Specifying Security for Computer Supported Collaborative Working

Simon N. Foley* Jeremy L. Jacob†

Abstract

CSCW systems provide computer support to facilitate cooperation between users. This paper proposes an approach to the formal specification of security requirements for CSCW applications, where a CSCW application is viewed as a collection of activities that users may participate in. The specification approach is straightforward, and can be used to capture a wide variety of security requirements, including integrity and confidentiality. It is illustrated with a case study of a secure electronic examinations system.

1 Introduction

The purpose of Computer Supported Collaborative Working (CSCW) is to provide computer support that facilitates collaboration and cooperation between users. This support may be as simple as the provision of electronic mail facilities, giving users the freedom to cooperate in an unstructured manner. It may be as sophisticated as some of the applications in concurrent engineering which exercise a high degree of control over the actions of the participants in an engineering design activity. An example is reported by Bowen and Bahler [1]; the system they describe mediates conflicts between user's contributions towards an overall design. CSCW support may be across a single computer environment, or more typically, across networks of heterogeneous systems.

Research in CSCW and groupware evolved from work on Decision Support Systems [9]. A great deal of work has been done on the technological aspects of CSCW: the problems of how one actually provides computer support for cooperation. Introductions to the subject may be found in the literature [17, 18].

In this paper we are interested in what is meant by security in CSCW applications. To do this, we require a clear and formal definition of what is meant by

*Department of Computer Science, University College, Cork, Ireland

†Department of Computer Science, The University of York, Heslington, York YO1 5DD, England

a CSCW application and the support that we would expect the CSCW system to provide. We are not concerned with, at least at this stage, the technological aspects of *how* security should be enforced. But we are concerned with how one might specify the security requirements for a CSCW application and the resulting properties that a CSCW system, providing support for the application, should uphold. While integrity security requirements can be characterized as safety or functionality properties, information flow based security requirements are described in terms of (harder to reason about) confidentiality properties [13]. Our proposed specification approach can be used to capture both types of requirement.

A simple case study motivates our proposal of what characterizes a CSCW system, and the security properties that it should maintain. The case study is a secure electronic examination system, where the users (professors, lecturers and students) cooperate in the setting, taking and grading of examinations. The case study contains many security requirements. For example, integrity requirements such as, *only lecturers appointed as setters of an exam paper may write an exam paper*, and confidentiality requirements such as, *a student may not learn or deduce anything about the contents of an exam paper until it has been released*. The security requirements for this case study are expressible, in a natural way, using our specification approach. To our surprise, all of the confidentiality requirements were expressible, using the same information-flow property. This is one of our main conclusions.

A CSCW application may be viewed as a collection of activities that users may participate in. Section 2 describes how users may join, leave and participate in, an activity during its lifetime. Our perspective of CSCW applications, while abstract, does appear to conform to the criteria given by Reinhard *et al.* [21] to identify CSCW systems. Based on this perspective, Section 3 investigates how we might formally specify functionality properties for CSCW activities. This gives rise to a number of safety properties that a CSCW system, supporting the activity, should uphold. Our approach is illustrated in Section 4 by the formal specification of the activities of a Secure Electronic Examinations (SEE) system.

Much of the SEE case study is an example of integrity and access-control style security specifications. These safety-property based requirements specify controls over whether, and how, users may participate in activities. Section 5 investigates how we might specify *confidentiality* properties for a CSCW system. The essence of our definition of confidentiality is that information may not flow from the participants in an activity to the non-participants, during the lifetime of the activity. Given this definition, we show how the confidentiality requirements for the SEE system can be expressed. We also provide an unwinding theorem which defines confidentiality in terms of states and state transitions.

In Section 6 we leave our main case study for a brief look at some others. A glossary of the mathematical notation used is given in Appendix B.

2 Computer Supported Collaborative Working

We view a collaboration as some agreed activity that a number of different users may participate in. A CSCW system is a system that provides support for these activities, ensuring that user participation in activities is in accordance to the requirements of the individual activities. In this section we propose an approach to organizing the description of the requirements of an activity.

Example 1 In a university, a number of people may be involved in the setting of an examination paper. Initially, the chair for the subject concerned is responsible for appointing staff to set the paper. Once appointed, a member of staff may set all or part of a paper; this typically involves devising a series of suitable questions for the exam paper. The chair also appoints checkers for examination papers. Checkers are members of staff who are responsible for checking the content/quality of an exam paper. An exam paper may be released to students only if it is deemed suitable by the checker(s). If it is considered unsuitable, then it must be re-set by the setter(s) and re-checked by the checker(s).

This cooperation required between chair, and staff on the development of an exam paper can be viewed as an activity. A CSCW system should provide support for this activity by ensuring that the requirements are upheld, for example, a requirement such as: only appointed staff may participate in the setting of a paper. $\square$

Let the set *USER* represent the set of all possible users. Let the set *EVENT* represent the set of all events that users may engage in, that is, the set of all possible interactions between the system and its environment (users).

We propose to characterize an individual CSCW activity by the following components.

- **join**, **leave** : $\mathbb{P}\,EVENT$. These define the sets of possible events that a user may engage to join and leave the activity, respectively. Note that $\mathbb{P}\,EVENT$ is the the powerset of *EVENT*.

- **do** : $\mathbb{P}\,EVENT$. Once a user is a member of the activity, she participates in the activity by engaging events from the set **do**.

- **obo** : $EVENT \leftrightarrow USER$ ('on behalf of'). Event e is engaged on behalf of user u for this activity if e **obo** u holds. If e **obo** u holds, the actual user may engage the event or the event may be engaged by a different user on behalf of user u. Note that, since **obo** is defined to be a relation, then it is possible for one user to engage an event on behalf of a number of other users.

- **start**, **finish** : $\mathbb{P}\,EVENT$. Users may participate in an activity only during the activity's lifetime. The activity must start with an event from **start**. An event from **finish** is necessary to terminate the activity.

- **conclude** : $\mathbb{P}(\mathrm{seq}_1 EVENT)$. Defines the sequences of activity events, that may lead to a successful conclusion of the activity. Note that $(\mathrm{seq}_1 EVENT)$ is the set of all possible non-empty sequences of events from $EVENT$.

This characterization forms the basis for the specification of functional requirements of activities. Section 3 will give a formal interpretation for this view of an activity.

Example 2 Consider a simplification of the set paper activity (Example 1), where there is just one exam paper to be set and the university has just one chair and one lecturer. A more realistic version of this activity will be considered in Section 4.

