Abstraction

One view of systems is said to be more abstract than another if it hides more detail, thereby identifying more processes.

Thus the view of a process in the traces model $\mathcal{T}$ is more abstract than that in the stable failures model $\mathcal{F}$.

Abstraction is a powerful tool since it allows us to capture the view of a system that is appropriate for a chosen purpose.

Abstraction in CSP

Apart from choice of model, there are three main ways of introducing abstraction in CSP:

- How much detail is recorded in the original program? What consititutes an event (it may be reasonable for one event in one view to be many in another, for example a transaction and its parts)? Are all decision mechanisms modelled faithfully or are they replaced by nondeterministic choice?
- Many-to-one renaming can be used to obscure details – for example the data components of channels – thereby allowing you to move easily between levels of abstraction. See, especially, examples in Chapter 13 (deadlock).
- If a process has alphabet $A \cup B$, we might be interested in how it looks to a user only able to see A.

We will concentrate on the last of these.

Assumptions

When discussing abstraction (i.e., following the whole of Chapter 12) we will assume that the process P we are trying to abstract is

- divergence free,
- $\checkmark$-free (i.e., never terminates via $SKIP$) *so that we don't have to worry about who can see* $\checkmark$, and
- finitely nondeterministic.

Note that this implies that the models $\mathcal{F}$, $\mathcal{N}$ and $\mathcal{U}$ all handle it equally well, giving it equivalent (and inter-translatable) values.

We will also assume that it has two users, Hugh and Lois, who respectively communicate with it in H and L (which partition its alphabet).

Eager abstraction

Given a process P and set of events L, it is tempting to believe that the abstracted view of P in L is given by $P \setminus (\Sigma \smallsetminus L)$.

In the traces model $\mathcal{T}$ this is the right answer, but over the richer models the situation is more complex since this definition does not usually deal with refusal sets and divergences appropriately.

$P \setminus X$ assumes that X has passed out of sight and control of the entire environment and hence that events from X become τ actions and thus 'eager'. Recall

$$failures(P \setminus X) = \{(s \setminus X, Y) \mid (s, Y \underline{\cup X}) \in failures(P)\}$$

We thus characterise hiding as *eager* abstraction:

$$\mathcal{E}_H(P) = P \setminus H$$

though, outside $\mathcal{T}$, this is of rather limited use as a means of abstraction. It only gives Lois' true view of P on the assumption that Hugh *always* offers P the whole of H.

Lazy abstraction

The conventional CSP view of Hugh is that he may accept or refuse any event.

Thus Hugh, from Lois' perspective, gets to introduce nondeterminism into the system both by the choice of *which* action to communicate and also by *whether* to communicate at all.

If Hugh decides to refuse all communications offered to him, P will appear to act like $P \underset{H}{\|} STOP$.

This suggests that the right way to form the abstraction is

$$(P \underset{H}{\|} Chaos_H) \setminus H$$

However, in $\mathcal{N}$ and $\mathcal{U}$ this turns out to be a little too drastic because it diverges ($\bot_{\mathcal{N}}$ or $\bot_{\mathcal{U}}$) whenever Hugh can communicate infinitely with P.

It is better to believe either than Hugh is sufficiently lazy not to hog the whole of P, or that P doesn't let him. Thus the abstraction is *always* divergence free (as P is).

There are three rather different ways of describing this *lazy* abstraction: one each in $\mathcal{N}$, $\mathcal{F}$ and $\mathcal{U}$.

Lazy abstraction over $\mathcal{N}$

Define a projection operator $P@L$ as a modified hiding: it is what Lois sees of all P's potential behaviours:

$$\{(s \setminus H, X) \mid (s, X \cap L) \in \mathit{failures}_\perp(P)\}$$

$\mathcal{L}_H^{\mathcal{N}}(P)$ is defined to be $P@L$.

For example, if $L = \{l1, l2\}$ and $H = \{h\}$, we might define P by

$$\begin{array}{rcl} P & = & l1 \to P \\ & & \Box\ l2 \to h \to P \\ & & \Box\ h \to P \end{array}$$

$\mathcal{L}_H^{\mathcal{N}}(P)$ is then equivalent to Q where

$$\begin{array}{rcl} Q & = & l1 \to Q \\ & & \Box\ l2 \to (STOP \sqcap Q) \end{array}$$

Lazy abstraction over $\mathcal{U}$

We can express this in terms of CSP operators.

