Checking noninterference in Timed CSP

A.W. Roscoe and Huang Jian

Oxford University Department of Computer Science

Abstract. A well-established specification of noninterference in CSP is that, when high-level events are appropriately abstracted, the remaining low-level view is deterministic. This is not a workable definition in Timed CSP, where many processes cannot be refined to deterministic ones. We argue that in fact "deterministic" should be replaced by "maximally refined" in the definition above. We show how to automated the resulting timed noninterference check within the context of the recent extension of FDR to analyse a discrete version of Timed CSP and how an extended theory of digitisation has the potential both to create more accurate specifications and to infer when processes are non-interfering in the more usual continuous-time semantics.

1 Introduction

Noninterference, a concept introduced by Goguen and Meseguer in [5], is a topic in the theory of computer security: it analyses whether information can flow between users of a system through their joint use of it. In the classic set up there is a high level user (say Hugh) and a low level one (Lois): we might well want to ask whether or not information can pass from Hugh to Lois. Thus noninterference is an asymmetric condition: we might not mind Hugh learning about Lois's activities.

In a practical setting it might be much harder to guard against situations where Hugh is actively trying to pass information to Lois using whatever feature of the system he can (something usually called a *covert channel*, as opposed to Lois spying on an unknowing Hugh. However, without knowing exactly what Hugh might to, proving the absence of information flow in the second scenario is the same as the in the first.

Noninterference is a wonderful specification for theorists to play with, because it exercises the nuances of their semantic models – something that will be well illustrated in the present paper. Given that semantic models tend to be based on things that observers at some level of abstraction can see about processes, it seems natural to pose the question of how one would couch noninterference by giving Lois the same powers of observation as the model, though restricted to her own interface with the system. So while refining a semantic model may be irrelevant to many practical specifications, this is rarely if ever true of noninterference because we can always imagine a more discerning Lois or spy.

Goguen and Meseguer's specification was in terms of machines that strictly alternate inputs and outputs, and on each cycle have an input, then an output, with each of its users. Process algebras like CSP offer a rather more flexible way of describing how processes look, and since they are essentially ways of describing *interaction*, quickly became an important focus of noninterference research. Initial characterisations in process algebras (for example [1, 6, 28]) had much in common with those of [5], but later ones such as those of Roscoe, Woodcock and Wulf [27, 21], and Focardi and Gorrieri [3] made much more use of the particular expressive qualities of process algebras. It is in this world that the present paper sits.

Different types of semantics (whether process algebra or otherwise) give different perspectives on noninterference. If the semantic model we are using does not capture some notion of behaviour that Lois might observe, then any specification of noninterference based

on that model is not going to capture information about Hugh's actions that she can see in that way. One obvious possibility is time. Neither standard input/output semantics of sequential programs nor most process algebras pay any attention to how long our system takes to perform its operations, or the wait between one communication and the next. Therefore none of the formulations of noninterference in the papers cited above can identify *timing channels*, one of the most common types of covert channel. The formulations we give in this paper are designed for exactly that purpose.

A further important question is whether or not we try to distinguish different sorts of nondeterminism – unpredictable behaviour by the system – from one another. Nondeterminism can either protect against information flow or allow it. If what Lois says covers the same nondeterministic range no matter what Hugh does it is impossible for her to deduce anything definitively about his actions, but on the other hand what he does (for example his timing) might affect the resolution of the nondeterminism in her view.

The relationship between refinement, nondeterminism and noninterference has generated considerable debate over the years. In particular one needs to be careful not to describe a system as secure and yet find it has insecure refinements – an instance of the so-called *refinement paradox*. Morgan and McIver have written about this issue – mainly in the context of imperative programs [13, 12], for example, by introducing shadows that assert unrefinable ignorance. We will find in this paper that there seems to be a greater need in Timed CSP than in the original CSP for ways of determining whether or not nondeterministic programs, possibly not maximal in the conventional refinement order, satisfy noninterference.

The rest of this paper is structured as follows. We first recall the CSP and Timed CSP languages. We then introduce and analyse the continuous and discrete semantic models we use for Timed CSP, deriving some new structural properties. A further background section recalls the functionality of FDR including its new timed capabilities. Section 5 recalls the definition of noninterference from [27, 21], and shows that it does not work in the same form in the standard model for Timed CSP (which uses a continuous model of time) or the corresponding model using discrete time. We then show how a revised formulation of the basic principles allows us to capture what is required.

The theory of *digitisation* [14, 15] allows us to relate the behaviour of Timed CSP processes in discrete and continuous time. We investigate the implications of this for systems modelled in the continuous models of Timed CSP, and show that one can provide results about the noninterference properties of a process's continuous semantics by analysing a suitable discrete semantics. To do this we establish some generalised results about digitisation.

The discrete-time form of Timed CSP has recently [2] been implemented in the CSP refinement checker FDR [20]. We show that the reformulated definition of noninterference can be implemented directly in that, though not so directly as the untimed variant, since it cannot use FDR's built-in determinism check. By applying this to some relatively simple case studies, we see some of the types of timing channels that can arise in shared-use systems and some strategies for avoiding these.

Finally, we contemplate potential application areas of timed noninterference analysis, including Cloud security.

The reader will discover that some of the constructions and arguments contained in this paper are complex. To keep them as simple as possible we make a number of assumptions:

1. The alphabet Σ over which we build processes is finite.
2. We only consider finitely nondeterministic CSP and Timed CSP: given the first assumption this just means that all uses of nondeterministic choice $\sqcap$ are over finite sets.

3. While we frequently use termination ($\checkmark$) and sequential composition in building process descriptions, the processes we test for noninterference will never terminate. We therefore ignore the complications of termination when defining and analysing the semantic models and in formulating noninterference conditions.

The present paper has its origins in the doctoral research of the second author [8], some of which relating to discrete Timed CSP was reported in [9]. This examined a variety of timed models of CSP-like processes and explored how established noninterference theories extended to them. We are now able to extend the parts of [8] on discrete and continuous Timed CSP by proving relationships between them and implementing them in FDR.

Acknowledgements

We are grateful to Joël Ouaknine for discussions on discrete Timed CSP and digitisation, and to Phil Armstrong for implementing Timed CSP in FDR.

The work reported in this paper was partially supported by grants from EPSRC and ONR.

2 The language of CSP and Timed CSP

CSP is a language which describes patterns of communication in some alphabet Σ of actions that are handshaken between the process and its environment. There are additional actions $\checkmark$, a signal for successful termination and τ, an invisible action representing internal progress within a process. In the original "untimed" treatment of CSP these patterns of communication include the order of actions, which sets are offered and maybe the ways in which possibilities branch, but not the exact times these things happen.

The following is a brief introduction to the main parts of the language: much more complete explanations can be found elsewhere [22, 29, 24].

There are process constants representing important patterns of communication: $STOP$ is a process that does nothing, while $SKIP$ just terminates. **div** just *diverges* by performing τs for ever. RUN_A is always ready to perform any event from $A \subseteq \Sigma$ while $CHAOS_A$ can always both accept and refuse any event from A.

There are operators for introducing communications: $a \rightarrow P$ and $?x : A \rightarrow P(x)$ allow an individual member or choice of actions from Σ. $P \,\Box\, Q$ makes the choices of both P and Q available to the environment, while $P \sqcap Q$ allows the process to select which of P and Q to behave like.

We can put processes in parallel that influence each other by synchronising on some of their events from Σ: $P \underset{A}{\parallel} Q$ makes them synchronise on the events they perform in A, $P \;{}_A\|_B\; Q$ makes them synchronise in $A \cap B$, while $P \,|||\, Q$ just lets them run freely.

$P \setminus A$ represents P running but with all events in A hidden: turned into τs. $P[\![R]\!]$ applies the relation $R \subseteq \Sigma \times \Sigma$ (usually assumed to be total on the events P uses) to P's actions: whenever P performs a, $P[\![R]\!]$ gives the environment the choice of all the b such that $a \,R\, b$.

CSP provides three ways of one process handing over to another P; Q runs them in sequence: P until it terminates via $\checkmark$ and then Q. $P \,\triangle\, Q$ allows Q to *interrupt* P by performing any visible event, while $P \,\Theta_A\, Q$ runs P until it performs any action in $A \subseteq \Sigma$, at which point Q starts.

There are also indexed versions of a number of these operators, and in many contexts we use infinitary versions of the choice operators. Recursive definitions are used in a rich

variety of ways, including defining infinite families of processes in mutual recursions. CSP is therefore a very rich language and is capable of describing many patterns representing both implementations and specifications.

In this paper we will use the "blackboard" style of the operators used above, but in fact our implementations of the ideas in this paper are all in the ASCII version of CSP, known as CSP_M which combines ASCII versions of the above operators with a Haskell-like functional programming language.

Timed CSP [17, 18, 29] does not need much more description because it is the same language given a timed interpretation. In our treatment there is only one new construct: $WAIT\ t$ behaves exactly like $SKIP$ but takes the non-negative time t before it terminates ($\checkmark$). As implemented in FDR it gives the programmer the option to assign a non-negative completion time to each event $a \in \Sigma$: in $a \rightarrow P$ and $?x : A \rightarrow P(x)$ there are $et(a)$ time units between the occurrence of the event a (assumed to be instantaneous) and the following process starting up. The other main principle underlying the timing is that we assume that as soon as any event is enabled in a process (either because like τ it needs no collaboration from the environment, or because the environment does allow it) some event does happen. This is the principle of *maximal progress*.

Some versions of Timed CSP include explicit time-out and timed interrupt operators: $P \rhd_t Q$ offers initial choices of P for time t and then lets Q take over if P has not communicated; $P \triangle_t Q$ makes Q take over after time t even if P has communicated (unless P has terminated). But since both of these can be defined in terms of $WAIT$ and other operators, we will not regard these as primitive operators here.

- $P \rhd_t Q = (P \,\Box\, (WAIT\ t;\ timeout \rightarrow Q)) \setminus \{timeout\}$ where $timeout$ is a new event with $et(timeout) = 0$.
- $P \triangle_t Q = (P \triangle (WAIT\ t;\ interrupt \rightarrow Q)) \setminus \{interrupt\}$ where $interrupt$ is a new event with $et(interrupt) = 0$.

Note how both these constructions depend on maximal progress: as soon as the fresh event becomes available it happens, forcing either resolution of the $\Box$ or an interrupt.