The chair appoints the lecturer as the setter of the exam paper by engaging event *appoint-setter*. When this event is engaged (by the chair) the lecturer becomes a member of the activity.

$$\mathbf{join} = \{\textit{appoint-setter}\}$$

We regard the chair as the initiator of the activity, rather than a participant of the activity. The lecturer may at any stage opt out of participation in the activity by resigning.

$$\mathbf{leave} = \{\textit{resign-setter}\}$$

Relation **obo** must be appropriately defined to reflect the fact that while the lecturer may engage event *resign-setter* on behalf of herself (*resign-setter* **obo** *lecturer*), the chair engages the event *appoint-setter* on behalf of the lecturer (*appoint-setter* **obo** *lecturer*). Once joined, the lecturer may set the paper using an event of the form *write.c*, where $c : TEXT$ gives the questions etc., of the paper. Revisions are made by writing further versions of the paper.

$$\mathbf{do} = \textit{write} : TEXT$$

Our notation $\textit{write} : TEXT$ denotes the set of all possible events of the form *write.c*, where $c \in TEXT$.

Participation in the activity may start once it is announced that the paper must be set ($\mathbf{start} = \{\textit{announce}\}$). The activity is finished when the paper (with its contents) is released to students.

$$\mathbf{finish} = \textit{release} : TEXT$$

If the paper has not yet been written then it is inappropriate to release the paper, and the activity should not be considered finished. The set **conclude** is used to define when the activity may successfully conclude, that is, the sequences of events that may lead to a successful termination. An exam paper may not be released if it has not yet been written. Thus the set paper activity may terminate with a *release.c* if it is preceded by a *write.c*.

$$\mathbf{conclude} = \{c : TEXT \bullet \langle write.c, release.c \rangle\}$$

The set notation defines the set of all sequences of the form $\langle write.c, release.c \rangle$, for $c \in TEXT$. Concluding sequences are drawn from events in **join**, **leave** and **do**. Implicit in our definition of an activity is a requirement that the lecturer participating in the activity (engaging a *write.c* event) is a member of the activity at that point. Thus, an activity having engaged in events $\langle appoint\text{-}setter, write.c, write.a, resign\text{-}setter, release.a \rangle$ is considered to have concluded since it contains the concluding sequence $\langle write.a, release.a \rangle$. Events not in **join**, **leave** and **do** are ignored: we have found that this leads to simpler **conclude** specifications. There are a number of similar requirements implicit with our characterization of CSCW activities; these will be considered in the next section. □

3 Specifying CSCW activities

In [10] a trace-based specification for a system is given as a predicate $p(t)$, which must hold for all possible traces t of the system (for the system to meet the specification). Such trace based specifications may be used to describe safety properties for a system.

In this paper we are concerned with the specification of a certain class of systems—those that are expected to manage a collection of CSCW activities. In Section 2 we described how activities can be viewed in terms of users joining, leaving, participating, etc. This perspective provides us with a basic idiom or template for specifying CSCW activities: it gives us a guideline on the way an activity can be organized for specification

Implicit in this approach to organizing and specifying an activity is a requirement that the components of the template are well-defined, that is, correctly specified. For example, the last event in a **conclude** trace should be an event from the set **finish**. Section 3.1 gives properties that an activity, described using our template, should possess to be correctly specified. Given a correctly specified activity, Section 3.2 defines (safety) properties that a system supporting the activity is expected to possess. To aid the reader, we typeset the names of these properties in a `typewriter` font. Names ending in `REQ` correspond to requirements for correct specification, while names ending in `SPEC` correspond to the safety properties that a CSCW system is expected to uphold.

3.1 Correct specification of activities

If an activity is defined using the template above (in terms of variables **join**,**leave**, and **do**, etc.) then for consistency, every user who can participate (**do**) in the activity must be capable of being a member of the activity. The users who can participate in the activity are those for whom events from **do** may be engaged on behalf of, that is, the set $\mathbf{obo}(\!|\mathbf{join}|\!)$. The notation $\mathbf{obo}(\!|\mathbf{join}|\!)$ is the relational image of **join** through **obo**. It is defined as the set of users $u \in USER$ to which **obo** relates some member of **join**. These users must be able to join and leave the activity. Thus we must have

$$\texttt{MemberDoREQ} \triangleq \mathbf{obo}(\!|\mathbf{do}|\!) \subseteq \mathbf{obo}(\!|\mathbf{join}|\!) \cap \mathbf{obo}(\!|\mathbf{leave}|\!)$$

We also require that the relation **obo** associates users for all events in **do**, i.e., $\mathbf{do} \subseteq \operatorname{dom} \mathbf{obo}$. Other events that may be engaged by, or on behalf of, participants are **join** and **leave**. However, we do not regard **start** and **finish** events as part of the participation and therefore, at this stage, are not concerned with who engages them. Thus we must have

$$\texttt{OnBehalfOfREQ} \triangleq (\mathbf{join} \cup \mathbf{leave} \cup \mathbf{do}) \subseteq \operatorname{dom} \mathbf{obo}$$

The variable **conclude** defines the sequence of activity events that may lead to a successful conclusion. Thus, a sequence from **conclude** must terminate with a finish event. Furthermore, we chose to define concluding sequences in terms of **start**, **finish** and **do** events only. We have found that adopting this approach leads to somewhat simpler specifications of **conclude**. Note that successful termination can still be dependent on membership of the activity—an example of this will be given in Section 4.3.

$$\begin{aligned}\texttt{ConclusionREQ} \quad \triangleq \quad & \mathbf{conclude} \neq \{\} \wedge \langle\rangle \notin \mathbf{conclude} \wedge \\ & last(\!|\mathbf{conclude}|\!) \subseteq \mathbf{finish} \wedge \\ & (\forall\, s : \mathbf{conclude} \bullet s \upharpoonright (\mathbf{start} \cup \mathbf{finish} \cup \mathbf{do}) = s)\end{aligned}$$

where, given $s : \operatorname{seq} EVENT$, and $A : \mathbb{P}\, EVENT$, then $s \upharpoonright A$ is trace s restricted to events from A, i.e., the trace s with all events not in A removed. Thus, the activity is correctly specified if `TemplateREQ` holds, where

$$\texttt{TemplateREQ} \triangleq \texttt{MemberDoREQ} \wedge \texttt{OnBehalfOfREQ} \wedge \texttt{ConclusionREQ}$$

3.2 Properties of activities

Given a correctly specified activity (based on our template and satisfying requirement `TemplateREQ`), we must specify the properties that a system supporting the activity is expected to possess. These properties are expressed as a predicate $p(t)$ over a trace variable t. For the system to maintain the property, $p(t)$ must be true for all possible traces t of the system. In the following discussion, the properties we specify will be in terms of trace t, with respect to a particular activity described in terms of variables **join**, **leave**, **start**, **finish**, **conclude** and **obo**.