Recall that $Fair(A, B)$ arbitrates fairly between the disjoint sets A and B (always allowing the whole of one set, but never allowing infinitely many of one without the other). Then

$$SemiFair(A, B) = RUN_A \mid\mid\mid Fair(A, B)$$

is a process which never refuses events in A, allows any finite combination of A's and B's, but only allows infinitely many B events to occur if infinitely many A's do as well.

We can then define

$$\mathcal{L}^{\mathcal{U}}_H(P) = (P \underset{\Sigma}{\parallel} SemiFair(L, H)) \setminus H$$

The hiding cannot introduce divergence because of the structure of $SemiFair(L, H)$, and this process has the same failures as $\mathcal{L}^{\mathcal{N}}_H(P)$.

Lazy abstraction over $\mathcal{U}$ – II

It is not quite equivalent to $\mathcal{L}_H^{\mathcal{N}}(P)$ because of examples like $Q1$ and $(Q2;\ STOP)$, where

$$\begin{array}{rcl} Q1 & = & l1 \to Q1 \\ & & \sqcap\ h \to Q1 \\ \\ Q2 & = & h \to (Q2;\ l1 \to SKIP) \\ & & \sqcap\ SKIP \end{array}$$

Here $\mathcal{L}_H^{\mathcal{N}}(Q1) = \mathcal{L}_H^{\mathcal{N}}(Q2;\ STOP) = \mu\, p.STOP \sqcap l1 \to p$. However, $\mathcal{L}_H^{\mathcal{U}}(Q2;\ STOP)$ does not have the infinite trace $\langle l1, l1, \ldots\rangle$ but $\mathcal{L}_H^{\mathcal{U}}(Q1)$ does.

This is a valid distinction, but not one that can be made without infinite traces.

Conclusion: lazy abstraction, properly, is an operator that can introduce unbounded nondeterminism! However, none of the applications we will be putting it to make use of the extra information $\mathcal{L}_H^{\mathcal{U}}(P)$ gives, and it is only necessary to worry about the introduction of unbounded nondeterminism *if you propose to apply other CSP operators to a lazy abstraction (especially hiding).*

Lazy abstraction over $\mathcal{F}$

The most useful, practically, is the version over the stable failures model $\mathcal{F}$. Divergence is not a concern for the simple reason that it is not recorded in the model:

$$\mathcal{L}_H^{\mathcal{F}}(P) = (P \underset{H}{\|} Chaos_H) \setminus H$$

It is easy to show that the failures of this are the same as for the other two versions (as the $Chaos_H$ gives the extra refusals $\setminus H$ needs to turn it into @L). It has no traces other than those of failures. Hence it is trivially equivalent, in power and content, to $\mathcal{L}_H^{\mathcal{N}}(P)$.

Equivalence of the three approaches

- The determinism of $\mathcal{L}_H^{\mathcal{F}}(P)$, $\mathcal{L}_H^{\mathcal{N}}(P)$ and $\mathcal{L}_H^{\mathcal{U}}(P)$ are all equivalent, and
- if S is a divergence-free and finitely nondeterministic (specification) process, then the three refinements

$$S \sqsubseteq_{\mathcal{F}} \mathcal{L}_H^{\mathcal{F}}(P) \quad S \sqsubseteq_{\mathcal{N}} \mathcal{L}_H^{\mathcal{N}}(P) \quad S \sqsubseteq_{\mathcal{U}} \mathcal{L}_H^{\mathcal{U}}(P)$$

  are all equivalent.

We will only use lazy abstraction when they *are* equivalent, and will therefore write $\mathcal{L}_H(P)$ (without the model).

$\mathcal{L}_H^{\mathcal{F}}(P)$ is the best form of lazy abstraction in *practice* simply because it is directly computable by FDR for finite-state P. It provides the second main example (after deadlock) of using $\mathcal{F}$ to tell us things about divergence-free P by putting them in contexts that might introduce divergence.

Lazy abstraction on FDR

In other words, $\mathcal{L}_H(P)$ is a process with the same failures as

$$(P \underset{H}{\parallel} Chaos_H) \setminus H$$

which cannot diverge. Therefore we use the model $\mathcal{F}$, which ignores any divergence, to reason about it with FDR.