The natural timed interpretations of the processes **div**, RUN_A and $CHAOS_A$ all breach an important principle of Timed CSP: they allow an infinite number of events in a finite time. Timed CSP assumes to the contrary that processes only perform a finite number in any finite interval. What we assume in this paper is any complete Timed CSP system has this property: it can sometimes be useful to use RUN_A and $CHAOS_A$ in these programs as long as they synchronise all their actions with a process that does have the *no-Zeno* property.

We will find in this paper that it is frequently useful to restrict Timed CSP so that all delays introduced by language constructs are integers. In other words, $et(a) \in \mathbb{N}$ and, in $WAIT\ t$, $t \in \mathbb{N}$. The language subset satisfying this restriction will be termed *integer Timed CSP*.

3 Semantic models

The most standard model of untimed CSP is the failures-divergences model alluded to above in which each process is represented as a pair (F, D) of sets of behaviours. F comprises *failures*, namely combinations (s, X) of a finite trace and a set X that the process can refuse in a stable state after s. D is the set of processes on which it can *diverge*, namely engage in an infinite consecutive sequence of τ actions. The model is *divergence strict*,

namely if $s \in D$ then $s^\frown t \in D$ and $(s^\frown t, X) \in F$ for all t and X: this does not imply that the process can really perform all these extra behaviours, but rather than we choose not to know whether it can or not and simply *assume* they are.

Aside from these divergence-closure conditions, the observation that if (s, X) is a failures then so is (s, Y) for $Y \subseteq X$ and the property that the set of all traces $\{s \mid (s, X) \in F\}$ is nonempty and closed under prefix, there is one further healthiness condition that characterises which pairs (F, D) represent realistic processes. That is

F3 $\quad (s, X) \in F \wedge Y \cap \{a \mid (s^\frown \{a\}, \emptyset) \notin F\} \Rightarrow (s, X \cup Y) \in F$

In other words, whenever our process refuses a set X it must also refuse (if offered) an extension of X by events that are *never* possible after s. **F3** ensures that the process has enough traces to be consistent with its refusal sets.

With exception of additional properties that are used to govern the behaviour of the special event $\checkmark$ representing successful termination (which we ignore, for simplicity, in this paper), these properties completely determine $\mathcal{N}$ over any given alphabet Σ. This model has many important properties, one of which is that (given the CSP language described in [24] *fully abstract* with respect to deciding whether any process is *deterministic* in the following sense:

- A deterministic process (F, D) is divergence free (i.e. $D = \emptyset$)
- It never has the choice whether to accept or refuse any event (i.e. if $(s^\frown \langle a \rangle, \emptyset) \in F$ then $(s, \{a\}) \notin F$), which is equivalent to saying that the failures (s, X) are exactly the ones forced from the set of traces by the property **F3** above.

This concept of determinism is one that relates to how a process can be observed rather than its internal construction. For example the operational semantics of the process $((a \rightarrow b \rightarrow a \rightarrow STOP) \sqcap b \rightarrow STOP) \setminus \{a\}$ branch, so one cannot be certain what state one is in after any trace, but no matter what happens a b followed eventually by $STOP$ occur.

The combination of the principle of maximal progress and the need to make models compositional under the CSP hiding operator (which turns visible actions into τs that are forced before time passes) makes the range of models for Timed CSP more restricted than for untimed. It is necessary to record the set of events refused at every point in a behaviour where time advances. Divergence is a much reduced issue, since thanks to the no-Zeno assumption any divergence is necessarily spread over infinite time – which when we are modelling time simplifies things greatly. In fact divergence will not be considered in the models we use in this paper.

In the case of continuous time this means that we have to record refusals as a subset of $\Sigma \times \mathbb{R}^+$ to accompany traces which attach a time in $\mathbb{R}^+$ (the non-negative real numbers) to each event, where the times increase, not necessarily strictly, through the trace. In fact, timed refusals are unions of sets of the form $X \times [t_1, t_2)$ where $0 \leq t_1 < t_2 < \infty$ –*refusal tokens*. $[t_1, t_2)$ is a *half-open interval* that contains t_1, all x with $t_1 < x < t_2$ but not t_2. This corresponds to the idea that if an event happens *at* time t then the refusal recorded at that time is the set of events refused at the same time *after* the event. So in $a \rightarrow P$, there will be behaviours in which a occurs at time 1, all events other than a are refused in the interval $[0, 1)$ and, on the assumption that the event a takes time δ to complete, all events including a are refused in the interval $[1, 1 + \delta)$.

So the *Timed Failures model* consists of pairs of the form $(t, \aleph)$, where t is such a timed trace, and $\aleph$ is such a timed refusal. First introduced in [18], there have been a number of variants of this model over the years, the main points of difference being:

- Is causality permitted between simultaneous events: can one have the timed trace $\langle (a,1),(b,1)\rangle$ but not the timed trace $\langle (b,1),(a,1)\rangle$?
- Can an event take zero time: in $a \rightarrow b \rightarrow P$ can the b happen at the same time as the a? (This question is very closely linked to the previous one: if the answer to this one is "yes", then the first must also be answered affirmatively.)
- Does recursion take time to unfold or not: is $\mu\, p.F(p)$ equivalent to $WAIT\ \delta;\ F(\mu\, p.F(p))$ for some $\delta > 0$ or just $F(\mu\, p.F(p))$?
- How is the assumption of no Zeno behaviour enforced? This says that only finitely many actions can occur in a finite time.
- Is information included about *stability*? This is the dual of divergence: the time after which no further internal actions occur without a visible one having occurred first.
- Can traces s and timed refusals $\aleph$ extend through all time or must they be finite. In either case they are always restricted so that up to any finite time they only have finitely many events or are the union of finitely many refusal tokens.

[14, 15] and [29] agree on most of these points, and give the same equivalence over Timed CSP restricted to finitely nondeterministic constructs. We agree with them on all of them except the one where they differ, which is the last. So the answers to the all but that question will be yes, yes, no, usually restricting recursions to ones which are *time guarded*, never starting a recursive call until some delay of at least $\delta > 0$ has been introduced by the context of the call, and no.

On the last point we will restrict to finite traces but allow timed refusals extending through all time. Both make later constructions easier, but under our assumption of finite nondeterminism do not change the expressive power of the model.

The axiom that coincides with F3 is more complex both because it deals with the richer structure of timed failures and because it captures the temporal concept of *no instantaneous withdrawal* or NIW: if a process cannot refuse an event up to a given time, then it can perform it at that time. The intuition here is that if event a was offered before time t when some internal event x occurred that removed the option of a, then a was a valid option to x at the point where it occurred and so could have happened instead. This property is important to the theory of digitisation and will play a major role in this paper. In this paper we will call it the continuous forcing axiom **CF**.

$$(s,\aleph) \in P \Rightarrow \exists\, \aleph' \subseteq \aleph.\, \forall\, t.\, \forall\, a.$$
$$\neg\, \exists\, \epsilon > 0.\{a\} \times [max\{end(s \upharpoonright t), t-\epsilon\}, t+\epsilon) \subseteq \aleph' \Rightarrow (s \upharpoonright t\hat{}\langle (a,t)\rangle, \aleph' \upharpoonright t) \in P$$

This says that whatever timed refusal $\aleph$ is actually observed by an experimenter alongside the timed trace s, there is an extension $\aleph'$ which records all the events the process would have refused (if offered) alongside the trace s through all time. The fact that these are *all* the refused events means that it must perform all actions that are either not in the set or have been withdrawn from the offer at the present instant.

The only other properties required to define the model are the following

- A process is non-empty, specifically containing $(\langle\rangle, \emptyset)$.
- Whenever $(s\hat{}s', \aleph) \in P$ and $\aleph' \subseteq \aleph \upharpoonright \mathit{begin}(s')$, then $(s', \aleph') \in P$, including the case where $s' = \langle\rangle$ and so $\mathit{begin}(s') = \infty$.
- For each P there is a bound on how many events it can perform in any finite time:

  $$\forall\, t.\, \exists\, .n.s \in \mathit{traces}(P) \Rightarrow \#s \upharpoonright t \leq n$$

  Our assumptions of finite alphabet and finite nondeterminism make this last property unproblematic.

The (continuous) timed failures model $\mathcal{F}_{\mathbf{T}}$ will be the set of all sets P of timed failures satisfying all of the above.

A corresponding model $\mathcal{F}_{\mathbf{DT}}$ exists for discrete time: where time is measured in discrete units, separated by some marker such as an event *tock* representing the regular passage of time. $\mathcal{F}_{\mathbf{DT}}$ and variants have been studied and described in [14–16, 11, 8, 24]. In Timed CSP the processes do not communicate this event: you should think of it as a clock in the hands of an external observer. The *discrete timed failures model* has behaviours that consist of a trace consisting of events including *tock*, including a refusal set of events before each *tock* and at the end. None of the refusal sets include *tock*. Another equivalent presentation is as a sequence of failures (s, X) where there is a notional *tock* between each consecutive pair. For consistency with the continuous treatment above, in this paper we will assume that the behaviour is infinite but contains only finitely many non-*tock* events. [Thus it contains a record of refusals at all integer times.]

The events between two consecutive *tock*s are thought of as occurring at one of a discrete series of "moments", and the refusal set records what is refused at the point time advances, exactly in the spirit of the continuous timed failures model.

Because of this structure in which all behaviours have infinitely many *tock*s, the usual empty trace $\langle\rangle$ or $(\langle\rangle, \emptyset)$ representing a process doing nothing is replaced by $\Delta = \langle \emptyset, tock\rangle^{\omega}$: where time passes for ever but nothing is seen to be refused.

While the complete Timed CSP language can be given a compositional semantics over $\mathcal{F}_{\mathbf{T}}$, to have a semantics over $\mathcal{F}_{\mathbf{DT}}$ a program needs to use only integer delays in *WAIT*s, event timings and any other places where a delay is introduced. In other words it must be an integer Timed CSP program as defined earlier.

$\mathcal{F}_{\mathbf{DT}}$ also has the NIW property: if an event cannot be refused before a *tock*, then it is possible after the *tock*. Intuitively, the withdrawal of the offer of some event a after a *tock* occurs because the *tock* enables some τ that changes the state. But since τ is after the *tock*, a is possible also up to that same point after *tock*. The property analogous to **CF** is the discrete forcing axiom **DF**:

$$s \in P \Rightarrow \exists\, s' \supseteq s.\, \forall\, a.$$
$$s' = s_1 \hat{}\langle X, tock\rangle \hat{} s_2 \wedge a \notin X \Rightarrow (s' \hat{}\langle a\rangle \hat{}\Delta \in P \wedge s' \hat{}\langle X, tock, a\rangle \hat{}\Delta \in P)$$

Here, $s \subseteq s'$ means that the traces (of ordinary events and *tock*s) in the two behaviours are the same, and that each refusal in s is a subset of the one at the corresponding point in s'.