Users may join and leave an activity any number of times. Given a system that has engaged a sequence of events denoted by trace t, then user u is currently a member of the activity if there is one more **join** event than **leave** event engaged on behalf of u (for this activity).

$$\texttt{Member}(u) \triangleq \#(t \restriction \{j : \textbf{join} \mid j \textbf{ obo } u\}) = \#(t \restriction \{l : \textbf{leave} \mid l \textbf{ obo } u\}) + 1$$

$$\texttt{NotMember}(u) \triangleq \#(t \restriction \{j : \textbf{join} \mid j \textbf{ obo } u\}) = \#(t \restriction \{l : \textbf{leave} \mid l \textbf{ obo } u\})$$

At any moment, every user must be either a member or not a member of the activity.

$$\texttt{MembershipSPEC} \triangleq \forall\, u : \mathit{USER} \bullet \texttt{Member}(u) \vee \texttt{NotMember}(u)$$

The activity is considered to have started once an event from **start** has been engaged. Note that the activity may be restarted at any time by the further engaging of start events. Given that $P[v/v']$ denotes the predicate P with all occurances of free variable v replaced by v', then define

$$\texttt{Started} \triangleq \mathit{last}(t \restriction (\textbf{start} \cup \textbf{finish})) \in \textbf{start}$$

The activity has successfully finished if a finish event has been engaged and the sequence of events that lead to the finish is a valid conclusion.

$$\texttt{Finished} \triangleq \mathit{last}\ t \restriction (\textbf{start} \cup \textbf{finish}) \in \textbf{finish} \wedge \exists\, c : \textbf{conclude} \bullet c \preceq (t \restriction \textbf{do} \cup \textbf{start} \cup \textbf{finish})$$

where $c \preceq s$ means that trace c ends trace s, i.e., there exists a sequence u such that $u \frown c = s$. Recall from requirement `ConclusionREQ` that concluding sequences are in terms of **start**, **finish** and **do** events only. If successful termination for a particular activity is dependent on activity membership, then since the specification of **conclude** is constructed in terms of free variable t, the current user membership may be obtained by reference to the predicate $\texttt{Member}(u)$. An example of this will be given in Section 4.3.

While an activity may be restarted at any time, once it has successfully finished, it may neither restart nor finish again. Furthermore, users may no longer participate in the activity. However, the user may join and/or leave the activity at *any* time.

$$\texttt{LifeSPEC} \triangleq (\mathit{last}\ t \in \textbf{start} \cup \textbf{finish} \cup \textbf{do}) \Rightarrow \neg\, \texttt{Finished}[t/(\mathit{front}\ t)]$$

A user participates in the activity by engaging events from **do**. A user may participate only if she is currently a member of the (started) activity.

$$\texttt{PartSPEC} \triangleq \forall\, u : \mathit{USER} \bullet \quad (\mathit{last}\ t \in \mathbf{do}) \wedge ((\mathit{last}\ t)\ \mathbf{obo}\ u) \Rightarrow \texttt{Member}(u) \wedge \texttt{Started}$$

Thus, the safety properties that must be upheld by a system providing support for an activity specified using the template described in Section 2 is that for all possible traces t of the system then `ActivitySPEC` holds, where

$$\texttt{ActivitySPEC} \triangleq \texttt{MembershipSPEC} \wedge \texttt{LifeSPEC} \wedge \texttt{PartSPEC}$$

3.3 A notation for activity specifications

An activity specification based on the proposed template gives rise to a property `ActivitySPEC` that the system must uphold. A syntactic form for presenting this requirement is given as

$$
\begin{array}{l}
\textbf{Activity}\ \langle \mathit{Name} \rangle [\langle \mathit{vars} \rangle]\ \textbf{is} \\
\quad \begin{array}{ll}
\textbf{start} & := \{\mathit{starting\ events}\} \\
\textbf{finish} & := \{\mathit{finishing\ events}\} \\
\textbf{join} & := \{\mathit{activity\ join\ events}\} \\
\textbf{leave} & := \{\mathit{activity\ leave\ events}\} \\
\textbf{do} & := \{\mathit{participation\ events}\} \\
\textbf{conclude} & := \{\mathit{successful\ concluding\ sequences}\} \\
\textbf{obo} & := \mathit{on\ behalf\ of\ relationship}
\end{array} \\
\textbf{end}\ \langle \mathit{Name} \rangle
\end{array}
$$

This syntax has the obvious interpretation: it defines a specification (on a free variable t) called $\langle \mathit{Name} \rangle$ which has property `ActivitySPEC` over variables **start**, **finish**, etc. The syntax is used to associate values with these variables for a particular activity. The optional entry $\langle \mathit{vars} \rangle$ is used to introduce any additional free variables used in the specification. The specification is well-defined if it satisfies `TemplateREQ`. Outside the syntactic structure, the variable names **start**, **finish**, etc., are considered out of scope, but may be referred to using the conventional 'dot' notation, for example $\langle \mathit{Name} \rangle$.**start**.

Example 3 Consider Example 2. The specification for the set paper activity is:

$$
\begin{array}{l}
\textbf{Activity}\ \texttt{SetPaper}\ \textbf{is} \\
\quad \textbf{start} \quad := \{announce\} \\
\quad \textbf{finish} \quad := \{release\} \\
\quad \textbf{join} \quad := \{appoint\text{-}setter\} \\
\quad \textbf{leave} \quad := \{resign\text{-}setter\} \\
\quad \textbf{do} \quad := write : TEXT \\
\quad \textbf{conclude} := \{\ c : TEXT \bullet \langle write.c, release.c \rangle\ \} \\
\quad \textbf{obo} \quad := EVENT \times \{lecturer\} \\
\textbf{end}\ \texttt{SetPaper}
\end{array}
$$

It is straightforward to prove that this specification is well-defined, in the sense that `TemplateREQ` holds. The relation **obo** is specified as a set of pairs, with the usual interpretation that $(x, y) \in \textbf{obo} \Leftrightarrow (x\ \textbf{obo}\ y)$. To keep the definition of **obo** simple, we specify that the lecturer engages all events on behalf of herself; this does not affect the system specification.

Suppose the professor appoints herself as a checker of the single exam paper. The process of checking an exam paper starts once something has been written. Thus, the *write* events are the start events for the check paper activity. The professor checks the paper and engages *ok* or *ko* to signify whether the paper is acceptable or not. The paper may be released only when the paper as been *ok*'ed. Thus we specify the activity as

$$
\begin{array}{l}
\textbf{Activity}\ \texttt{CheckPaper}\ \textbf{is} \\
\quad \textbf{start} \quad := \texttt{SetPaper}.\textbf{do} \\
\quad \textbf{finish} \quad := \texttt{SetPaper}.\textbf{finish} \\
\quad \textbf{join} \quad := \{appoint\text{-}checker\} \\
\quad \textbf{leave} \quad := \{resign\text{-}checker\} \\
\quad \textbf{do} \quad := \{ok, ko\} \\
\quad \textbf{conclude} := \{c : TEXT \bullet \langle ok, release.c \rangle\} \\
\quad \textbf{obo} \quad := EVENT \times \{professor\} \\
\textbf{end}\ \texttt{CheckPaper}
\end{array}
$$

This specification implies that if a paper that has been *ok*'ed is subsequently written to (by $write.c$), then it is considered re-started, and must be re-checked and *ok*'ed again before it may successfully conclude. □

4 SEE–Secure Electronic Examinations

In this section we extend the simple examination system described in Examples 1 to 3. This case study serves two purposes: firstly, to illustrate that our template is suitable for organizing the specification of CSCW activities, and secondly, it provides a useful collection of activities for which confidentiality requirements will be defined in the next section.