We can, for these purposes, pretend/deduce that any results (determinism or refinement) that FDR tells us of the above formulation in $\mathcal{F}$ are also true of $\mathcal{L}_H(P)$ in $\mathcal{N}$ *even though the checks in the latter model of the same definition might well fail through divergence!*

Lazy abstraction and separability

If $P = P_L \mid\mid\mid P_H$ where $traces(P_L) \subseteq L^*$ and $traces(P_H) \subseteq H^*$, then $\mathcal{L}_H(P) = P_L$ and $\mathcal{L}_L(P) = P_H$.

The proof comes easily from operator definitions over $\mathcal{F}$.

A process that can be factored into two components with disjoint alphabets like this is called *separable*. (One always has a particular partition $\{H, L\}$ in mind.) *Note that a process may well be separable even when it is not defined as a parallel composition.*

Lazy abstraction and separability II

Whenever P satisfies our basic assumptions, then

$$P \sqsupseteq \mathcal{L}_L(P) \mid\mid\mid \mathcal{L}_H(P)$$

(in any of $\mathcal{F}$, $\mathcal{N}$ and $\mathcal{U}$). Additionally, over the traces model $\mathcal{T}$:

$$P \sqsupseteq_T P \setminus H \mid\mid\mid P \setminus L$$

Proof (For $\mathcal{N}$) Suppose $(s, X) \in \mathit{failures}_\perp(P)$. Then, $(s \setminus H, X) \in \mathit{failures}_\perp(\mathcal{L}_H^{\mathcal{N}}(P))$ and $(s \setminus L, X) \in \mathit{failures}_\perp(\mathcal{L}_L^{\mathcal{N}}(P))$.
Since $s \in (s \setminus H) \mid\mid\mid (s \setminus L)$ for any $s \in \Sigma^*$, it follows from the definition of $\mid\mid\mid$ over $\mathcal{N}$ that

$$(s, X) \in \mathit{failures}_\perp(\mathcal{L}_H^{\mathcal{N}}(P) \mid\mid\mid \mathcal{L}_L^{\mathcal{N}}(P))$$

which establishes our result. ∎

The above refinement is, however, often strict: if $P = l \to h \to P$

$$\mathcal{L}_{\{h\}}(P) = l \to \mathit{Chaos}_{\{l\}} \quad \mathcal{L}_{\{l\}}(P) = \mathit{Chaos}_{\{h\}}$$

This is hardly surprising, or else all processes would be separable!

A criterion for separablity

P is deterministic and separable (over $\{H, L\}$) if and only if the processes $\mathcal{L}_H(P)$ and $\mathcal{L}_L(P)$ are both deterministic.

PROOF The "$\Longrightarrow$" half of this result follows from the earlier result.

For the "$\Longleftarrow$" half, suppose both the abstractions are deterministic. Then, as ${}_A\|_B$ preserves determinism, so is

$$\mathcal{L}_H(P) \;{}_L\|_H\; \mathcal{L}_L(P) = \mathcal{L}_H(P) \;|||\; \mathcal{L}_L(P)$$

But we know P refines this process, so the two must be equal by maximality of deterministic processes under refinement. ∎

Conclusions

Lazy abstraction has many more pleasant algebraic properties, for example if $A \cap H = \{\}$, then

$$\mathcal{L}_H(P \underset{A}{\parallel} Q) = \mathcal{L}_H(P) \underset{A}{\parallel} \mathcal{L}_H(Q)$$

See also the exercises in Chapter 12.

Our results imply it is intimately connected with the *factorisation* of processes, and, through separability, with security.

In general, it is the most satisfactory form of abstraction when we believe Hugh's communications with P follow the CSP handshake model.

Mixed abstraction

Sometimes, however, the handshake model is not really accurate: there may be some communications that the environment cannot really delay. (For example, outputs appearing on a VDU.)

While communications are visible, you can allow for this, but care is necessary if they are abstracted: lazy abstraction may well predict deadlocks that are not really possible.