DF can be paraphrased as saying that each observed behaviour of P must have arisen from an actual behaviour of the underlying machine performing the same trace, where the complement of the refusal at each *tock* were the events being offered at that point. Each such event could therefore occur either before or (because of NIW) after the *tock*.

In both $\mathcal{F}_{\mathbf{T}}$ and $\mathcal{F}_{\mathbf{DT}}$, the no instantaneous withdrawal property gets in the way of the idea of determinism. If we continue to identify determinism with processes being unable, after any given trace, both to accept and refuse any event, it is clear that no process that ever withdraws an offer can be deterministic.

Definition 1. *[19] A timed process is said to be* quasi-deterministic *when any visible event that occurs at time t* either *is the first to occur at that time and has not been refused in an interval up to t,* or *is not refused at t. Specifically:*

– *Over* $\mathcal{F}_{\mathbf{T}}$*, if* $(s\hat{}\langle(a,t)\rangle, \aleph) \in P$ *then* either $t > 0$ *and* $end(s) < t$ *and there exists* $\epsilon > 0$ *such that* $(s\hat{}\langle(a,t)\rangle, \aleph \cup \{a\} \times [t-\epsilon, t)) \notin P$ or *there is no* $\epsilon > 0$ *such that* $(s, \aleph \restriction t, \aleph \cup \{a\} \times [t, t+\epsilon) \in P$.
– *Over* $\mathcal{F}_{\mathbf{DT}}$*, if* $s\hat{}\langle a\rangle\hat{}s' \in P$ *then* either s *has the form* $s''\hat{}\langle X, tock\rangle$ *and* $s''\hat{}\langle X \cup \{a\}, tock, \{a\}, tock\rangle\hat{}\Delta \notin P$ or $s\hat{}\langle\{a\}, tock\rangle\hat{}\Delta \notin P$.

In each model, the first case allows a process to be quasi-deterministic even though there are traces after which an event can both be accepted and refused thanks to NIW. The following lemma, which follows immediately from the above, gives characterisations of processes that are not quasi-deterministic.

Lemma 1. *(i) Over* $\mathcal{F}_{\mathbf{T}}$ *the process* P *is non-quasi-deterministic if and only if it has a behaviour* $(s, \aleph)$ *such that one of the following applies:*
(a) There exist a *and* $t_2 > t_1 = end(s)$ *such that both* $(s\hat{}\langle(a,t_1)\rangle, \aleph \restriction t_1)$ *and* $(s, \aleph \cup \{a\} \times [t_1, t_2))$ *are in* P.
(b) There exist a *and* $t_3 > t_2 > t_1 \geq end(s)$ *such that both* $(s\hat{}\langle(a,t_2)\rangle, \aleph \restriction t_2)$ *and* $(s, \aleph \cup [t_1, t_3))$ *are in* P.
(ii) Over $\mathcal{F}_{\mathbf{DT}}$ *the process* P *is non-quasi-deterministic if and only if it has a behaviour* $s\hat{}\Delta$ *such that one of the following applies:*
(c) s *does not end in tock, and both* $s\hat{}\langle a\rangle\hat{}\Delta$ *and* $s\hat{}\langle\{a\}, tock\rangle\hat{}\Delta$ *belong to* P.
(d) s *has the form* $s'\hat{}\langle X, tock\rangle$*, and both* $s\hat{}\langle a\rangle\hat{}\Delta$ *and* $s'\hat{}\langle X \cup \{a\}, tock, \{a\}, tock\rangle\hat{}\Delta$ *are in* P.

Some simple examples are:

– $(WAIT\,1 \,\Box\, a \rightarrow STOP);\ STOP$, which in either model can either perform a or refuse it at time 1 is quasi-deterministic.
– On the other hand

$$((WAIT\,1 \,\Box\, a \rightarrow STOP);\ STOP) \sqcap (WAIT\,1;\ ((a \rightarrow STOP) \,\Box\, (b \rightarrow STOP)) \setminus b)$$

is not because behaviours in which the second a occurs after the refusal of a at any point in $[0,1)$ (continuous model) or before the first *tock* (discrete model) have neither of the alternative properties.
It is interesting to note that this process is, in both models, equivalent to $WAIT\,1;\ ((a \rightarrow STOP) \,\Box\, (b \rightarrow STOP)) \setminus \{b\}$, namely a process that can always refuse a but can also perform it at time 1.

Over the continuous model $\mathcal{F}_{\mathbf{T}}$, quasi-determinism exactly captures the concept of refinement maximality[1].

Theorem 1. *Over* $\mathcal{F}_{\mathbf{T}}$ *a process* P *is refinement maximal (i.e.* $Q \sqsupseteq P \Rightarrow Q = P$*) if any only if it is quasi-deterministic.*

Proof The proof of this rests on the structural axiom **CF** quoted above. In fact both quasi-determinism and refinement maximality are equivalent to the following:

$$(*)\quad \begin{array}{l} \forall (s, \aleph) \in P.\, \exists! \aleph'.\aleph' \supseteq \aleph.(s, \aleph') \in P \wedge \forall\, t \geq 0.\, \forall\, a. \\ (s \restriction t\hat{}\langle a\rangle, \emptyset) \notin P \Leftrightarrow (\exists\, \epsilon > 0.[max\{end(s \restriction t), t-\epsilon\}, t+\epsilon) \subseteq \aleph') \end{array}$$

[1] It was not equivalent to maximality in [19] because that paper used the concept of stability discussed above.

In other words, there is a unique (i.e. $\exists!$) timed refusal $\aleph'$ associated with every trace with the property that the trace can be extended at any time by any event just when that event has either just been withdrawn in $\aleph'$ or is not refused in $\aleph'$. $\aleph'$ is thus the only possible complement of the set of events the process actually offers through the trace.

Note that the uniqueness of $\aleph'$ means that, for any fixed s, we are certain to get the same $\aleph'$ for any $\aleph$ such that $(s, \aleph) \in P$. So in particular every $\aleph$ must be a subset of the one generated by (*) for $(s, \emptyset)$.

If P does satisfy the above then it is straightforwardly quasi-deterministic. It is maximally refined because if $(s, \aleph) \in P - P'$ for some refinement P' then either $s \in traces(P')$ or not. If so we get a contradiction because the $\aleph'$ given for s in P' by **CF** necessarily omits some $\{a\} \times [t_1, t_2) \subseteq \aleph$ where no event in s appears in the given interval. **CF** then implies that $s \restriction t_1 \hat{} \langle (a, (t_1 + t_2)/2) \rangle$ is a trace of P' even though it cannot be one of P. We can thus infer that the $\aleph$s associated with each trace s of P' are the same as those for s in P, and that there is is some shortest trace $s \hat{} \langle (s, t) \rangle$ in P but not in P'. That gives a contradiction since the trace $s \hat{} \langle (a, t) \rangle$ is implied by **CF** applied to trace s.

For any process $P \in \mathcal{F}_{\mathbf{T}}$ one can construct a refinement P' satisfying property (*) by induction on the length of trace: we start with $\aleph^{\langle\rangle}$ chosen by **CF** for $(\langle\rangle, \emptyset)$. The length 1 traces are then just those implied by $(\langle\rangle, \aleph^{\langle\rangle})$ under **CF**. Each such trace $s = \langle (t, a) \rangle$ gives an $\aleph^s$ implied by **CF** from $(s, \aleph^{\langle\rangle} \restriction t)$ where it can be assumed that $\aleph^s \restriction t = \aleph^{\langle\rangle} \restriction t$. We then simply continue this process inductively for longer and longer traces, and finally identify P' with the set of all $(s, \aleph)$ such that the trace s is generated at some point in this process and $\aleph \subseteq \aleph^s$.

P' satisfies property (*) by construction. It is necessarily equal to P if the latter is refinement maximal, demonstrating that maximality implies (*).

If P is quasi-deterministic, then P' omits no behaviour of P: if it did then this behaviour would differ from those picked for P' after some shortest trace s on which they agree. Whether the extra behaviour were a refusal token $\{a\} \times [t_1, t_2)$ or event (a, t) after s, it would contradict quasi-determinism using arguments similar to the above.

This concludes the proof of Theorem 1. ∎

The equivalence shown above to (*) establishes the following corollary.

Corollary 1. *A quasi-deterministic process in $\mathcal{F}_{\mathbf{T}}$ is completely determined by its traces. In other words, if P and Q are quasi-deterministic and have the same set of traces, then $P = Q$.*

It is natural to expect, given the above, that quasi-determinism corresponds to refinement maximality over $\mathcal{F}_{\mathbf{DT}}$ as well, but it is not true. Consider the processes

$$P_1 = a \rightarrow STOP$$

$$P_2 = ((a \rightarrow STOP) \,\Box\, WAIT\,1);\ (WAIT\,1;\ a \rightarrow STOP)$$

Over the continuous model these are not comparable in the refinement order: in fact they are both quasi-deterministic and therefore maximal. Note in particular that there are traces that the first process has but the second does not, for example $\langle (a, 1.5) \rangle$.

However over the discrete model every trace of P_1 ($\{\langle\rangle\} \cup \{\langle (a, n) \rangle \mid n \in \mathbb{N}\}$) is also one of P_2: the trace $\langle (a, 1) \rangle$ is present by NIW. However the first has less refusals since it does not have the behaviour $\langle \emptyset, tock, \{a\}, tock \rangle \hat{} \Delta$ which the second does. Therefore P_2 is not maximal even though it satisfies the definition of quasi-determinism over $\mathcal{F}_{\mathbf{DT}}$.

In order to be refinement maximal over $\mathcal{F}_{\mathbf{DT}}$, any withdrawal of an offer must be for at least two time units.

Theorem 2. *Over $\mathcal{F}_{\mathbf{DT}}$, a process is refinement maximal if and only if it is quasi-deterministic* and *satisfies the following:*

– *If $s^\frown\langle a\rangle^\frown\Delta$ and $s^\frown\langle\{a\}, tock\rangle^\frown\Delta$ both belong to P, then $s^\frown\langle\{a\}, tock, \{a\}tock\rangle^\frown\Delta \in P$.*

In other words, if a is withdrawn at time t then it must be withdrawn for two time units.

