does not affect any local channel, and provided the "rest" of the parallel composition is waiting on a local channel that is currently empty. This permits us to assume that the "visible" piece of code happens first, even though operationally what really happens is that the two parallel components are interleaved fairly. No generality is lost because the blocked component contributes only stuttering steps to the traces, and these steps are absorbed by the closure conditions.

2.3 Dining cryptographers

We now put our semantics to use in the analysis of a security protocol from the literature [8]. The scenario involves three cryptographers sharing a meal around a circular table in an expensive restaurant. Each is told (secretly) by their employer (NSA) if he must pay. At most one is told to pay, and is then responsible for the entire bill. If no-one is told to pay the NSA will pick up the tab. The cryptographers want to discover collectively if NSA is paying, while retaining anonymity: no individual, when told not to pay, should be able to determine which of the other two has been told to pay.

In Chaum's protocol [8] each cryptographer tosses a private coin and shows it to his right-hand neighbor. Each examines the two coins he can see and votes publicly on the relationship between these coins. If told not to pay, he lies about whether the coins agree. If told to pay, he tells the truth. The participants tally the votes; the parity of the number of **true** votes predicts if lunch is free.

We model the cryptographers $C[i]$ $(i = 0, 1, 2)$ as processes:

$$
\begin{array}{l}
\textbf{local } \mathit{coin}[i], c, p \textbf{ in} \\
\quad \textbf{begin} \\
\qquad \mathit{toss}(\mathit{coin}[i]); \\
\qquad \mathit{look}[i \ominus 1]!\mathit{coin}[i];\ \mathit{look}[i]?c;\ \mathit{pay}[i]?p; \\
\qquad \mathit{vote}[i]!(p \textbf{ xor } \mathit{coin}[i] \textbf{ xor } c) \\
\quad \textbf{end}
\end{array}
$$

Here *toss* is a random boolean assignment, the *look* channels link each participant to his right-hand neighbor, the *pay* channels link up with the NSA, and the *vote* channels are "public". We write $\ominus$ and $\oplus$ for subtraction and addition, *modulo* 3, and **xor** is "exclusive-or".

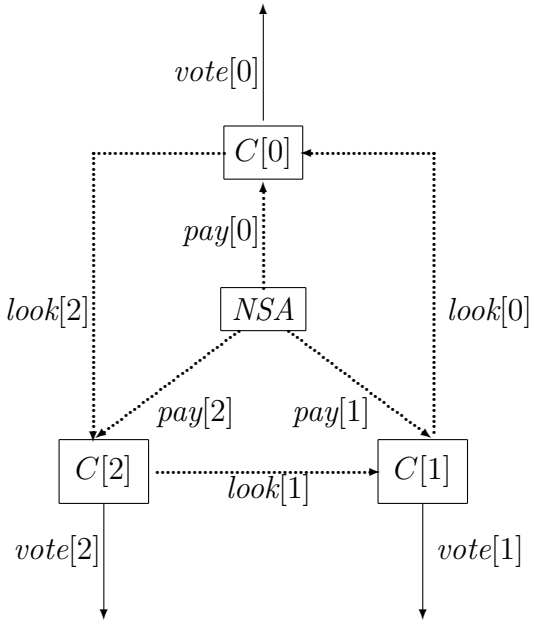

Figure 1: Three dining cryptographers

We represent the NSA process as a non-deterministic choice:

NSA $=$ PAYS **or** FREE
PAYS $=$ PAYS[0] **or** PAYS[1] **or** PAYS[2]
PAYS[i] $=$ $pay[i]!\mathbf{true} \parallel pay[i \oplus 1]!\mathbf{false} \parallel pay[i \ominus 1]!\mathbf{false}$
FREE $=$ $pay[0]!\mathbf{false} \parallel pay[1]!\mathbf{false} \parallel pay[2]!\mathbf{false}$

The network is shown pictorially in Figure 1, with "internal" channels drawn as dotted lines.