The right abstraction to use is then a *mixed* one, with delayable events treated lazily and signal events abstracted eagerly (i.e., by ordinary hiding). If S are P's signal events then we define

$$\mathcal{M}_H^S(P) = \mathcal{L}_H(P \setminus (H \cap S))$$

This abstraction makes no sense if the hiding introduces divergence (being an infinite sequence of signals to Hugh). Over $\mathcal{F}$

$$\mathcal{M}_H^S(P) = (P \underset{H \smallsetminus S}{\|} \mathit{Chaos}_{H \smallsetminus S}) \setminus H$$

Mixed abstraction often makes sense in dealing with *transactions* with input and output phases.

Example: memory device

$$
\begin{array}{rcl}
Mem(s) & = & write?user?a?v \rightarrow \\
& & \quad (ok.user \rightarrow Mem(update(s, a, v)) \\
& & \quad \lessdot a \in WriteAddresses(user) \gtrdot \\
& & \quad (reject.user \rightarrow Mem(s))) \\
& \Box & read?user?a \rightarrow \\
& & \quad (value.user!fetch(s, a) \rightarrow Mem(s) \\
& & \quad \lessdot a \in ReadAddresses(user) \gtrdot \\
& & \quad (reject.user \rightarrow Mem(s)))
\end{array}
$$

The functions $update$ and $fetch$ respectively modify the state s to execute a write, and look up the value for a read.

Delayable events are $\{\mid write, read \mid\}$, and $S = \{\mid ok, reject, value \mid\}$ are signals.

We will, naturally, assume that the set of users is $\{Hugh, Lois\}$.

Abstracting the memory

Notice how each interaction with the memory is an input/output transaction in two parts.

Lazily abstracting one of the users would create deadlock, because it would assume he/she could refuse the output communications.

Mixed abstraction does not, though one user can affect the other by writing to variables the other can read.
$\mathcal{M}_H^S(Mem(s)) = M_L(s)$, where

$$
\begin{array}{rcl}
M_L(s) & = & write.Lois?a?v \rightarrow \\
& & \quad (ok.Lois \rightarrow M_L(update(s,a,v)) \\
& & \quad \not< a \in WriteAddresses(Lois) \not> \\
& & \quad (reject.Lois \rightarrow M_L(s))) \\
& \Box & read.Lois?a \rightarrow \\
& & \quad ((value.Lois!fetch(s,a) \rightarrow M_L(s) \\
& & \qquad \not< a \notin WriteAddresses(Hugh) \not> \\
& & \qquad \bigsqcap\{value.Lois!v \rightarrow M_L(s) \mid v \in T\}) \\
& & \quad \not< a \in ReadAddresses(Lois) \not> \\
& & \quad (reject.Lois \rightarrow M_L(s)))
\end{array}
$$

This process will be deterministic if, and only if,

$$ReadAddresses(Lois) \cap WriteAddresses(Hugh) = \{\}$$

Reducing specifications

Often we want to prove that complex processes meet simple specifications, that only refer to small subsets of their alphabets. But it then makes no sense to ask the direct question

$$Spec \sqsubseteq Impl\,?$$

because $Impl$ performs events $Spec$ apparently doesn't allow.

For traces specifications, it is easy to work out what to do:

$$Spec \sqsubseteq_T Impl \setminus X$$

where X are the irrelevant events from the point of view of $Spec$.

For example

$$\mu\, p.a \to b \to p \sqsubseteq_T P \setminus (\Sigma \setminus \{a, b\})$$

tests whether a and b alternate irrespective of what else happens.

What about failures?

For a failures specification, we must worry not only about abstracting events $Spec$ does, but also about abstracting its refusals.

Under what circumstances do we want a refusal to count?

- Only when all abstracted events are refused?
- Or whatever abstracted events are refused?

This is, or course, the difference between eager and lazy abstraction.

There is no single right answer: it depends on what you want the specification to say.

Deadlock

We can reduce deadlock freedom to a specification with an empty alphabet via the first approach:

$$\mathbf{div} \sqsubseteq_F P \setminus \Sigma$$

or, if P might terminate,

$$SKIP \sqsubseteq_F P \setminus \Sigma$$

This *exploits* the fact that eager abstraction only records those refusals where all abstracted events are refused. In using it to formulating other specifications you should be careful that this is the effect you want!