So if we were to model the Timed CSP process

$$((a \rightarrow STOP) \,\Box\, WAIT\,1);\ (WAIT\,1;\ a \rightarrow STOP)$$

in a domain where one time unit between *tock*s is 0.5 of the one used to measure *WAIT*s we would get a refinement-maximal process, since a is withdrawn for 2 of the *tock*-units.

4 The functionality of FDR

FDR is a model checker which by now has many features. That relevant to this paper is its ability to check for two properties of processes: refinement over a variety of models, and determinism. These are well known and well documented for untimed CSP. For that, the main models for calculating refinement are traces, failures and failures-divergences, the last two being equivalent for divergence-free processes.

Determinism means the combination of divergence freedom and the process never having both the trace $s^\frown\langle a\rangle$ and the failure $(s, \{a\})$ (i.e. it can perform s and then refuse to perform any member of the set $\{a\}$.) As well as the natural determinism check over the failures-divergences model, FDR can also attempt to perform a check which ignores potential divergence. When the latter is known to be impossible this gives the same result, but there is a use relevant to this paper where divergence is sometimes possible and makes FDR fail to produce an answer. See Section 5 below.

All the above is well known for untimed CSP, but new capabilities [2] allow it to do these things in the context of *integer* Timed CSP. That is, Timed CSP where all *WAIT*s and event delays are integer, and where we only record the integer part of the time when each event occurs. This is reported in [2], and allows the user to mix, in a single script, Timed CSP and "*tock*-CSP", namely the language of untimed CSP in which the passage of time units is represented via the event *tock* that is included in programs like ordinary members of Σ. In fact FDR's implementation of Timed CSP works by translating that language to a special form of *tock*-CSP that is semantically equivalent to it over $\mathcal{F}_{\mathbf{DT}}$.

In running both Timed CSP and *tock*-CSP, FDR requires the user to apply an operator that gives internal events τ priority over the passage of time via *tock*. This operator is

$$pri(P) = priority(P, \{\}, \{tock\})$$

in the priority notation used by FDR. This is needed to achieve maximal progress as described above.

FDR can perform refinement checks between Timed CSP processes, where time is represented via the *tock* event, using all the usual models that it supports (traces, failures etc). These are frequently the most appropriate models for comparing a complete Timed CSP process against a specification, often written in *tock*-CSP, but it is important to remember

that most of them are not compositional over Timed CSP: one cannot for example infer over Timed CSP that $P \sqsubseteq_T Q \Rightarrow C[P] \sqsubseteq_T C[Q]$ for a Timed CSP context $C[\cdot]$.

FDR is also capable of checking refinement in $\mathcal{F}_{\mathbf{DT}}$ between integer Timed CSP programs, which is compositional. At present this is done by using the refusal testing model embedded within FDR and a transformation on the Timed CSP processes it generates, but it may be implemented directly in future versions.

5 Noninterference via determinism?

This ability to characterise a deterministic process even though its internal construction includes nondeterministic choice is the key to the definition of noninterference in [27, 21, 22]. Given a process P with two users whose disjoint alphabets H and L partition its own, we can say that a process P can transmit no information from H to L if $\mathcal{A}_H(P)$ is deterministic, where $\mathcal{A}_H(P)$ abstracts away the behaviour of a most nondeterministic user controlling H. We consider all of the things the high level might do on its side of P, take the nondeterministic choice of all of them, and specify that the low level user's view must be deterministic despite that.

This is a very elegant definition, but (as with every other definition of noninterference over complex behaviours that we are aware of) it is not perfect:

- It only captures information flow that is visible in the patterns of behaviour recorded in the model being used: so if we are using the failures-divergences model a process can pass this specification despite having timing channels. [However, as remarked in [23], this definition is insensitive to which of a large class of untimed models for concurrency are used.]
- It does not distinguish between nondeterminism that is causally linked to the actions of H and that which is intrinsic to P's behaviour. Even nondeterminism that is built in to help conceal H behaviour from L will mean that P is deemed insecure. Thus the definition is only exact for deterministic P; for nondeterministic P it is conservative in the sense that it never deems an insecure process secure, but might say a secure one is insecure. [As discussed in [22], for example, the class of models in which $\mathcal{N}$ rests are simply incapable of making the necessary distinctions when P is nondeterministic.]

In this paper we are addressing the first of these problems. To handle the second without admitting insecure processes as secure would require much more operational and intensional models of processes.

The abstraction used should capture all the ways in which P can be influenced by the process interacting with it in H. If this interaction follows the standard CSP model then potentially that user can not only select which H action is picked when several are made available, but also whether one is selected at all. In this case the correct abstraction to use is *lazy abstraction*, defined over the stable failures model (in which the divergence component of $\mathcal{N}$ is replaced by one of finite traces) by

$$\mathcal{L}_H(P) = (P \parallel CHAOS_H) \setminus H$$

(The use of $\mathcal{N}$ with this formulation creates problems because it can introduce divergence that is not appropriate.)

An alternative form of abstraction called *mixed abstraction* is used when H is partitioned into two sets H_D and H_S, where the user is assumed to be able to delay the first but not

the second, which are *signals* from process to user.

$$\mathcal{L}^{H_S}_H(P) = (P \parallel CHAOS_{H_D}) \setminus H$$

In this paper we will concentrate on lazy abstraction, but everything we do would work under an analogous treatment of mixed abstraction.

So the definition of noninterference on which our work in this paper will be based is the following.

Definition 2. *The process P is said to be* lazily independent *of H over the failures-divergences model $\mathcal{N}$ if $\mathcal{L}_H(P)$ is deterministic.*

Numerous examples of how this definition works in characterising information flow can be found in [21, 27, 22], as can results such as the demonstration that a deterministic process P is equivalent to the independent parallel composition $P_H \mid\mid\mid P_L$ where $P_L = \mathcal{L}_H(P)$ and $P_H = \mathcal{L}_L(P)$ if and only if both these processes are deterministic. In other words P is *separable* if and only if P is lazily independent of both L and H.

This immediately suggests that the way to check if a finite state process satisfies this over $\mathcal{N}$ is to ask FDR if $\mathcal{L}_H(P)$, formulated using $CHAOS_H$ as above, is deterministic. However it is not quite as simple as that, since the check can be subverted by the same divergences (resulting from infinite sequences of hidden H actions in P) that mean the definition does not work in $\mathcal{N}$. In fact these divergences can even subvert a determinism check carried out in the stable failures model, since FDR's algorithm to do that does not always work on a divergent process – see [22]. As remarked there, one reliable method for doing this is the pair of checks

- $P \underset{H}{\parallel} STOP$ deterministic (over $\mathcal{N}$)
- $P \underset{H}{\parallel} STOP \sqsubseteq_F (P \underset{H}{\parallel} CHAOS_H) \setminus H$

This pair of checks together imply that $\mathcal{L}_H(P)$ is deterministic, and do not allow an infinite sequence of H actions to cause a problem.

A second reliable method is to replace FDR's built-in check for determinism by a method that can be implemented directly in terms of the tool's refinement checking capabilities, namely to compare two copies of the process P being checked and forcing the second to follow exactly every trace that the first follows. If it never diverges and this always succeeds then P is deterministic, otherwise it is not. Since we will be adapting this idea (originally due to Lazić [10]) later in this paper, we realise it below in a way easily implemented in FDR. Here *clunk* is an event that the process P does not use itself and $E = \Sigma - \{clunk\}$:

$$\begin{aligned}
CReg &= x?E \rightarrow clunk \rightarrow CReg \\
Clunking(P) &= P \underset{E}{\parallel} CReg \\
Test &= x?E \rightarrow x \rightarrow Test \\
RHS(P) &= ((Clunking(P) \underset{\{Clunk\}}{\parallel} Clunking(P)) \underset{E}{\parallel} Test) \setminus \{clunk\} \\
LHS &= STOP \sqcap x?E \rightarrow x \rightarrow LHS
\end{aligned}$$

The use of *clunk* keeps the two copies of P within one event of each other, so that each pair of events come one from each copy. *Test* forces the two to follow the same trace.

The specification LHS allows anything this $RHS(P)$ might do except for one process being unable to follow the other's lead causing deadlock, or P diverging.

$LHS \sqsubseteq_{FD} RHS(P)$ is then true if and only if P is deterministic, and if $\sqsubseteq_{FD}$ is replaced by $\sqsubseteq_F$ we get a test for the failures model version of determinism that is not vulnerable to the issue described above. Therefore applying the above to the $CHAOS_H$ formulation of $\mathcal{L}_H(P)$ gives our second reliable test of lazy independence.

The intuition of the deterministic low-level abstraction implying absence of information seems equally valid in Timed CSP, and indeed any of the checks for it listed above works as least as well in the discrete version implemented in FDR. Indeed the absence of Zeno behaviour implies that the simple formulation using the failures or failures/divergences determinism check is guaranteed to work as hiding high-level events cannot introduce divergent behaviour.

$$(P \underset{H \cup \{tock\}}{\parallel} TCHAOS_H) \setminus H$$

can be tested for determinism where $TCHAOS_H$ is the *tock*-CSP process

$$TCHAOS_H = tock \rightarrow TCHAOS_H \,\Box\, (STOP \sqcap ?x : \Sigma - \{tock\} \rightarrow TCHAOS_H)$$

Note that this process violates the no-Zeno assumption, but that if P satisfies it then so does the construction for lazy abstraction above.

If the lazy abstraction of a Timed CSP process P is deterministic, then this does imply absence of information flow at least as measurable in the model being considered. The fact that no process which ever withdraws an offer is deterministic thanks to NIW represents a major problem for this definition.

Given our analysis in Section 3, we should contemplate modifying our definition so that it is deemed free of information flow if the abstraction is either refinement maximal or quasi-deterministic. Over the continuous model $\mathcal{F}_{\mathbf{T}}$ these are the same thing, but it is as well to ask which if either is in principle the right answer.

If we believe that the model $\mathcal{M}$ we are using records all the observations that Lois might make are the ones recorded in whatever semantic model we are using, then the right answer appears to be "maximally refined". For we know that the observations she can make will be those possible for some process P_H in $\mathcal{M}$, depending on how the high-level user Hugh chooses to behave. Whatever Hugh does will be a refinement of the least refined process he can be. So if Lois's view is already maximally refined for the least refined Hugh, nothing he can do can change her view. This gives us a much stronger guarantee than simply saying that Lois's view is independent of Hugh's behaviour, because it also allows for possible variability in the system P's.