The fact that the cryptographers can tell if the meal is free is deducible from the following characterizations of trace sets, which are straightforward to derive from the laws given above:

$$\textbf{local } pay, look \textbf{ in } \text{PAYS}[i] \parallel C[0] \parallel C[1] \parallel C[2] = \bigcup\{(vote[0]!v_0 \parallel vote[1]!v_1 \parallel vote[2]!v_2) \mid \#\mathbf{true}\{v_0, v_1, v_1\}\text{is odd}\}$$

$$\textbf{local } pay, look \textbf{ in } \text{FREE} \parallel C[0] \parallel C[1] \parallel C[2] = \bigcup\{(vote[0]!v_0 \parallel vote[1]!v_1 \parallel vote[2]!v_2) \mid \#\mathbf{true}\{v_0, v_1, v_1\}\text{is even}\}$$

Note that the votes can come in any order, and no-one is able to deduce any secret information based on the voting sequence.

In establishing these characterizations much use is made of the *local input* and *local output* laws, since most of the activity in the network is carried out "locally"; we thus avoid having to deal with arbitrary interleavings of "internal" actions.

The desired anonymity property amounts to the following equivalences ($i = 0, 1, 2$):

$$L_i[\mathrm{PAYS}[i \ominus 1]] \;=\; L_i[\mathrm{PAYS}[i \oplus 1]],$$

where

$$\begin{aligned} L_i[-] \;=\; & \mathbf{local}\ look[i \ominus 1], look[i \oplus 1], pay\ \mathbf{in} \\ & [-] \parallel C[0] \parallel C[1] \parallel C[2]. \end{aligned}$$

Again it is straightforward to show that both processes have trace sets of the same form:

$$\begin{aligned} & (look[i]?b_{i\oplus 1};\ vote[i]!(b_i\ \mathbf{xor}\ b_{i\oplus 1})) \\ \parallel\ & (look[i]!b_{i\oplus 1};\ vote[i \oplus 1]!(\neg b_{i\oplus 1}\ \mathbf{xor}\ b_{i\ominus 1})) \\ \parallel\ & vote[i \ominus 1]!(b_{i\ominus 1}\ \mathbf{xor}\ b_i) \end{aligned}$$

where $b_0, b_1, b_2 \in \{\mathbf{true}, \mathbf{false}\}$. Certain timing constraints are implicitly present in these descriptions, notably that in any interleaving the output on channel $look[i]$ (done by $C[i \oplus 1]$) must precede the input (done by $C[i]$). Again the secrets are not deducible from the order in which events happen.

The case when four or more cryptographers are involved offers further potential for trouble. See Figure 2 for a picture of the network structure. Again local channels are drawn as dotted lines.

As before we have the same kind of anonymity as above, and a similar proof suffices to show this. However, there is now the possibility of *collusion* between two cryptographers, in an attempt to determine which of the other two is the payer.

If two neighbors collude they do not gain enough information to break anonymity. This is shown by the following equivalence:

$$C_{i,i\oplus 1}[\mathrm{PAYS}[i \oplus 2]] \;=\; C_{i,i\oplus 1}[\mathrm{PAYS}[i \oplus 3]],$$

where

$$\begin{aligned} C_{i,i\oplus 1}[-] \;=\; & \mathbf{local}\ look[i \oplus 2], look[i \oplus 3], pay\ \mathbf{in} \\ & [-] \parallel C[0] \parallel C[1] \parallel C[2] \parallel C[3]. \end{aligned}$$