Using lazy abstraction

Suppose we want to say that every stable state of P offers a or b. This is best written

$$\mu\, p.(a \rightarrow p) \sqcap (b \rightarrow p) \sqsubseteq \mathcal{L}_{\Sigma \setminus \{a,b\}}(P)$$

Similarly

$$\mu\, p.(a \rightarrow b \rightarrow p) \sqsubseteq \mathcal{L}_{\Sigma \setminus \{a,b\}}(P)$$

says that a and b alternate in P, with the next one due never refused.

This effect is probably what is required in most sub-alphabet failures checks????

Example: token ring

A ring of processes pass a token round, with the posession of the token conveying some privilege. It is a standard solution to the mutual exclusion problem when at most one process may perform a *critical region* at one time....

A very simple implementation:

$$
\begin{array}{rcl}
\mathit{Empty}(i) & = & \mathit{ring}.i \rightarrow \mathit{Full}(i) \\
\mathit{Full}(i) & = & \mathit{ring}.i \oplus 1 \rightarrow \mathit{Empty}(i) \\
 & & \square\ \mathit{start\text{-}cr}.i \rightarrow \mathit{end\text{-}cr}.i \rightarrow \mathit{Full}(i)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Ring} & = & \|_{i=0}^{N-1} (\mathit{Full}(i) \lhd i = 0 \rhd \mathit{Empty}(i), A_i) \\
\text{where } A_i & = & \{\mathit{ring}.i, \mathit{ring}.i \oplus 1, \mathit{start\text{-}cr}.i, \mathit{end\text{-}cr}.i\}
\end{array}
$$

Abstract specification

We might want to specify this in the alphabet $CR = \{\mid start\text{-}cr, end\text{-}cr \mid\}$ as follows:

$$Spec = \bigsqcap\{start\text{-}cr.i \rightarrow end\text{-}cr.i \rightarrow Spec \mid i \in \{0 \ldots N-1\}\}$$

Interpreted as a failures specification, this says that the ring is always able to perform some event in CR, and furthermore the events in this set that do occur comprise the sequencing of a number of disjoint critical regions.

This is an abstract specification of *any* mechanism for implementing disjoint critical regions.

It can be tested by the abstracted check

$$Spec \sqsubseteq \mathcal{L}_{\{\mid ring \mid\}}(Ring)$$

This says "However the passing of tokens round the ring is managed, $Spec$ holds."

Abstracting errors

CSP is frequently used for modelling systems with error-prone components.

The obvious approach is to make the errors appear nondeterministically. An arbitrarily lossy medium might be

$$M = \mathit{left}?x \rightarrow ((\mathit{right}!x \rightarrow M) \sqcap M)$$

But when we build a system out of these:

- There is no easy way to see how system performance depends on error patterns, and
- no way of relating the errors that appear in different components.

The solution: put errors under the control of extra events, and abstract these events once the system is built.

$$M_E = \mathit{left}?x \rightarrow ((\mathit{right}!x \rightarrow M_E) \,\Box\, (\mathit{error} \rightarrow M_E))$$

Criteria for error modelling

A medium that might lose or duplicate messages becomes

$$
\begin{array}{rcl}
C_E(in, out) & = & in?x \rightarrow \; (out!x \rightarrow C'_E(in, out, x) \\
 & & \qquad\qquad \square \; loss \rightarrow C_E(in, out)) \\
C'_E(in, out, x) & = & dup \rightarrow out!x \rightarrow C'_E(in, out, x) \\
 & & \square \; C_E(in, out)
\end{array}
$$

We get a nondeterministically controlled medium by abstraction:

$$\mathcal{L}_E(C_E(in, out)) = C(in, out)$$

If errors are banned, we get a perfect medium:

$$C_E(in, out) \underset{E}{\parallel} STOP = COPY(in, out)$$

These two things are requirements of any model of a 'controlled error' process.

Alternating bit protocol

We can insert two of these into the ABP network, leaving the error events *visible* and *unsynchronised*:

$$ABP_E = ((S \underset{\{|a,d|\}}{\|} (C1_E \mid\mid\mid C2_E)) \underset{\{|b,c|\}}{\|} R) \setminus \{| \ a, b, c, d \ |\}$$

where S and R are as in earlier versions and

$$\begin{array}{rcl} C1_E & = & C_E(a, b) \\ C2_E & = & C_E(c, d)[\![{loss', dup'}/{loss, dup}]\!] \\ E & = & \{loss, loss', dup, dup'\} \end{array}$$

We thus extend E to be $\{loss, loss', dup, dup'\}$.