We assume that there exists a function $usr : EVENT \rightarrow USER$ where, given an event x, then $usr\ x$ defines the user who may engage that event. For convenience, every event that we will define will be of the form $u.command$, where u represents the user engaging the event. The function usr serves as useful default value for relation **obo**. This corresponding to the situation when users engage events on behalf of themselves.

4.1 Setting exam papers

The `SetPaper` activity between the chair and lecturer in Example 3 can be generalized to many exam papers and staff. In this case we assume that the cooperation is between the chair of the department and the lecturers. There will be a set paper activity for every exam paper. Thus the `SetPaper` activity specification is parameterized in variable $p : PAPER$, where $PAPER$ represents the set of all possible exam papers.

At various times the chair decides what papers should be set and by whom. Rather than engage a single *announce* event for every paper to be set, a chair (user u) engages an event of the form $u.announce.P$, where $P : \mathbb{P}\, PAPER$ defines the exam papers to be set. This is an example of a single event acting as the start event for a number of activities. Similarly, given

$$ASSNS = PAPER \leftrightarrow USER$$

the chair engages an event of the form $u.appoint\text{-}setter.S$, where $S \in ASSNS$, to appoint the setters for exam papers. The user $v : USER$ is appointed to set paper $p : PAPER$ when the event $u.appoint\text{-}setter.S$, is engaged by the chair u and $(v, p) \in S$. Event $u.appoint\text{-}setter.S$ is an example of an event that is engaged by the chair u but on behalf of many other users, in this case, joining the `SetPaper` activity on behalf of the setters. For simplicity we assume there is no cooperation between departments on the setting of exam papers. The activity is specified as

Activity `SetPaper`$(p : PAPER)$ **is**

$$
\begin{array}{ll}
\textbf{start} & := USER : announce : \{P : \mathbb{P}\, PAPER \mid p \in P\} \\
\textbf{finish} & := USER : release.p : TEXT \\
\textbf{join} & := USER : appoint\text{-}setter : ASSNS \\
\textbf{leave} & := USER : resign\text{-}setter.p \\
\textbf{do} & := USER : write.p : TEXT \\
\textbf{conclude} & := \{\ u, v : USER;\ c : TEXT \bullet \langle u.write.p.c, v.release.p.c \rangle\ \} \\
\textbf{obo} & := \begin{pmatrix} usr \oplus \{\ S : ASSNS;\ u, v : USER \mid (p, v) \in S \\ \qquad \bullet (u.appoint\text{-}setter.S \mapsto v)\ \} \end{pmatrix}
\end{array}
$$

end `SetPaper`

For the sake of clarity we have generalized the ':' operator described earlier, so that a term such as $USER : release.p : TEXT$ is the set of all events of the form

$u.release.p.c$, where $u : USER$ and $c \in TEXT$. Operator $\oplus$ is relation override; the relation $usr \oplus R$ relates every event in the domain of R to the same users as R does, and everything else in the domain of usr to the same users as usr does.

This specification permits *any* user $u \in USER$ to engage in start events of the form $u.announce.P$. Section 4.2 will consider how this may be restricted to just those users who are appointed chairs of departments.

The `SetPaper` activity is terminated by $v.release.p.c$, where c corresponds to the content of the exam paper. For a successful conclusion the system must ensure that the content of the paper released corresponds to its content when it was *last* written to. Just one appointed setter writing the paper is a sufficient conclusion condition.

4.2 Policing membership of activities

Events such as $u.announce.P$ and $u.appoint\text{-}setter.S$ are engaged by a chair u to announce and appoint the setters of exam papers. However, we must ensure that any user engaging such an event has actually been appointed a chair. One could achieve this by limiting the events that users can engage, but that would require advance knowledge of what users were chairs, what users where lecturers, etc. This is not realistic. Our approach is to view this responsibility of a chair as a simple activity with just one participant, the chair herself. There is potentially one of these activities for every user, however the activity is initialized only when the user gets appointed as a chair. Thus the specification is

$$
\begin{array}{l}
\textbf{Activity}\ \texttt{Chair-Duties}(prof : USER)\ \textbf{is} \\
\quad \begin{array}{ll}
\textbf{start} & := USER : appoint\text{-}chair.prof \\
\textbf{finish} & := \{prof.resign\text{-}chair\} \\
\textbf{join} & := \textbf{start} \\
\textbf{leave} & := \textbf{finish} \\
\textbf{do} & := \left(\begin{array}{l} prof.announce : (\mathbb{P}\ PAPER) \cup \\ prof.appoint\text{-}setter : ASSNS \cup \\ prof.appoint\text{-}checker : ASSNS \cup \\ prof.release : PAPER : TEXT \end{array}\right) \\
\textbf{conclude} & := \{\langle prof.resign\text{-}chair\rangle\} \\
\textbf{obo} & := usr \oplus \{u : USER \bullet (u.appoint\text{-}chair.prof \mapsto prof)\}
\end{array} \\
\textbf{end}\ \texttt{Chair-Duties}
\end{array}
$$

The new events in **do** will be introduced in later subsections. An appointment is made on behalf of the chair by some other user (for example the Registrar of the College), for which we assume there is an appropriate activity defined.

Activity specifications `SetPaper` and `Chair-Duties` are a good example of how integrity and access-control style requirements can be specified. Activity specification `SetPaper` defines a role for lecturers—that of setting papers. Lecturers may take on this role (participate in the activity) only if they are members of the activity. Only users who are appointed chairs have the right to grant membership of activity `SetPaper`(p) to lecturers.