We illustrate this with an example: suppose $LEAK$ is any process that passes information from Hugh to Lois, for example

$$LEAK = hugh?x \rightarrow lois!x \rightarrow LEAK$$

Now suppose that M is any process with alphabet L (implying that $\mathcal{L}_H(M) = M$ such that $M \sqsubseteq \mathcal{L}_H(LEAK)$. Then if $P = M \sqcap LEAK$ then Lois's view of the combination of P and $Hugh$ will be equivalent to M no matter what process with alphabet H we pick for $Hugh$. Nevertheless the system P is allowed to behave like $LEAK$ which is not secure.

In fact, because M never communicates with Hugh, the latter knows that anything he communicates to P will immediately be sent to Lois.[2]

Thus insisting that $\mathcal{L}_H(P)$ is maximally refined shows that neither Hugh's decisions or the ways in which P can behave as a more refined process can affect Lois's view.

All this is, of course, very similar to the justification of the determinism-based definition of noninterference, which is not surprising. Where maximally refined processes are not deterministic this new definition requires the knowledge that whatever nondeterminism that remains cannot be resolved by whatever Hugh does and whatever internal decisions are made in P. In other words, whatever nondeterminism remains in a maximally refined process must remain in the mechanism that Lois observes. With this caveat, we can express the following re-characterisation of noninterference.

Generalised characterisation of noninterference *Suppose we have a semantic model in which refinement coincides with the reduction of all visible nondeterminism that can be eliminated by implementation decisions. Then if the abstraction $\mathcal{A}_H(P)$ characterises how P appears to L in the presence of the most nondeterministic conceivable behaviour in H, we can deem P to be independent of L if $\mathcal{A}_H(P)$ is maximal in the refinement order.*

In the discrete case we have found where it is possible to refine nondeterminism in a quasi-deterministic process, one can construct an example to show that having a discrete quasi-nondeterministic abstraction need not exclude information flow. With *tock*-time unit 1, with a an event in L and h an event in H and b a further event, where both h and b take the same time (say d) to complete, we can define:

$$R = (h \rightarrow P_2 \,\Box\, b \rightarrow P_1) \setminus \{b\}$$

This process is certain to perform either h or the hidden b in the first time step, and if Lois ever sees a refused after the delay d but before a has occurred, then she will know h has occurred. The natural lazy abstraction of R is just $WAIT\ d;\ P_2$, which is quasi-deterministic.

So we have concrete evidence that quasi-determinism of the abstraction over discrete models does not always mean absence of information flow, and a powerful argument that under certain assumptions the refinement maximality of the abstraction does guarantee it. Nevertheless we will find in Section 6 that quasi-determinism over discrete models can be useful nevertheless.

5.1 Abstraction over timed failures

In order to give substance to the specifications of noninterference implied above, we must formulate abstraction over the continuous and discrete timed failures models. We concentrate on lazy abstraction but remark that mixed abstraction poses no problems other than getting the lazy part of it right: high level actions that cannot be delayed by Hugh are still hidden.

We start with $\mathcal{F}_{\mathbf{DT}}$. We want $\mathcal{L}_H(P)$ to represent how P looks to an observer unable to see alphabet H on the assumption that there is a user interacting with P in H with the full capability of offering subsets of events to the process that vary (a) when an event occurs and (b) with time.

[2] If different instances of *Hugh* combined with P produce different answers, this is concrete evidence that information be passed through P, so we can certainly use such comparisons to search for covert channels. It is just that such a comparison cannot easily be justified as a complete test for information flow.

Simply translating the untimed formulation to Timed CSP:

$$(P \underset{H}{\parallel} CHAOS_H^-) \setminus H \quad \text{where}$$
$$CHAOS_H^- = STOP \sqcap ?x : H \to CHAOS_H^-$$

brings a number of problems.

- If some events in H take more than 0 time to complete, $CHAOS_H^-$ defined like this is not as general as it should be since it cannot immediately follow up such an event with another, even though we can imagine Hugh as a parallel process that can. *The natural way to solve this problem is to define $CHAOS_H^-$ in an environment where all events take zero time.*
- If some events in H take 0 time (which they will if we follow the solution above) then the recursion for $CHAOS_H^-$ is not time guarded and the process can perform an infinite number of events in a finite time. *This means that $CHAOS_H^-$ is not a proper Timed CSP process. However it is still reasonable to regard the parallel composition $P \underset{H}{\parallel} CHAOS_H^-$ as one since $CHAOS_H^-$ can perform no more actions than P does. So this is more of an apparent problem than a real one.*
- More subtly, imagine the situation where the $CHAOS_H^-$ process defined above has resolved its nondeterministic choice in the first time step, but no H action occurs before the first *tock*. The operational semantics of CSP give it no way of changing its mind for the second time step. At the level of timed failures, the semantics of this $CHAOS_H^-$ does not contain the behaviour $\langle H, tock, h\rangle^\frown\Delta$ for $h \in H$. It is because this definition is deficient in this way that we have given it the superscript $-$.
  An efficient definition that does work (still subject to the assumption that events in it take zero time, and that it must be put in parallel with a non-Zeno process) is

  $$CHAOS_H^A = (?x : H \to CHAOS_H^A) \rhd (WAIT\,1;\ CHAOS_H^A)$$

  where $\rhd$ is the asymmetric choice operator that initially offers the choice of the events of its left-hand argument with a τ that takes it to its right-hand argument. ($P \rhd Q$ is equivalent to $(P \,\Box\, a \to Q) \setminus \{a\}$ for an event a not appearing in either P or A.) Thus $CHAOS_H^A$ can perform any sequence of H events in a given time unit but may at any time opt not to perform any more before the next *tock*. This version never offers events from H in a stable state, which could be problematic in some contexts, but will not be when events from H are hidden as they are in abstraction: that explains the superscript A (for abstraction).

So our definition of lazy abstraction over $\mathcal{F}_{\mathbf{DT}}$ will be

$$\mathcal{L}_H(P) = (P \underset{H}{\parallel} CHAOS_H^A) \setminus H$$

which gives us our first concrete timed definition of noninterference.

It is also possible to use the *tock*-CSP process *TCHAOS* defined earlier, which is also able to change its decisions about whether or not to offer H events each *tock*.

Definition 3. *A process defined in integer Timed CSP is* 1-independent of H *if $\mathcal{L}_H(P)$ (defined as above) is maximally refined in $\mathcal{F}_{\mathbf{DT}}$.*

The reason for the 1 in this name will become apparent in Section 6. In examining examples of timed noninterference we will largely restrict ourselves to examples which satisfy *untimed* noninterference, such as

$$P = l \rightarrow l \rightarrow P \,\Box\, h \rightarrow LS \quad \text{where}$$
$$LS = l \rightarrow LS$$

This, seemingly, just offers the event $l \in L$ for ever, possibly interrupted by a single $h \in H$ after an even number of ls. Since l is always on offer this satisfies the untimed definition of noninterference in the usual direction, but note that L can pass information to H by choosing an odd or even number of ls. So over untimed CSP, $\mathcal{L}_H(P)$ is deterministic and $\mathcal{L}_L(P)$ is nondeterministic.

For the definition of this P to be time guarded, we need l to take non-zero time to complete. However $\mathcal{L}_H(P)$ is only maximally refined over $\mathcal{F}_{\mathbf{DT}}$ if h takes zero time, for otherwise there is a period after h when l is refused in a way in which it would not have been it h had not happened. So for example the abstraction will have the behaviours $\langle l \rangle^\frown \Delta$ and $\langle \{l\}, tock \rangle^\frown \Delta$ if h take more than 0 time to complete. That would not be compatible with being maximally refined. On the other hand, if h does take time 0 (so that P is willing to communicate l immediately after h), the abstraction is equivalent to LS.

This example teaches us an expected lesson: *if H and L share access to a sequentially defined process P, considerable care is necessary to eliminate all timing channels from H to L.*

It is also interesting to note that if h takes one time unit then $\mathcal{L}_H(P)$ is quasi-deterministic even though non-maximal, but that if either h takes at least two units, or we use the model $\mathcal{F}_{\mathbf{DT}}$ with the *tock* unit 0.5, then it is not quasi-deterministic.

We will see further examples of timed noninterference analysis later.

The problem with defining abstraction over the continuous model $\mathcal{F}_{\mathbf{T}}$ is that time moves forward continuously rather than discretely. The process $CHAOS_H^A$ depends crucially on there being a *next* time at which things happen. The best way of defining lazy abstraction over $\mathcal{F}_{\mathbf{T}}$ is as a primitive operator over this model:

$$\mathcal{L}_H(P) = \{(s \setminus H, \aleph \cup \aleph') \mid (s, \aleph) \in P \wedge \aleph' \supseteq H \times [0, \infty)\}$$

In other words P is allowed to perform any behaviour at all, but any offers in H it makes are not visible to the outside world. The assumption here is that at times when $\aleph$ does not contain the whole of H, the abstracted copy of H is refusing any such events that P offers. It is interesting to contrast this with the definition of hiding which insists that P is always forced to perform as many H events as it can – namely when time progresses the whole of H is hidden:

$$P \setminus H = \{(s \setminus H, \aleph) \mid (s, \aleph \cup H \times [0, \infty)) \in P\}$$

It should not be too hard to see that our discrete time definition of $\mathcal{L}_H(P)$ using $CHAOS_H^A$ can be re-written in a form similar to the continuous one above. There is a strong reason to code the discrete definition in the Timed CSP language that does not apply in the continuous case, namely that the discrete model has been implemented in FDR.

Definition 4. *A process defined in (general) Timed CSP is lazily independent of H over $\mathcal{F}_{\mathbf{T}}$ if $\mathcal{L}_H(P)$ (defined as above) is maximally refined, or equivalently quasi-deterministic in $\mathcal{F}_{\mathbf{T}}$.*

This continuous definition gives the same result for the simple process P that we studied in the discrete case above.

An obvious question we can ask at this stage is whether the two definitions coincide for integer Timed CSP. Unfortunately the answer to this is "no", with the problem arising because of the distinction between maximally refined and quasi-deterministic processes over $\mathcal{F}_{\mathbf{DT}}$.

It is clearly the case that any process actually constructed as the parallel composition of two processes P_L and P_H with alphabets respectively L and H, not communicating at all, is unable to pass information from H to L or vice versa: $P = P_L \;|||\; P_H$. Both our definitions of lazy abstraction give $\mathcal{L}_H(P) = P_L$ (as is also the case in untimed CSP), and so the question of whether our definitions of noninterference are satisfied by such a P comes down to whether P_L is maximally refined when interpreted in the discrete and continuous models respectively. If our two definitions of noninterference coincided, then they would have to agree on this question also.