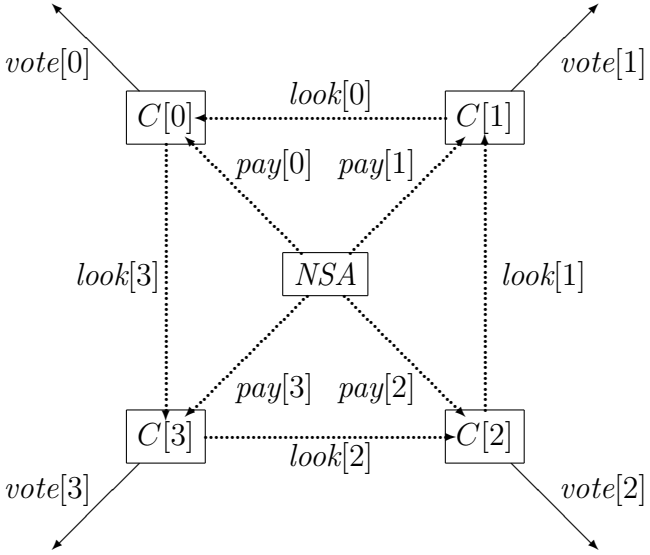

Figure 2: Four dining cryptographers

Again a simple "normal form" for the trace sets can be derived.

However, if cryptographers sitting in opposite chairs collude they *can* break anonymity. The trace semantics shows this too, since

$$C_{i,i\oplus 2}[\mathrm{PAYS}[i \oplus 1]] \;\neq\; C_{i,i\oplus 2}[\mathrm{PAYS}[i \ominus 1]],$$

where

$$\begin{aligned} C_{i,i\oplus 2}[-] \;\; = \;\; & \mathbf{local}\ look[i \ominus 1], look[i \oplus 1], pay\ \mathbf{in} \\ & [-] \parallel C[0] \parallel C[1] \parallel C[2] \parallel C[3]. \end{aligned}$$

Of course in this particular example the danger of cross-table collusion is intuitively obvious: between them the two opposite cryptographers get to see all four of the coins. Nevertheless our semantics lends itself naturally to specification of the desired security and correctness properties, and the succinctness of the representations given for trace sets is appealing. Our characterization of anonymity as a program equivalence is rather different from the way FDR is used in [25]. It seems very natural that an anonymity property like this turns out to be expressible as a program equivalence, and we believe this idea is rather widely applicable. Our approach thus offers a fresh way to look at this kind of problem.

The benefits of using a flexible, simple and widely applicable semantic foundation should be significant. Our semantic model has the advantage of building in fairness, and supports laws of program equivalence that can help tame the combinatorial explosion. In cases such as this we are able to give a proof couched in easily comprehensible terms, rather than resorting immediately to the use of a model checker. It should also be noted in any case that the use of a model checker does not come for free: the task of converting a specification into a format acceptable as input to the model checker is error-prone and time-consuming. Moreover, while model checking is applicable only for finite-state programs, our semantics can also be used to reason about infinite-state programs and can be used to reason inductively about parameterized families of networks. Of course in larger-scale examples the need for automation may be more compelling.

2.4 Further work

Laws of program equivalence such as *global promotion* and *local input/output* can be used to help tame the combinatorial explosion inherent in naive exploration of the state space of finite-state systems. Such laws enable us to reason "as if" execution happens in a particularly helpful order, without loss of generality. We will show how this can lead to straightforward and streamlined proofs of correctness for a variety of communications protocols and security protocols from the literature. For example, we can prove the correctness of the *alternating bit protocol*, assuming that a message can be dropped or duplicated an arbitrary finite number of times. We can prove directly, using our semantically-based laws, that the system consisting of sender and receiver processes and the lossy communication medium, with all internal channels localized, is *exactly* equivalent to a one-place buffer process; this shows both safety (any output must have been input earlier) and liveness (every input is eventually output). Similarly we can prove the equivalence of a network based on a *sliding window protocol* with parameter n (as given in CSP by [18]) and an $(n + 2)$-place buffer; again we obtain both safety and liveness, unlike the proof given in [18], which focusses on safety.

References

[1] Blamey, S., *The soundness and completeness of axioms for CSP processes*, in: Topology and category theory in computer science, Reed, M. and Roscoe, A.W., and Wachter, R. (editors), Oxford University Press, 1991.