4.3 Checking exam papers

The `CheckPaper` activity for a particular paper p is between a chair, setters and checkers. The activity is initiated when the exam paper gets written to by a setter. The chair appoints checkers by engaging an event of the form *appoint-checker.C* ($C : ASSNS$), which has an interpretation that is similar to *appoint-setter* above. The activity is terminated when the paper is released and the most recent version of the paper has been *ok*'ed by every one of the appointed checkers. It is specified as

$$
\begin{array}{l}
\textbf{Activity } \texttt{CheckPaper}(p : PAPER) \textbf{ is} \\
\quad \begin{array}{lcl}
\textbf{start} & := & USER : write.p : TEXT \\
\textbf{finish} & := & USER : release.p : TEXT \\
\textbf{join} & := & USER : appoint\text{-}checker : ASSNS \\
\textbf{leave} & := & USER : resign\text{-}checker.p \\
\textbf{do} & := & USER : \{ok.p, ko.p\} \\
\textbf{conclude} & := & \left(\begin{array}{l} \{\ s : \mathrm{seq}_1(\textbf{do});\ u : USER;\ c : TEXT \mid \\ \qquad (\forall\, v : USER \bullet \texttt{Member}(v) \Rightarrow \\ \qquad\qquad s \upharpoonright (\textbf{obo}^{\sim}(\!|\{v\}|\!)) \neq \langle\rangle) \\ \quad \bullet\ s \frown \langle u.release.p.c\rangle\ \} \end{array} \right) \\
\textbf{obo} & := & \left(\begin{array}{l} usr \oplus \{\ C : ASSNS;\ u, v : USER \mid (p, u) \in C \\ \qquad\qquad \bullet\ (v.appoint\text{-}checker.C \mapsto u)\ \} \end{array} \right)
\end{array} \\
\textbf{end } \texttt{CheckPaper}
\end{array}
$$

Note that $\textbf{obo}^{\sim}$ denotes the *inverse* of the relation **obo**. Note how, in the definition of **conclude**, we use the predicate `Member` to identify what lecturers are currently members of the activity (given the free variable t).

4.4 Taking exam papers

An examination for a particular paper is started by a *release* event and terminates when the exam is *close*'ed. An exam is closed with a value $A : ANSWRS$, where

$$ANSWRS = USER \twoheadrightarrow TEXT$$

A gives the final answers to the paper, by the students participating in the exam. To take the exam, students must join the activity by engaging (*register*). This may be done before the exam starts or during it. The student may cancel registration (**leave**) at any time (including, during the actual examination) by engaging a *deregister* event. In Section 5 we consider how to specify the requirement that no collusion should occur between students during this activity.

Activity `TakePaper`$(p : PAPER)$ **is**

$$
\begin{array}{ll}
\textbf{start} & := \texttt{SetPaper}.\textbf{finish} \\
\textbf{finish} & := USER : close.p : ANSWRS \\
\textbf{join} & := USER : register.p \\
\textbf{leave} & := USER : deregister.p \\
\textbf{do} & := USER : answer.p : TEXT \\
\textbf{conclude} & := \left(\begin{array}{l} \{\ u : USER;\ s : \mathrm{seq}(\textbf{do});\ A : ANSWRS \mid \\ \qquad (\mathrm{dom}\, A) \subseteq \{\ st : USER \mid \texttt{Member}(st)\ \} \wedge \\ \qquad (\forall\, st : USER;\ a : TEXT \bullet \texttt{Member}(st) \Rightarrow \\ \qquad\qquad last(s \upharpoonright \textbf{do}) = st.answer.p.a \Leftrightarrow (st \mapsto a) \in A) \\ \qquad \bullet\ s \frown \langle u.close.p.A \rangle\ \} \end{array} \right) \\
\textbf{obo} & := usr
\end{array}
$$

end `TakePaper`

The `TakePaper` activity is successfully concluded when it is *close*'ed with answers for all students who were registered (and did not cancel), and those answers correspond to the final versions of answers given by the students.

Following the discussion in Section 4.2, activity `TakePaper` does not control whether a student should be permitted to register for an examination: any student may register. `TakePaper` deals only with the taking of the exam paper. A further activity should be specified which would ensure that students may register for an exam only if they have studied that topic. In turn, a student should be allowed study a topic only if the student has registered in the University. We envisage many more of these 'control' activities which are used to control, membership in and initiation of, other activities.

4.5 Grading exam papers

Once the examination is finished, the answers may be graded. For simplicity, we have defined the graders to be the setters of the paper. A grader u assigns a grade $g : \mathbb{N}$ to the answers of each student $st : USER$ by engaging $(u.grade.p.st.g)$. When grading is completed the results may be published (and the activity terminated). Grades are published in the form $G : RSLTS$, where

$$RSLTS = USER \rightarrowtail \mathbb{N}$$

G gives the grade for any individual student who took the paper.

Activity `GradePaper`$(p : PAPER)$ **is**

$$
\begin{array}{ll}
\textbf{start} & := USER : close.p : ANSWRS \\
\textbf{finish} & := USER : publish.p : RSLTS \\
\textbf{join} & := \texttt{SetPaper}.\textbf{join} \\
\textbf{leave} & := \texttt{SetPaper}.\textbf{leave} \\
\textbf{do} & := USER : grade.p : USER : \mathbb{N} \\
\textbf{conclude} := & \left(\begin{array}{l}
\{\ u, v : USER;\ s : \mathrm{seq}(\textbf{do});\ A : Answers;\ R : RSLTS \mid \\
\qquad (\mathrm{dom}\ \ A) = (\mathrm{dom}\ \ R) \wedge \\
\qquad (\forall\, w : \mathrm{dom}\ \ R \bullet \\
\qquad\qquad last(s \upharpoonright \{\ v : USER;\ g : \mathbb{N} \bullet v.grade.p.w.g\ \}) \\
\qquad\qquad\qquad \in \{\ v : USER \bullet v.grade.p.w.(R\ \ w)\ \}) \\
\qquad \bullet \langle u.close.p.A \rangle \frown s \frown \langle v.publish.p.R \rangle\ \}
\end{array} \right) \\
\textbf{obo} & := \texttt{SetPaper}.\textbf{obo}
\end{array}
$$

end `GradePaper`

The `GradePaper` activity is successfully concluded when the results are published for *all* those students who sat the paper. This is an example of successful conclusion depending on the starting event of the activity.

The reader should note that we have not considered who should be responsible for *close*'ing an exam and *publish*'ing results. A simple solution would be to modify activity `Chair-Duties`, making the chair responsible for these actions. However, in reality, university examinations are highly bureaucratic, requiring participation in a variety of activities by many different users. For reasons of space we cannot consider it here.

5 Confidentiality Properties

5.1 The problem

We now investigate how we might specify *confidentiality* properties for a CSCW system.

In the Secure Electronic Examinations example there are some obvious confidentiality requirements:

- No examinee should learn any details of the contents of the examination before the start of the examination.
- No examinee should learn any details of the contents of any other examinees answer paper *between* the start of the paper and the end of the examination.
- No examinee should learn any details of the marking until results are posted.

These requirements are not appropriate for every institution. For example, some have a special paper where the list of essay topics is announced beforehand, while the essay titles themselves are concealed. Some announce details of a candidate's marks to that candidate alone in advance of a general release of overall marks. Group examinations are held in some institutions, but it is generally required that no group plaigerise from any other group.

All of these properties (including the rarer ones immediately above) can be constructed out of a statement of the form:

> *No information flows from source to sink between start and finish of activity.*

As in the earlier sections we now construct a specification template (giving both a concrete syntax and its semantics) to write down such confidentiality properties. This template is based on (a version) of non-interference expressed in the semantic space associated with Hoare's CSP [10]. Non-interference and variants have been well studied [3, 4, 7, 8, 11, 12, 14, 15, 16, 20, 22] since its introduction by Goguen and Meseguer [5, 6].