Assume both $l1$ and $l2$ are low level events that take time 0 to complete.

$$P_L = (Q_1 \;|||\; Q_2) \underset{\{l2\}}{\parallel} (l2 \rightarrow STOP) \quad \text{where}$$
$$Q_1 = (l1 \rightarrow ((l2 \rightarrow STOP) \,\Box\, WAIT\,1);\; STOP)$$
$$Q_2 = WAIT\,4;\; l2 \rightarrow STOP$$

P_L offers $l2$ for one time unit after $l1$ occurs, and the offer is then withdrawn. However a second and indistinguishable offer of $l2$ is always made at time 4 unless the other one has been taken up first. Note that the parallel composition with $l2 \rightarrow STOP$ ensures that only one $l2$ can occur in total, meaning that if both $l2$s are available at the same time, the nondeterminism over which occurs has no visible consequences.

Over the continuous model this process is quasi-deterministic and hence maximal: the offer of $l2$ is withdrawn if $l1$ occurs early enough and later reappears thanks to Q_2. Over the discrete model it is still quasi-deterministic but not maximal in the case where the gap between the end of the first offer and time 4 is one *tock*. Decreasing the interval between the *tock*s to 0.5 (or any other reciprocal) does not help here as it did in an earlier example, because the event $l1$ can always happen at the time that will leave the gap at one *tock*.

So for this example no time interval for *tock*s that makes the *WAIT*s in the program an integer multiple of it will make this P_L maximal.

If we regard the continuous semantics as definitive and the discrete ones as approximations, it is comforting to note that the discrete model of noninterference differs only in the direction of being more conservative. We believe that this is always the case for programs where the discrete models are applicable.

6 Digitisation: playing with time

The theory of digitisation was introduced by Henzinger, Manna and Pnueli in [7] as a way of proving properties about continuous systems by analysing discrete approximations. It was adapted for Timed CSP by Ouaknine [14, 15] who showed that one can prove certain properties of systems over the continuous model $\mathcal{F}_{\mathbf{T}}$ by demonstrating analogous properties of discrete approximations. In particular he showed that every integer Timed CSP program has the property of being *closed under digitisation* and therefore refines any specification that is *closed under inverse digitisation* (certain, but not all integer Timed CSP programs) if any only if the refinement holds over $\mathcal{F}_{\mathbf{DT}}$.

In this section we will examine how properties such as quasi-determinism and noninterference behave under digitisation. Our objective will be to find a way of verifying that an integer Timed CSP program is lazily independent of H by analysis over $\mathcal{F}_{\mathbf{DT}}$.

In order to make the following analysis easier we will assume that the only sources of time delays in our programs are integer *WAIT* statements: one can recode delays that occur when events happen or recursion unfolds into this form if required.

We characterise digitisation as the application of certain sorts of transformations that change the times of the actions (visible, invisible and evolutions through time) that occur in process's execution in standardising way but which preserve the validity of the execution. They are formalised in terms of the operational semantics of Timed CSP defined by Schneider in [29].

Before introducing digitisation we consider more general *retimings* of operational semantics: monotonic (but not necessarily strictly so) mappings from $\mathbb{R}^+$ to itself intended to preserve the validity of operational semantics when applied to the beginning and end of all times of operational semantic transitions. Such transitions are visible and visible actions, which are instantaneous, and timed evolutions such as $P \overset{t}{\rightsquigarrow} Q$ whose end is t after its beginning.

One cannot arbitrarily re-time such behaviours because of maximal progress: an event that happens at the time a τ action becomes available can only be re-timed to the moment the corresponding τ becomes available in the transformed behaviour. A retiming is *valid* if this it true. Given our assumption that *WAIT*s are the only source of delays, the only way in which a τ (or any other action) can become available through a time evolution is when a *WAIT* expires somewhere within the program.

We restrict our attention to *integer periodic* retimings ϕ which map each integer time to itself, and which map each $n + x$ (for $0 < x < 1$) to $n + \phi(x)$ (with $\phi(x)$ necessarily being in the closed interval $[0, 1]$. The fact that we are considering only *integer* Timed CSP means that if we retime the start of a *WAIT* n from $k + x$ to $k + \phi(x)$ then the end of it is retimed from $k + x + n$ to $k + \phi(x) + n$, which is of course the time that our *WAIT* n ends when its start is retimed. In general a Timed CSP term that does not immediately have a τ can evolve through any time up to and including the first moment a *WAIT* statement within it elapses. So one can prove via a combination of structural induction on a term with mathematical induction on the number of actions and time evolutions that have occurred that

- After k steps the original and retimed programs are in the same state except for the exact times remaining on non-zero *WAIT*s. If the remaining time is zero on a *WAIT* in the original, then it also is in the retimed one. A non-zero time remaining on a *WAIT* in the original program can become zero in the retimed one when the original program's time and the time when that wait expire map to the same value. In this case the original behaviour certainly has an action between these two times, so the retimed one has an action *at* the same (retimed) moment.
- The original and retimed programs have exactly the same set of actions available except where the retimed version has a *WAIT* retimed to 0 as discussed above, in which case the retimed one has an additional τ. Time evolutions are available up to and including the minimum remaining time on any *WAIT* each of the original and retimed states respectively contain.

– In any case (i) any action that the original performs now is valid in the retimed state (ii) if the retimed process is obliged by maximal progress to perform an action now then the retimed behaviour contains an action at the same time.
– If the present time is t and the original and retimed programs have a *WAIT* that elapses respectively in times x and y, then $\phi(t+x) = t+y$.

We can conclude:

Theorem 3. *1. An integer periodic retiming ϕ is valid on an integer Timed CSP program P.*
2. If $(s, \aleph)$ is in the $\mathcal{F}_{\mathbf{T}}$ representation of P, then so is $(\phi(s), \phi(\aleph))$, where ϕ acts on the times of events and the end points of the half-open intervals during which $\aleph$ is constant. (Note that if $\aleph$ includes some $X \times [x, y)$ where $\phi(x) = \phi(y)$ then $\phi(\aleph)$ retains no "memory" of this since $[\phi(x), \phi(y))$ is then empty.)

We definite a *digitisation* to be an integer periodic retiming whose image in any interval $[n, n+1]$ is finite. In other words, a digitisation transforms all of the actions in a behaviour to ones that happen at members of a pre-determined discrete set of times.

Ouaknine, in developing the above ideas, concentrates on digitisations that map every time t to the integer above or below it, and specifically $[t]_\epsilon$ for $0 < \epsilon \leq 1$ that maps $n + x$ to n or $n+1$ depending on whether $x < \epsilon$ or $x \geq \epsilon$. For our purposes we need a little more flexibility. Identify $[t]_\epsilon$ with $[t]_{\langle\epsilon\rangle}$ and allow the subscript, in general, to be any finite, nonempty and strictly monotonic sequence of numbers in the range $(0, 1]$.

Definition 5. *$[t]_{\langle\epsilon(1),\ldots,\epsilon(n)\rangle}$ is the retiming that maps $r + x$ ($r \in \mathbb{N}$, $0 \leq x < 1$) to r if $x < \epsilon(1)$, to $r+1$ if $x \geq \epsilon(n)$ and to $r + \frac{m}{n}$ if $\epsilon(m) \leq x < \epsilon(m+1)$ for $1 \leq m < n$.*

Below, *frac*(x) is defined to be the unique number $0 < y \leq 1$ such that $x - y$ is an integer. So in particular *frac*$(n) = 1$ for $n \in \mathbb{N}$.

Lemma 2. *If $0 \leq t_1 < t_2 < t_3$ then we can choose $0 \leq \epsilon 1 < \epsilon 2 \leq 1$ such that $[\cdot]_{\langle\epsilon 1,\epsilon 2\rangle}$ maps t_1, t_2, t_3 to distinct values, necessarily separated by at least 0.5.*

Proof If $t_3 - t_1 > 1$ then this can be done with a single ϵ, so we will concentrate on the case of $t_3 - t_1 \leq 1$.

(i) If $n \leq t_1 < t_2 < t_3 \leq n+1$ then put $\epsilon 1 =$ *frac*(t_2) and $\epsilon 2 =$ *frac*(t_3).
(ii) If $t_1 < n \leq t_2 < t_3$ then set $\epsilon 1 =$ *frac*(t_3) and $\epsilon 2 =$ *frac*(t_1).
(iii) If $t_1 < t_2 < n < t_3$ then set $\epsilon 1 =$ *frac*(t_1) and $\epsilon 2 =$ *frac*(t_2).

It seems clear that the above could be generalised to the case of $n+1$ distinct points being mapped by a suitable retiming $[\cdot]_s$ to points at least $1/n$ apart.

Ouaknine establishes a crucial connection between the discrete and continuous semantics of integer Timed CSP. It is easy to see a relationship between $\mathcal{F}_{\mathbf{DT}}$ behaviours and integer $\mathcal{F}_{\mathbf{T}}$ behaviours – ones where everything (i.e. events and changes in $\aleph$) happens at an integer time – given one of the former s we map it to $\psi(s) = (u, \aleph)$, where u consists of the sequence of all non-*tock* events in s, each given as its time the number of *tock*s that precede it in s, and, for the unit of time up to time k $\aleph$ refuses those events refused prior to the kth *tock* in s.

Theorem 4. *[14] For any integer Timed CSP process P, the integer behaviours in its $\mathcal{F}_{\mathbf{T}}$ semantics are exactly $\psi(s)$ as s ranges over its $\mathcal{F}_{\mathbf{DT}}$ semantics.*

It is clear that the continuous time semantics of any Timed CSP program P are isomorphic to those of kP for k an integer, which is the same program except that all $WAIT\, n$ are transformed to $WAIT\, (kn)$ provided we scale all behaviours of kP by dividing all the times by k. Since kP is an integer Timed CSP program if P is, we can deduce the following lemma. Here a *half integer* $\mathcal{F}_{\mathbf{T}}$ behaviour is one where all events and changes in $\aleph$ occur either at integers or $n + \frac{1}{2}$ for an integer n.

Lemma 3. *For any integer Timed CSP process P, the half-integer behaviours in its $\mathcal{F}_{\mathbf{T}}$ semantics are exactly the scalings by $\frac{1}{2}$ of $\psi(s)$ as s ranges over the $\mathcal{F}_{\mathbf{DT}}$ semantics of $2P$.*

This is exactly the result we need to create a decision procedure for noninterference defined over $\mathcal{F}_{\mathbf{T}}$.