[2] Brookes, S., *Full abstraction for a shared-variable parallel language*, Proc. 8th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1993), 98–109. Journal version in: Information and Computation, vol 127, No. 2, Academic Press (June 1996).

[3] Brookes, S., *The essence of Parallel Algol*, Proc. 11th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1996) 164–173. Journal version to appear in *Information and Computation*, 1998.

[4] Brookes, S., *Idealized CSP: Combining Procedures with Communicating Processes*, MFPS'97, Pittsburgh, March 1997. Electronic Notes in Theoretical Computer Science 6, Elsevier Science (1997). URL: `http://www.elsevier.nl/locate/entcs/volume6.html`.

[5] Brookes, S., *On the Kahn Principle and Fair Networks*, MFPS'98, Queen Mary Westfield College, May 1998. Full version submitted to *Theoretical Computer Science*.

[6] Brookes, S. and Roscoe, A.W., *An improved failures model for CSP*, Seminar on concurrency, Springer-Verlag, LNCS 197, 1984.

[7] Brookes, S. and Roscoe, A.W., *Deadlock Analysis in networks of communicating processes*, Distributed Computing (1991) 4:209-230.

[8] Chaum, D., *The dining cryptographers problem: unconditional sender and recipient untraceability*, J. Cryptology (1), 1988.

[9] Clarke, E. M. and Emerson, E. A. and Sistla, A. P., *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems, 8 (2), 1986

[10] Formal Systems (Europe), Ltd., *Failures-Divergence Refinement: FDR2 Manual*, 1997.

[11] Hoare, C. A. R., *Communicating Sequential Processes*, Comm. ACM, 21(8):666–677 (1978).

[12] Kahn, G. and MacQueen, D. B., *Coroutines and Networks of Parallel Processes*, Information Processing '77, North Holland, 1977.

[13] Lowe, G., *Breaking and fixing the Needham-Schroeder public-key potocol using FDR*, 2^{nd} International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Springer-Verlag, LNCS vol. 1055, 1996.

[14] Marrero, W. and Clarke, E. M., and Jha, S., *Model Checking for Security Protocols*, Carnegie Mellon University, Technical report CMU-CS-97-139, May 1997.

[15] Meadows, C., *Analyzing the Needham-Schroeder public-key protocol: a comparison of two approaches*, European Symposium on Research in Computer Security, Springer-Verlag, 1996.

[16] Mitchell, J. and Mitchell, M. and Stern, U., *Automated Analysis of Cryptographic Protocols Using Murϕ*, IEEE Symposium on Security and Privacy, IEEE Computer Society Press, 1997.

[17] Needham, R. and Schroeder, M., *Using encryption for authentication in large networks of computers*, Comm. ACM, 21(12), pages 993-9, 1978.

[18] Paliwoda, K. and Sanders, J.W., *The sliding window protocol in CSP*, Oxford University Computing Laboratory, Programming Research Group, Technical Report PRG-66 (1988).

[19] Park, D., *On the semantics of fair parallelism*. In D. Bjørner, editor, **Abstract Software Specifications**, Springer-Verlag LNCS vol. 86 (1979), 504–526.

[20] Roscoe, A.W., *Model checking CSP*, in **A Classical Mind: Essays in honor of C. A. R. Hoare**, Prentice-Hall, 1994.

[21] Roscoe, A.W., *Modelling and verifying key-exchange protocols using CSP and FDR*, IEEE Workshop on Computer Security Foundations, IEEE Computer Society Press, 1995.

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

[23] Schneider, S., *Security properties and CSP*, IEEE Symposium on Security and Privacy, IEEE Computer Society Press, 1996.

[24] Schneider, S., *Verifying authentication protocols in CSP*, IEEE Symposium on Security and Privacy, IEEE Computer Society Press, 1997.

[25] Schneider, S. and Sidiropoulos, A., *CSP and Anonymity*, ESORICS, Springer-Verlag, LNCS vol. 1146, 1996.