5.2 A simple confidentiality template

Each activity has a given ending point marked by the occurrence of one member of a given set of events: **finish**. This set is exactly as in the discussion of functionality properties above. We can define a predicate over traces of events which describes when an activity has finished.

$$\mathit{finished}\ t \Leftrightarrow \#(t \upharpoonright \mathbf{finish}) > 0$$

This equation says that (an instance of) an activity has finished at the end of trace t if an event which marks its finish has occurred. Note that '*finished* t' is weaker than the predicate '`Finished`' of Section 3.2.

Similarly we define a predicate over traces to say if, in the behaviour defined by the trace, the events in the **do** set are visible to the non-**do**. Let "$t \downarrow s$" be "the trace t, but with all events in the set s removed" and T be the set of all traces of the system; then:

$$\mathit{visible}\ t \Leftrightarrow t \downarrow \mathbf{do} \notin T$$

This is a variant of the classic non-interference definition.

Now we give a predicate that expresses that no information flows from an activity to the events outside the activity:

$$\mathit{Conf}\ t \Leftrightarrow (\mathit{visible}\ t \Rightarrow \mathit{finished}\ t)$$

Intuitively, if a trace t occurs, before the activity finishes, there must be another trace, equivalent from the external viewpoint, in which the activity has not started. After the activity has finished, any information about it may flow through the system.

A couple of points about this definition need elucidating. First, it is not necessary to refer to the start of an activity: no relevant information has been entrusted to the system before the start; indeed the **do** events will be forbidden to occur.

Second, note that this definition makes the events of the activity which are not **do** events —**join** and **leave** events— part of the "other" world, and so not able to be effected by the **do** events, nor able to be hidden from the events outside the activity. The latter is not a restriction; we give an example below to show how to hide the **join** events of an activity (Example 7). The former *is* a restriction; but not a serious one in practice: it prevents schemes where who joins and leaves an activity is dependent on the interactions of the activity so far, rather than the whim of a manager. (The **start** and **finish** events are ignored above because there can be no **do** events before a **start** event, and the presence of a **finish** event means that no restrictions are applied.)

To specify such a strong level of protection we write the keyword **Protected** in an activity specification. Then, for every trace $t \in T$, we require that $Conf\ t$ holds, with **do** instantiated as the **do** of the activity.

Example 4 It is enough to write

> **Protected activity** `GradePaper(p:PAPER)` **is**
> *...as before...*
> **end** `GradePaper`

to specify that no action of the activity $GradePaper(p)$ (for a given paper p) may interfere with any events outside the activity, until a *publish* event.

Note that this does *not* say that an examination grader cannot use *another* activity of the system, by suitable coding, to pass information about an examination in which she is involved. But this is just the same as not being able to prevent a grader colluding directly with an examinee outside the scope of the system. □

Example 4 is rather strong. If we took its analogue for *SetPaper*, for example, it would prevent any interference with *CheckPaper*, an unwanted outcome. For the purposes of confidentiality specification, we can combine just the necessary fields of existing activities into a new one.

Example 5 The protected activity *ProducePaper* partially combines activities *SetPaper* and *CheckPaper*. It does not give all of the functionality specification in those two descriptions.

> **Protected activity** `ProducePaper(p:Paper)` **is**
> **finish** $:= SetPaper(p).\mathbf{finish} \cap CheckPaper(p).\mathbf{finish}$
> **do** $:= SetPaper(p).\mathbf{do} \cup CheckPaper(p).\mathbf{do}$
> **end** `ProducePaper`

The binding **finish** $:=$ *SetPaper*(p).**finish** $\cap$ *CheckPaper*(p).**finish** contains redundant information, but we prefer not to break the symmetry of the description. $\square$

The confidentiality properties of *TakePaper* are most easily described on a per examinee basis. *TakePaper* above is not described this way, and so we need to extract from a set the events that belong to a given user. For a set of events S and a user $u \in USER$ we define $S \gg u$ to be the projection of S on to the events of user u.

Example 6 We can now describe the confidentiality properties for taking a paper as follows.

Protected activity `SitPaper(p:PAPER;u:USER)` **is**
  finish $:= (\mathit{TakePaper}(p).\mathbf{finish}) \gg u$
  do $:= (\mathit{TakePaper}(p).\mathbf{do}) \gg u$
end `SitPaper`

$\square$

Now we show how to code the secrecy of joining and leaving the team of examiners.

Example 7 Suppose we wish to protect the identity of the examiners who set the paper. We can do this by creating an activity whose **do** events are the **join** and **leave** events of *SetPaper*.

Protected activity `JoinExaminers(p:PAPER)` **is**
  finish $:= \mathit{SetPaper}(p).\mathbf{finish}$
  do $:= \mathit{SetPaper}(p).\mathbf{join} \cap \mathit{SetPaper}(p).\mathbf{leave} \cap \mathit{CheckPaper}$
end `JoinExaminers`

$\square$

5.3 An unwinding theorem

In this section we will identify a state of a system with any representative of the equivalence class of traces that leads to it. Two states are called equivalent if the portions accessible to the non-**do** events are equal; we will write this $s \equiv t$, for (representative) traces s and t.

We can now state two conditions under which an unwinding theorem holds:

- Equivalent states give rise to equivalent states:

$$s \equiv t \Rightarrow (s \frown \langle e \rangle \equiv t \frown \langle e \rangle \vee \{s \frown \langle e \rangle, t \frown \langle e \rangle\} \cap T = \{\})$$

  Note that we need to check if an event is possible in a state; if it is not then it must be impossible in both states (impossible events may be thought of as null transitions on both states).

- When an activity is not finished the result of a **do** event is equivalent to the immediately preceding state:

$$\forall\, d : \mathbf{do};\ t : T \bullet t \frown \langle d \rangle \in T \wedge \neg \mathit{finished}(t \frown \langle d \rangle) \Rightarrow t \frown \langle d \rangle \equiv t$$

The proof of this theorem is given in Appendix A.

5.4 Discussion

The major confidentiality properties of the SEE example are very simply specified. This is perhaps surprising. It comes about partly because so much has been captured by the (simpler to reason about and implement) functionality specification, but also because we have been able to modularise the description of the system into activities which are naturally non-interfering. Whether this will hold for domains with less drastic security requirements remains to be seen.