Having constructed a complete system with visible errors like this, we can

- abstract them immediately, or
- put some bound on what errors can happen by parallel composition with a regulator process first.

With this version of ABP we do the former: you can prove

$$COPY \sqsubseteq \mathcal{L}_E(ABP_E)$$

What does this mean?

Abstraction's implications

$$Spec \sqsubseteq \mathcal{L}_E(Impl_E)$$

means that

- we do not have to *rely* on any error occurring for correctness;
- the traces of $Impl_E$, with errors hidden, satisfy $Spec$.
- whatever errors $Impl_E$ has allowed, if they stop long enough the system will become stable, making an offer allowed by $Spec$.

Thus we have shown that ABP_E is tolerant of an arbitrary *but finite* pattern of errors.

This is subtly different from what we did with the infinite traces model $\mathcal{U}$ – because we have related the pattern of errors to external behaviours rather than the activities of the channels on which they occur. See the book.

Limiting faults

Imagine a supposedly fault tolerant processor built out of four identical (necessarily deterministic) units enclosed in a context which feeds them all identical inputs and gathers all their results.

Any one of them might fail in two ways: halting or going malicious:

$$(Proc(i) \triangle_{halt.i} STOP) \triangle_{byzantine.i} Chaos_{A(i)}$$

Once they are combined to create a system FTP we will have error events

$$E = \{halt.i, byzantine.i \mid i \in \{0, 1, 2, 3\}\}$$

We might reasonably hope the system to be tolerant of up to three halting faults but only one Byzantine one. Thus

$$\mathcal{L}_E(FTP \underset{E}{\parallel} (Halt(3) \sqcap Byzantine(1))$$

should meet our proposed specification........

You can also limit how many errors happen *relative* to the pattern of other events (often $tock$: no more than k errors each N time units).

An inbuilt specification

You can ban errors entirely:

$$NoFaults = System_E \underset{E}{\|} STOP$$

We can thus specify *fault tolerance* without having to create a separate specification of our system:

$$NoFaults \sqsubseteq \mathcal{L}_E(System_E)$$

This says that the system with errors allowed behaves no worse than it does when there are no errors.

This doesn't show that $System$ behaves properly, only no worse than $NoFaults$. Thus $Chaos$ is fault tolerant!

So you must prove that $NoFaults$ meets whatever functional specifications you need. But the above check is useful where a complete functional specification is too complex to develop.

Fault tolerance, specified like this, is not a *behavioural* specification (though, unlike determinism, it is closed under nondeterministic choice).

Information flow and security

Imagine Hugh has higher security clearance than Lois. When can a system allow information from *high* to *low*? Imagine either of the following.

(a) Lois, a spy, attempts to find out about Hugh's interactions with P, the assumption here being that Hugh is unaware of Lois' presence.

(b) Hugh is a 'mole', namely an agent working in the high security levels of his enemies. He is trying to set up a 'covert channel' with Lois, who is his friend on the outside, so that the information he has learned can be relayed back to their spy-masters. Therefore they try to use P to get information from one to the other, and may well have agreed some code in advance.

Those these may seem different, they are, in an absolute sense identical. We would like to ensure that whatever Hugh does, it has no effect on what Lois sees.

Independence

Lois sees everything through her own abstracted view.

When can we be sure that this does not depend on what Hugh does?

When it is deterministic.

We can thus *define*

- P is *lazily* independent from H (written $\mathcal{L}\mathrm{IND}_H(P)$) if and only if $\mathcal{L}_H(P)$ is deterministic, and
- P is *mixed*, or *transaction* independent from H, with respect to the set of signals S (written $\mathcal{M}\mathrm{IND}_H^S(P)$) if and only if $\mathcal{M}_H^S(P)$ is deterministic.

Note the connection with separability: we showed that a process satisfying $\mathcal{L}\mathrm{IND}_H(P)$ and $\mathcal{L}\mathrm{IND}_L(P)$ is separable.

See the book for the notion of semi-separability which is equivalent to lazy independence.

Examples

Let $P1 = h \to l \to P1$. This is insecure because $\mathcal{L}_H(P1) = Chaos_{\{l\}}$, which is nondeterministic. Intuitively, this process is insecure because Lois can see exactly how many events Hugh has performed.