Theorem 5. *An integer Timed CSP process P is lazily independent of H (judged over $\mathcal{F}_{\mathbf{T}}$) if and only if $\mathcal{L}_H(2P)$ is quasi deterministic when judged over $\mathcal{F}_{\mathbf{DT}}$ (or equivalently if $\mathcal{L}_H(P)$ is quasi-deterministic when judged over $\mathcal{F}_{\mathbf{DT}}$ in which the length of one tock is 0.5).*

Proof If $\mathcal{L}_H(2P)$ is not quasi-deterministic then one of conditions (c) and (d) from Lemma 1 (ii) applies. From this, and the definition of $\mathcal{L}_H$, we get two cases:

- If (c) (the case where the behaviour visible to L prior to the nondeterminism does not end in *tock*) then for some L event l, $2P$ has behaviours over $\mathcal{F}_{\mathbf{DT}}$ of the forms $s_1{}^\frown\langle l\rangle{}^\frown\Delta$ and $s_2{}^\frown\langle\{l\}, tock\rangle{}^\frown\Delta$ where deleting the H events in s_1 and s_2 leaves the behaviour s. Without loss of generality we assume that all the pre-*tock* refusals in these are $\emptyset$. Theorem 4 then tells us that $2P$ has the $\mathcal{F}_{\mathbf{T}}$ behaviours $(\psi(s_1)_1, \{l\} \times [n, n+1))$ and $(\psi(s_2)_1{}^\frown\langle(l, n)\rangle, \emptyset)$ where there are n tocks in each of s, s_1 and s_2 and $\psi(s)_1$ takes the non-*tock* events in s and makes a timed trace by attaching the number of *tock*s preceding each as its time.
  Here ψ is the map defined earlier from $\mathcal{F}_{\mathbf{DT}}$ behaviours to integer $\mathcal{F}_{\mathbf{T}}$ ones, so $\psi(s)_1$ extracts just the timed trace from this.
  The definition of $\mathcal{L}_H$ over $\mathcal{F}_{\mathbf{T}}$ then tells us that $\mathcal{L}_H(2P)$ has the behaviours $(\psi(s)_1, \{l\} \times [0, 1))$ and $(\psi(s)_1{}^\frown\langle(l, n)\rangle, \emptyset)$, meaning it is not quasi-deterministic since it is easy to see that $end(\psi(s)_1) = n$ by construction and therefore case (a) of Lemma 1 (i) applies to the continuous semantics of $\mathcal{L}_H(2P)$.
- The second case is where (d) applies. A very similar argument then shows that (b) applies to the continuous semantics of $\mathcal{L}_H(2P)$.

We can deduce that $\mathcal{L}_H(2P)$, and therefore $\mathcal{L}_H(P)$, is not quasi-deterministic.

Note that the multiplier 2 was not necessary for this direction of the implication. It is, however, necessary for showing that if the continuous behaviour is not quasi-deterministic then so also is the discrete.

Assuming that $\mathcal{L}_H(P)$ is not quasi-deterministic over $\mathcal{F}_{\mathbf{T}}$ gives us the two options of Lemma 1 (i), of which the second is more interesting.

- In case (b), we can assume that δ has been chosen sufficiently small so that the in the two behaviours $(s_1{}^\frown(l, t), \emptyset)$ and $(s_2, \{l\} \times [t-\delta, t+\delta))$ that P must have for some $l \in L$ with $s_1 \setminus H = s_2 \setminus H$ and $end(s_1 \setminus H) < t$, no event of s_1 or s_2 occurs in $[t-\delta, t+\delta)$ except at t. We can also assume that $\delta \leq \frac{1}{2}$.
  We can now invoke Lemma 2 with $t_1 = t - \delta$, $t_2 = t$ and $t_3 = t + \delta$ to get a valid digitisation of integer Timed CSP that maps these three times to three consecutive

(thanks to $\delta \leq \frac{1}{2}$) members of the series $\langle \frac{n}{2} \mid n \in \mathbb{N} \rangle$. Applying this to $(s_1 \hat{} \langle (l,t) \rangle, \emptyset)$ and $(s_2, \{l\} \times [t-\delta, t+\delta))$ and then scaling by 2 tells us that $2P$ has behaviours $(s_1' \hat{} \langle (l,t') \rangle, \emptyset)$ and $(s_2', \{l\} \times [t'-1, t'+1))$ where all events occur at integer times, t' is an integer, and $s_1' \setminus H = s_2' \setminus H$. Theorem 4 and the definition of lazy abstraction over $\mathcal{F}_{\mathbf{DT}}$ then tells us that $\mathcal{L}_H(2P)$ is not quasi-deterministic over $\mathcal{F}_{\mathbf{DT}}$. [Note that both s_1' and s_2' can have some events happening at the image of t under the digitisation.]

– The case where (a) applies is simpler than the above because we can consider an interval $[t, t+\delta)$ rather than $[t-\delta, t+\delta)$, and so only two points are involved in the digitisation, meaning that we do not need the factor of 2.

This completes the proof of Theorem 5. ∎

This result is very powerful in the context of noninterference since it tells us that a particular discrete model, which observes an integer Timed CSP process only at discrete times, is sufficient to prove that there is no information flow to an observer who can observe it at any and all times.

It suggests the following discrete definition of noninterference:

Definition 6. *For $k \in \mathbb{N} - \{0,1\}$, we define the integer Timed CSP P to be k-lazily independent of H if kP is quasi-deterministic when judged in $\mathcal{F}_{\mathbf{DT}}$.*

Observing that the multiplier 2 in the formulation and proof of Theorem 5 could have been replaced by any integer $k \geq 3$, we get the following result as a corollary.

Theorem 6. *For integer Timed CSP, the conditions k-lazy independence are all equivalent for $k \geq 2$.*

The example given in Section 5.1 of a process P_L which is quasi-deterministic over all models, but not maximal in any discrete one (which is equivalent to kP_L not being maximal in DTF for any $k > 0$) tells us something rather unexpected about the discrete characterisations of noninterference. This is that processes (such as $P_H \;|||\; P_H$ for any P_H that only communicates in H) can be k-independent ($k > 1$) without $\mathcal{L}_H(kP)$ being maximal in DTF. This seems at odds with the analysis given earlier that non-maximality leads to information flow. This is not in fact an issue because $\mathcal{L}_H(kP)$ is maximal amongst the images of kP' as P' varies over integer Timed CSP processes.

The reason for the doubling of the "metronome" in the discrete approximation used to decide quasi-determinism derives from needing a witness to the three distinct times that exemplify one sort of failure of quasi-determinism. It it worth noting that if we restrict ourselves to processes that never withdraw offers (so quasi-determinism and determinism are the same) then it is not necessary to use the doubling, because we can decide whether or not the continuous semantics of an integer Timed CSP process are deterministic using the natural rather than doubled discrete model. This is because the failure of a continuous process being deterministic shows up (after the timed trace s) in *two* times $end(s) \leq t_1 < t2$ where both $s \hat{} \langle (a, t_1) \rangle \in traces(P)$ and $(s, \{a\} \times [T_1, t_2))$ is a timed refusal. This means that it is not necessary to use the extended form of digitisation we used above, and in fact we get the following.

Theorem 7. *For an integer Timed CSP process P:*

(a) P is deterministic in the continuous semantics if and only if it is deterministic in the ordinary discrete semantics.

(b) $\mathcal{L}_H(P)$ is deterministic over $\mathcal{F}_{\mathbf{T}}$ if and only if $\mathcal{L}_H$ is over $\mathcal{F}_{\mathbf{DT}}$.

Here (b) is not a trivial consequence of (a) because of the different definitions of lazy abstraction over the two models. However a simplified form of the argument used for Theorem 5 in which we pay attention to when all actions in P occur (τ, H and L) works.

7 Deciding noninterference using FDR

It is possible to test for both quasi-determinism and refinement maximality over $\mathcal{F}_{\mathbf{DT}}$, using FDR. We deal first with quasi-determinism, developing a variant of Lazić's test for determinism. As in that, we compare two copies of a process, this time checking that for every visible action a the first copy of a process P performs, a second one *either* cannot refuse it after the same trace *or* a occurred immediately after a *tock* and the second copy was unable to refuse a prior to the corresponding *tock*.

We therefore have to keep two copies of P running – say P_1 and P_2 where P_1 performs actions and P_2 has to prove for each one that it cannot refuse them appropriately. From the description above it is clear that P_2 has to follow the same trace – including *tock*s – as P_1 but that it has sometimes to be at an earlier time than it. Namely, when P_1 performs an event after *tock*, we have to test whether P_2 can refuse it before, and if so after exactly the same *tock*. Therefore P_1 and P_2 are not synchronised on *tock*: in fact the latter can be 0, 1, and sometimes 2 tocks behind P_1.

We can immediately deduce that the check we devise to check for this is not going to be constructed in Timed CSP, but rather in *tock*-CSP with multiple *tock* events. As with the check for determinism that we are adapting, our check will take the form:

$$LHS \sqsubseteq_F RHS(P)$$

where $RHS(P)$ this time consists of two copies renamed so that they give different names to *tock*, and where they strictly alternate their non-tock events. The two copies are put in a testing harness, and because of the nature of the latter we give the "follower" copy of P two separate names for *tock*. Overall

$$\begin{aligned}
RHS(P) &= \Pi((first(P) \underset{\{turn1,turn2\}}{\parallel} second(P)) \setminus \{turn1, turn2\}) \underset{\Sigma}{\parallel} QDTest \\
first(P) &= P[\![^{tock1}/_{tock}]\!] \underset{E}{\parallel} FReg \\
second(P) &= P[\![^{tock2,\, tock2'}/_{tock,\, tock}]\!] \underset{E}{\parallel} SReg \\
FReg &= ?x : E \rightarrow turn2 \rightarrow turn1 \rightarrow FReg \\
SReg &= turn2 \rightarrow ?x : E \rightarrow turn1 \rightarrow SReg
\end{aligned}$$

where *QDTest* is a testing process and Π a priority operator that we will describe below. The synchronisation with and between *FReg* and *SReg* ensures that the two copies strictly alternate non-*tock* events, with $first(P)$ first.