6 Further examples

Example 8 A scientific journal controls its quality by having its contributions reviewed by the authors' peers. To protect the reviewers their identity is kept anonymous forever. We can specify this as follows:

$$\begin{array}{l}
\textbf{Protected activity } \mathit{AnonReview}(\mathrm{p} : \mathit{ARTICLE}) \textbf{ is} \\
\quad \begin{array}{ll}
\textbf{finish} & := \{\} \\
\textbf{conclude} & := \{\} \\
\textbf{start} & := \{a : \mathit{ACADEMIC} \bullet a.\mathit{submit}.p\} \\
\textbf{do} & := \{a : \mathit{ACADEMIC} \bullet a.\mathit{appointed_reviewer}.p\} \\
 & \quad \cup \{a : \mathit{ACADEMIC} \bullet a.\mathit{accept}.p\} \\
 & \quad \cup \{a : \mathit{ACADEMIC} \bullet a.\mathit{reject}.p\} \\
\textbf{join} & := \{a : \mathit{ACADEMIC} \bullet \mathit{meet_editor}.a\}
\end{array} \\
\textbf{end } \mathit{AnonReview}
\end{array}$$

We deliberately leave the other fields empty for them to take their default values. (And we have kept the possible **do** actions smaller than it is in reality.)

Note how the **finish** and **conclude** fields are set to the empty set. This means that the activity can never finish, and the reviewers anonymity is protected for ever. This is unlike activity *JoinExaminers* (Example 7), where anonymity of examiners is maintained only during the setting of exam papers. □

Example 9 We can code granting and removing of access rights to a resource as follows:

Protected activity *AccessControl*(r : *RESOURCE*) **is**
  start $:= \{commission.r\}$
  finish $:= \{decommission.r\}$
  join $:= \{u : USER \bullet grant\_access.u.r\}$
  leave $:= \{u : USER \bullet remove\_access.u.r\}$
  do $:= \{u : USER;\ p : PARAMETER \bullet access.u.r.p\}$
end *AccessControl*

This is a coarse grained control of access. Any kind of access (coded by the different values of *PARAMETER*) is allowed. We could split a resource into, say, a read activity and a write activity. For example, the write activity could be modelled:

Protected activity *WriteAccessControl*(r : *RESOURCE*) **is**
  start $:= \{commission.r\}$
  finish $:= \{decommission.r\}$
  join $:= \{u : USER \bullet grant\_access.u.r\}$
  leave $:= \{u : USER \bullet remove\_access.u.r\}$
  do $:= \{u : USER;\ v : Value \bullet write.u.r.v\}$
end *WriteAccessControl*

Note that if these activities were unprotected then it would still be illegal for users without read access (say) to engage in a read event. However, this is not enough to ensure the confidentiality of the datum in the resource: events of other activities could leak information about its value. Thus, we may view such unprotected access control activities as capable of supporting Bell-LaPadula style MAC requirements, while their protected versions capture non-interference style flow security requirements. $\square$

7 Conlusions

We have presented a notation for describing and specifying CSCW systems where confidentiality is an issue. We believe that —apart from some rough edges noted below— this notation is suitable for use by people who are not trained in formal methods. We believe this for the following reason: *The notion of confidentiality used is* very *simple.*

Despite its simplicity it works well for the case study presented here. A danger is that in other applications than SEE it is too simple. Clearly we must try further case studies before definitely concluding that we have a sophisticated enough notion.

We are reasonably happy with the "outer" form of the language. The concepts to which one must give values are (for the most part) easy to comprehend.

But the notation in which the values are described is still "over-mathematical" for the audience we have in mind. These rough edges must be smoothed over before we can claim to have been more than minimally successful.

Acknowledgements

This paper is extended from an earlier one presented at the 8th IEEE Computer Security Foundations Workshop in Kenmare, Ireland [2]. We are grateful to the comments we had from the reviewers and participants at the workshop. We are also grateful for reviewers comments on this version of the paper.

This work was partly funded by a joint grant from Forbairt (Ireland) and the British Council.

References

[1] J.A. Bowen and D. Bahler. Constraint-based software for concurrent engineering. *IEEE Computer*, 26(1):66–68, January 1993.

[2] S.N. Foley and Jacob J.L. Specifying security for CSCW systems. In *Proceedings Computer Security Foundations Workshop*, pages 136–145, Kenmare, Co. Kerry, Ireland, June 1995. IEEE Computer Society.

[3] Janice Glasgow, Glen MacEwen, and Prakash Panangaden. Reasoning about knowledge and permission in secure distributed systems. In Jonathan K. Millen, editor, *Proceedings of the Computer Security Foundations Workshop*, pages 139–146, Bedford, MA, U. S. A., June 1988. The MITRE Corporation.

[4] Janice Glasgow, Glen MacEwen, and Prakash Panangaden. A logic for reasoning about security. In J. Thomas Haigh, editor, *Proceedings of the Computer Security Foundations Workshop III*, pages 2–13. Computer Society Press of the IEEE, June 1990.

[5] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings 1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

[6] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings 1984 IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society, 1984.

[7] John Graham-Cumming and J. W. Sanders. On the refinement of non-interference. In Ravi Sandhu, editor, *Proceedings of the Computer Security Foundations Workshop IV*, pages 35–54, Franconia, NH, U. S. A., 18–20 June 1991. Computer Society Press of the IEEE.

[8] J. W. Gray III. Toward a mathematical foundation for computer security. In *Proceedings 1991 IEEE Symposium on Research in Security and Privacy*, pages 21–34, May 1991.

[9] J. Grudin. Computer-supported cooperative work: History and focus. *IEEE Computer*, 27(5):19–26, May 1994.

[10] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice Hall International, Hemel Hempstead, 1985.

[11] Jeremy Jacob. Categorising non-interference. In *Proceedings Computer Security Foundations Workshop III*, pages 44–50. IEEE Computer Society Press, 1990.

[12] Jeremy Jacob. A uniform presentation of confidentiality properties. *IEEE Transactions on Software Engineering*, 17(11):1186–1194, November 1991.

[13] Jeremy Jacob. Basic theorems about security. *Journal of Computer Security*, 1(4):385–411, 1992.

[14] D. Johnson and J. Thayer. Security and the composition of machines. In *Proceedings Computer Security Foundations Workshop*, pages 98–111. The MITRE Corporation, 1988.

[15] D. McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings 1987 IEEE Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society Press, 1987.

[16] D. McCullough. Noninterference and the composability of security properties. In *Proceedings 1988 IEEE Symposium on Security and Privacy*, pages 177–186. IEEE Computer Society Press, 1988.

[17] J.D. Palmer and N.A. Fields. Special issue on computer supported cooperative work. *IEEE Computer*, 27(5), May 1994.

[18] Duska Rosenberg and Chris Hutchinson, editors. *Design Issues in CSCW.* Computer Supported Cooperative Work. Springer UK, 1994.

[19] J. M. Spivey. *The Z Notation: A Reference Manual.* Series in Computer Science. Prentice Hall International, Hemel Hempstead, UK, second edition, 1992.

[20] D. Sutherland. A model of information. In *Proceedings 9th National Computer Security Conference.* U. S. National Computer Security Center and U. S. National Bureau of Standards., 1986.