$P2 = (h \to l \to P2) \,\square\, (l \to P2)$ is secure, since $\mathcal{L}_H(P2) = \mu\, p.l \to p$. It is not, however, separable since the abstraction into H is nondeterministic.

$$\begin{array}{rcl} P3 & = & l1 \to P3' \\ & & \square\; h1 \to P3 \\[1em] P3' & = & l2 \to P3 \\ & & \square\; h2 \to P3' \end{array}$$

$P3$ is secure since $\mathcal{L}_H(P3) = \mu\, p.l1 \to l2 \to p$. It is not secure in the reverse direction since

$$\mathcal{L}_L(P3) = \mu\, q.(h1 \to q) \sqcap (h2 \to q)$$

Mixed examples

Assume $ls1, ls2, hs1, hs2$ are signal events

$$\begin{array}{rcl} P4 & = & l1 \to ls1 \to P4 \\ & & \square\ l2 \to ls2 \to P4 \\ & & \square\ h1 \to hs1 \to P4 \\ & & \square\ h2 \to hs2 \to P4 \end{array}$$

This satisfies $\mathcal{M}\mathrm{IND}_H^S(P)$ but not $\mathcal{L}\mathrm{IND}_H(P)$: if Hugh could delay $hs1, hs2$ he could pass information to Lois.

$$P5 = hs1 \to l \to P5$$

satisfies $\mathcal{M}\mathrm{IND}_H^S(P)$ because Lois *knows* Hugh will get the signal $hs1$.

On the other hand

$$\begin{array}{rcl} P6 & = & hs1 \to l1 \to P6 \\ & & \sqcap\ hs2 \to l2 \to P6 \end{array}$$

fails the mixed condition because the value Hugh gets is not predictable on the basis of what Lois legitimately knows.

Recall the Memory...

We observed that (for $\mid T \mid > 1$), $\mathcal{M}_H^S(Mem(s))$ is deterministic iff

$$WriteAddresses(Hugh) \cap ReadAddresses(Lois) = \{\}$$

It should not be a surprise that this is the criterion for security in that system!

Sometimes the security flaws these methods discover are rather more subtle, for example:

- Contention for a file name.
- Contention for physical resources (memory, disc, etc).
- Record locks.
- Effects on the scheduling of low-level transactions brought about by the presence or absence of high-level ones.

See the example files for illustrations...

Sharing a communication medium

Hugh and Lois respectively want to send messages to Henry and Leah, who share their security clearances. Consider the network

$$\begin{aligned} Comm_1 = \;& ((SendH \;|||\; SendL) \underset{\{|in|\}}{\|} Medium) \\ & \underset{\{|out|\}}{\|} (RecH \;|||\; RecL) \end{aligned}$$

where

$$\begin{aligned} SendL &= send.lois?x \rightarrow in.lois.leah!x \rightarrow SendL \\ SendH &= send.hugh?x \rightarrow in.hugh.henry!x \rightarrow SendH \\ RecL &= out.leah?x \rightarrow rec.leah!x \rightarrow RecL \\ RecH &= out.henry?x \rightarrow rec.henry!x \rightarrow RecH \\ Medium &= in?s?t?x \rightarrow out!t!x \rightarrow Medium \end{aligned}$$

$\mathcal{L}_H(Comm_1)$ is nondeterministic: the low users can detect certain sorts of high-level activity through their messages getting blocked.

Solution 1

The first solution is simply to give the low-level users priority in access to resources: we could replace the process $Medium$ by

$$\begin{array}{rcl} Medium2 & = & in.lois?t?x \to out!t!x \to Medium2 \\ & & \Box\ in.hugh?t?x \to Medium2'(t,x) \\ Medium2'(t,x) & = & in.lois?t'?x' \to out!t'!x' \to Medium2 \\ & & \Box\ out!t!x \to Medium2 \end{array}$$

This simply throws away any high-level communication that is sitting in the medium when a low-level one comes along. In the revised system, the low view becomes deterministic (it is a 3-place buffer), but there is the great disadvantage of losing some high level messages.

That in turn can be got around by a further revised medium informing a modified $SendH$ whether each message got through successfully or not, with each thrown-away message being resent, *see book*.

Solution 2

Attaining security by giving priority to low-level users does not seem that attractive!