[21] Reinhard W. et al. CSCW tools: Concepts and architectures. *IEEE Computer*, 27(5):28–36, May 1994.

[22] J. T. Wittbold and D. Johnson. Information flow in nondeterministic systems. In *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 144–161. IEEE Computer Society Press, May 1990.

A Proof of the unwinding theorem

A.1 Proof strategy

We can prove the unwinding theorem of Section 5.3 by appealing to a general result on unwinding, expressed in category-theoretic terms, by Jacob [11].

Jacob's Theorem 3 and its precursors are expressed very abstractly, and in a somewhat unusual notation. To make use of the theorem each of the abstract concepts needs to be instantiated, and then the formulæ simplified.

A.2 Instantiations

The terms that we need to instantiate are:

A category, S This represents the states of the system. We instantiate this as the possible *traces* of the system. A trace represents the state of the system after its execution. Of course, several traces may lead to the same state, so we must take care to refer to traces as only as members of an equivalence class.

A subcategory, C **of** $[S \to S]$ This represents all possible programs over the states.

The generators, G **of** C This represents the individual steps, or events that can be composed together to make a member of C.

A collection of functors, q_a for each object a of S. This represents a state-dependent purge function. In our case it is specialised as a collection of functions, indexed by trace. If two traces lead to the same state, then their purges must lead to identical states.

The state must hold a record of whether a started activity is finished or not. The purge function depends on this record. If the activity is finished, then the purge function is the identity. If the activity is not finished, then the purge function removes all **do** events. For a trace, t, (representing the state it leads to) and an event, e:

$$purge_t\ e = \begin{cases} t \frown \langle e \rangle & \text{if } \mathit{finished}(t \frown \langle e \rangle) \\ t \restriction \mathbf{do} & \text{if } \neg \mathit{finished}(t \frown \langle e \rangle) \wedge e \in \mathbf{do} \\ t \restriction \mathbf{do} \frown \langle e \rangle & \text{otherwise} \end{cases}$$

That is, the purged output of e in the state reached by t is the same as the output in the state reached by:

- $t \frown \langle e \rangle$ if the activity is finished;
- $t \upharpoonright \mathbf{do}$ if the activity is ongoing and e is a **do** event; and
- $t \upharpoonright \mathbf{do} \frown \langle e \rangle$ if the activity is ongoing and e is not a **do** event.

A "low users' view arrow" w This represents the projection of the information in a state on to the low users' view. As we only need it in contexts such as: $a;\ w = b;\ w$ we will write $a \equiv b$ when the states (reached by the traces) a and b are identical to the low user.

A further point to note is that the category-theoretic treatment demands some sort of input-totality, a concept which the CSP notion of an event being impossible in a state seems to contradict. However this is not the case: we can treat impossibility of an event as a special kind of null operation on the state, and as it is so we do not even bother to keep it in the trace. This is why it is important, below , to insert a clause forcing events to be possible in both states or neither.

And last, two minor pieces of notation: by "$s \rhd f$" is meant the application of function f to value s: "$f(s)$"; by "$f;\ g$" is meant the functional composition of g and f: "$g \circ f$".

A.3 Simplification

The first of Jacob's criteria, VS, states that we have to check that low-equivalent states lead to low-equivalent states, for each generator:

$$\forall\, a, b : \mathit{Obj(S)}, g : G \bullet a \rhd w = b \rhd w \Rightarrow a \rhd (g;\ w) = b \rhd (g;\ w)$$

This translates as:

$$s \equiv t \Rightarrow (s \frown \langle e \rangle \equiv t \frown \langle e \rangle \vee \{s \frown \langle e \rangle, t \frown \langle e \rangle\} \cap T = \{\})$$

Note the clause that legislates an event must be impossible in both states if it is impossible in one.

The second of Jacob's criteria, VC, says that the high actions have no affect on the low actions:

$$\forall\, a : \mathit{Obj(S)};\ g : G \bullet (g \rhd q_a) = g;\ w$$

This translates as:

$$\forall\, t : T;\ g : \mathit{Event} \bullet \mathit{purge}_t\ g \equiv g$$

Simplification of this gives:

$$\forall\, d : \mathbf{do};\ t : T \bullet t \frown \langle d \rangle \in T \wedge \neg \mathit{finished}(t \frown \langle d \rangle) \Rightarrow t \frown \langle d \rangle \equiv t$$

B Notation

The mathematical notation used in this paper is based on Z [19]. We avoid using many operators that peculiar to Z; much of the notation can be easily recognized from its set and logic origins.

A set may be defined in Z using set specification in comprehension. This is of the form $\{ D \mid P \bullet E \}$, where D gives declarations, P a predicate and E an expression. The components of $\{ D \mid P \bullet E \}$ are the values taken by expression E when the variables introduced by D take all possible values that make the predicate P true. For example, the set of squares of all even natural numbers is defined as $\{ n : \mathbb{N} \mid (n \bmod 2) = 0 \bullet n^2 \}$. When there is only one variable in the declaration and the expression consists of just that variable, then the expression may be dropped if desired. For example, the set of all even numbers may be written as $\{ n : \mathbb{N} \mid (n \bmod 2) = 0 \}$. An equivalent way to define this set is as $\{ n : \mathbb{N} \bullet (2 * n) \}$—the predicate may be dropped if it equals true. Sets may also be defined in display form such as $\{1, 2\}$.

In Z, relations and functions are represented as sets of pairs. A (binary) relation R, declared as having type $A \leftrightarrow B$, is a component of $\mathbb{P}(A \times B)$. For $a \in A$ and $b \in B$, then the pair (a, b) is written as $a \mapsto b$, and $a \mapsto b \in R$ means that a is related to b under relation R. Functions are treated as special forms of relations.

$\mathbb{P}\, A$	The power set of A
$A \leftrightarrow B$	The set of all relations between A and B
$A \rightarrowtail\!\!\!\!\!\!+ B$	The set of all partial functions from A to B
$a \mapsto b$	The ordered pair (a, b)
$R(\!\mid A \mid\!)$	The image of the set A through the relation R. If R has range type B, then $R(\!\mid A \mid\!) = \{ b : B \mid (\exists\, a : A \bullet a \mapsto b \in R) \}$
$R^{\sim}$	The inverse of relation R.
$R \oplus G$	The relational override of R by G
$\mathrm{seq}\ A$	The set of all finite sequences over A
$\mathrm{seq}_1\ A$	The set of all non-empty finite sequences over A
$\mathit{last}\ t$	The last component of non-empty sequence t
$\mathit{front}\ t$	The sequence t with its last component removed
$\#\ t$	The length of sequence t
$t \upharpoonright A$	The restriction of sequence t to components in A
$t \downarrow A$	The sequence t but with components in A removed
$t \frown s$	The catenation of sequences t and s