The main alternative is to limit *both* users access sufficiently to prevent any contention. This may be better or worse.

In the present example, the latter philosophy leads us to ensure that neither user can clog up the system by using flow control:

$$\begin{aligned}
SendL4 &= send.lois?x \to in.lois.leah!x \to \\
&\quad out.lois?x \to SendL4 \\
SendH4 &= send.hugh?x \to in.hugh.henry!x \to \\
&\quad out.hugh?x \to SendH4 \\
RecL4 &= out.leah?x \to rec.leah!x \to \\
&\quad in.leah.lois.Zero \to RecL4 \\
RecH4 &= out.henry?x \to rec.henry!x \to \\
&\quad in.henry.hugh.Zero \to RecH4
\end{aligned}$$

The resulting $Comm_4$ is not only secure, it is separable and equivalent to the composition of a one-place buffer (i.e., $COPY$) between each of the two pairs of processes.

Security *versus* fault tolerance

These two specifications involve similar ideas, and are in fact closely related. Suppose $\mathcal{L}\mathrm{IND}_H(P)$ holds. The determinism of $\mathcal{L}_H(P)$ easily implies that it equals

$$\mathcal{L}_H(P \underset{H}{\parallel} Q)$$

for any divergence-free process Q with alphabet H.

This is reassuring, since it means that what Lois sees does not depend on what high-level process we put in parallel with P. It also means that

$$\mathcal{L}_H(P) = P \underset{H}{\parallel} STOP$$

and so P is 'fault tolerant' if you think of the high-level actions as the errors.

If you think about it, this is actually a rather natural correspondence.

Security *versus* fault tolerance II

On the other hand, a process can be fault tolerant without the abstraction

$$\mathcal{L}_E(P)$$

being deterministic, so the reverse implication does not hold.

You might think of security as an *absolute* property that either a process has or it doesn't, whereas fault tolerance judges a process *relative* to some varying standard.

If it should happen that some error resolves a nondeterministic choice which would have had to be made anyway, this does not matter in judging fault tolerance, but it might convey information to the user about the pattern of errors.

Multiple security levels

In real applications one will frequently have more than two security levels.

One could have any number with a partial order *security policy* relating them: user U_i's behaviour can affect U_j if and only if $clearance(i) \leq clearance(j)$.

There is, however, no need to extend our theory further since all you have to do is to show that, for each security level λ, the collection of all users with this level or below is independent of the collection of all the others. Thus you simply look at a series of two-way partitions of the overall alphabet.

For example, if there is another user with alphabet M whose clearance is mid-way between Hugh's and Lois', then to establish security of a process P you should prove

$$\mathcal{L}\mathrm{IND}_{H \cup M}(P) \quad \text{and} \quad \mathcal{L}\mathrm{IND}_{H}(P)$$

The limitations of security modelling

You should always bear in mind that a security specification formulated in a model can only reveal information flow in those behaviours that the model records.

For example, our models do not record time, and it is easy to imagine how *timing channels* might arise.

It would be possible to address this, at least in part, via the *tock* model of Chapter 14.

One of the main advantages of the determinism-based approach is that the differences between process algebra models tend to disappear for them: it would seem odd if the definition of security depended crucially on whether the spy was using failures equivalence or bisimulation!

Is our condition too strong?

There certainly are processes whose low-level abstraction is nondeterministic that are *really* secure, even though our condition fails them.

For example, any process *implemented* as

$$P_H \mid\mid\mid Chaos_L$$

is secure.

However, just because one implementation of a given value in our models is secure, it does not mean they all are: you would not naturally think of

$$Chaos_{H \cup L} \sqcap LEAK$$

as secure since it is free to behave like $LEAK$ every single time it is used. It is, however, equivalent to

$$Chaos_H \mid\mid\mid Chaos_L$$

No: the model isn't strong enough!

The real problem is that our models do not reveal whether nondeterminism is being resolved innocently or in a way that allows information to flow.

We are being caught out by the usually-desirable decisions to identify many different sorts of nondeterminism, and to make operators distributive (so we can't see *when* a decision is made).

Conclusion: the standard CSP models are not discriminating enough to tell us whether P is secure when it is nondeterministic, just as $\mathcal{T}$ is too weak to tell us whether a nondeterministic process can deadlock.