The testing process has the following states:

$$\begin{aligned} QDTest &= ?x : E \rightarrow x \rightarrow QDTest \\ &\quad \sqcap tock1 \rightarrow QDTest' \\ &\quad \sqcap STOP \\ QDTest'(x) &= tock1 \rightarrow tock2 \rightarrow QDTest' \\ &\quad \sqcap ?x : E \rightarrow QDTest''(x) \\ &\quad \sqcap STOP \\ QDTest''(x) &= (x \rightarrow STOP \,\Box\, tock2' \rightarrow x \rightarrow QDTest) \\ &\quad \sqcap tock2 \rightarrow x \rightarrow QDTest \end{aligned}$$

These are explained:

- The tester is in state $QDTest$ when $first(P)$ and $second(P)$ have performed equal numbers of $tock$s, and their non-$tock$ traces are equal. Necessarily this is when the most recent communication of $first(P)$ was not $tock1$, because the tester deliberately then holds $second(P)$ back.
- $QDTest'$ is when the two have performed equivalent traces except that $first(P)$ has moved ahead by one one $tock1$, which was the most recent event it has performed. If $first(P)$ performs $tock1$ then $second(P)$ must perform (as it will certainly be able to) $tock2$ so it is still one behind. If $first(P)$ performs any other event x then we must check that $second(P)$ obeys the refusal requirements on it, and moves to $QDTest''(x)$ to check this.
  Neither of theses first two states insists that $first(P)$ is able to do anything – hence the inclusion of $STOP$ in the nondeterministic choice.
- $QDTest''(x)$ is when $second(P)$ has fallen behind $first(P)$ by $tock$ followed by x. It now has to do two things: check that $second(P)$ offers (i.e. cannot refuse) x either before or after its next $tock$, and also ensure that the tester is in the right state to check P's behaviour on longer traces. The latter happens automatically in other states, but requires more care here.
  For the first of these tasks, it must create an error state when $second(P)$ can refuse x both before and after P's *tock on the same execution*. This is not possible using a non-prioritised check over $\mathcal{F}$. It could have been done with the refusal testing model $\mathcal{RT}$, but in this presentation we use priority.
  The left-hand branch of the $\sqcap$ in $QDTest''(x)$ is responsible for this part of the check. It initially offers either x or $tock2'$ to $second(P)$, overall we apply the priority operator Π to the system, which is defined to give $tock2'$ lower priority than every other action, all others being equivalent. That means that $tock2'$ can only happen when x is refused by $second(P)$.
  If x does occur in that state then $second(x)$ has passed the test created when $first(P)$ performed x. However the two copies of P have performed different traces ($tock$ and x in opposite orders) and so are permitted to behave differently. We therefore do not carry on with the test from this point on: hence this trace leads to $STOP$ in $QDTest''(x)$ (and the LHS process we define below).
  If $tock2'$ has occurred then $second(x)$ fails the test unless it accepts x in its post-$tock2'$ state. However if it does perform the x we can carry on the check because the two P's have now performed the same trace.
  If we were simply to have done the above, then some future behaviours of P would not get tested. These are the ones where x is guaranteed to be available in $second(P)$ before

$tock2'$, for the priority relation then means that $second(P)$ never performs it after. This problem is solved by the second branch of $QDTest''(x)$'s $\sqcap$. That offers just $tock2$ rather than the choice of $tock2'$ and x, which brings $second(P)$ into a state where

- If is not *obliged* to be able to perform x.
- However we know that in at least one execution it can perform x after $tock2$, since after all the other copy of P has already performed x on the same trace.

So we can carry out the continuing test on this branch, confident in the knowledge that P's behaviour after all possible traces will now be explored.

The reason why we have used $tock2$ and $tock2'$ is so the specification can tell which of the two testing branches has been followed. If $tock2'$ has occurred it will insist that x is offered while after $tock2$ it will not.

It is interesting to note that we have carried out two separate tests on $second(P)$ by using the $\sqcap$ operator between processes that perform them.

The specification against which this is checked in $\mathcal{F}$ is then

$$
\begin{aligned}
LHS &= STOP \sqcap (?x : E \rightarrow x \rightarrow LHS) \sqcap (tock1 \rightarrow LHS') \\
LHS' &= STOP \sqcap (tock1 \rightarrow tock2 \rightarrow LHS') \sqcap (?x : E \rightarrow LHS''(x)) \\
LHS''(x) &= (x \rightarrow STOP) \sqcap (tock2' \rightarrow x \rightarrow LHS) \sqcap (tock2 \rightarrow (STOP \sqcap x \rightarrow LHS))
\end{aligned}
$$

Here, the states correspond in an obvious way to the states of $QDTest$. This specification is guaranteed to be trace refined by RHS since $QDTest$ refines it, so the only way it can fail is when the required offers are not made. These correspond to the various failures of quasi-determinism discussed above.

So we may test an integer Timed CSP process P for independence by running the check

$$LHS \sqsubseteq_F RHS(pri((CHAOS^A_H \underset{H}{\parallel} 2P) \setminus H)$$

where $2P$ and $CHAOS^A_H$ are as described above. Here pri is the time-priority function that represents the boundary between Timed CSP (inside it) and $tock$-CSP (outside it).

Pragmatics

Running two copies of an implementation process in parallel like this to check for noninterference is potentially expensive since it means that in the worst case the state space of $RHS(P)$ is quadratic in that of P. We can reduce this problem by using either or both of the following techniques.

- We can use an FDR compression operator on P before applying RHS to it. Because the definition of RHS involves priority, this must be a compression that is valid inside the FDR *prioritise* operator: at the time of writing, by far the best option is (divergence-respecting) weak bisimulation *wbisim* as described in [24]. *wbisim* is a recent addition to FDR.
- Following the first of the two alternatives presented earlier for reliably checking $\mathcal{L}_H(P)$'s determinism over untimed models, we can break the check for the abstraction's quasi-determinism over $\mathcal{F}_{\mathbf{DT}}$ into the same two parts:
  (a) $P \underset{H}{\parallel} STOP$ is quasi-deterministic
  (b) $P \underset{H}{\parallel} STOP \sqsubseteq \mathcal{L}_H(P)$ where refinement is judged over $\mathcal{F}_{\mathbf{DT}}$.

This is attractive because in the absence of any H actions, the process being checked here for quasi-determinism is potentially significantly smaller than $\mathcal{L}_H(P)$, which is used only by itself in (b).

In the case studies below, we will find that making (a) above true is sometimes challenging. In that case (b) alone makes (as discussed earlier) a useful but incomplete check for noninterference.

8 Case studies

In this section we present two related case studies, one of a sequential process and one of a parallel system. Both are intended to multiplex high and low level communications through a common medium. They indicate potential sources of timing channels in shared systems as well as possible cures. Throughout this section we assume that events take unit time to complete: $et(a) = 1$ for all $a \in \Sigma$.

8.1 Sequential shared medium

In both our examples we will imagine (as in the corresponding untimed examples in Section 12.4 of [22]), that *Hugh* is sending messages to *Henry* and *Lois* is sending them to *Leah*

The following is a simple sequential process that communicates such messages:

$$\begin{aligned} TM(C) = {} & send?x : (S - \{s \mid s \in C\})?m \rightarrow TM(C \cup \{(x,m)\}) \\ & \Box \, (\Box_{(x,m)\in C} \, rec!dual(x)!m \rightarrow TM(C - \{(x,m)\}) \end{aligned}$$

where *Leah* and *Lois*, and *Hugh* and *Henry* are two pairs of duals and the initial value of C is just $\emptyset$. C contains the data presently in the medium. As untimed CSP the above definition would be equivalent to the interleaving of two separate one-place buffers, one at each level, and would satisfy every conceivable independence condition between H and L.

This is not true for Timed CSP unless the delays attached to H events was 0 (meaning that the process would be capable of Zeno behaviour). In reality the sharing of a resource like this is always likely to create a *timing channel*: the fact that *Hugh* is sending *Henry* a message will delay *Lois* from sending one to *Leah* if that takes any time at all. This is a simple example of the most common sort of timing chennel: L sees the timing effects of H's consumption of system resources.

Our timed noninterference check easily identifies this problem. Note that checking it with all event times set to 2 is equivalent to checking $2\,TM(\emptyset)$ because there are no other sources of delay in this program.

One way of preventing information flow though systems is partitioning whatever resource gives rise to a potential channel between the levels. We can do this in the present example by only permitting high and low events at disjoint sets of times in such a way that whenever a high-level event occurs its delaying effect is over by the time that L might again perform an event.

With a process like TM we can imagine either redesigning it to a sequential process with the above quality, or modelling an operating system component that schedules access to it. We can realise the latter by building a scheduler that is placed in parallel with our system, synchronising on all events, which in different phases makes and withdraws offers in L and H. Assume there are constants LO, LB, HO, HB representing the length of the

offer of low events, the break after this before high events are allowed, and the same two for high. Then we can create our scheduler:

$$\begin{aligned} LOW = (?x : L \to STOP) \triangle WAIT\ LO);\ & WAIT\ LB; \\ ((\mu\, p.?x : H \to p) \triangle WAIT\ HO);\ & WAIT\ HB;\ LOW \end{aligned}$$

Putting this in parallel with TM creates a process lazily independent of H provided that $HB \geq 1$.

Note that we have allowed an arbitrary number of H-actions, and only one L-action, per time slot. That is because allowing multiple L actions creates nondeterminism itself by interaction with the scheduler: when some action is just becoming available when the time-out fires, it may or map not be offered. One can handle this in one of two ways:

- If we were to replace $?x : L \to STOP$ by $\mu\, p.?x : L \to p$ above the system would satisfy the weaker noninterference specification

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

  but this is not an absolute guarantee of independence.
- We can allow multiple phases of L actions for each of H.

Both of these are illustrated in the file that implements this case study.

8.2 Parallel implementation

There are various proposals in Section 12.4 of [22] for how to solve the same shared channel problem in an untimed context using a network where, in addition to the channel itself, there are processes which act as intermediaries between it and each of the four users. Some of these failed and some succeeded. One which succeeded was using flow control to ensure that the central medium never gets blocked: *Leah*'s and *Hugh*'s terminal processes do not accept a second input until they receive an acknowledgement that the first has been delivered.

$$\begin{aligned} TLois &= send.lois?x \to in.lois!x \to out.lois.Ack \to TLois \\ THugh &= send.hugh?x \to in.hugh!x \to out.hugh.Ack \to THugh \\ TLeah &= out.leah?x : T \to rec.leah!x \to in.leah.Ack \to TLeah \\ THenry &= out.henry?x : T \to rec.henry!x \to in.henry.Ack \to RHenry \\ Medium &= in?s?x \to out!dual(s)!x \to Medium \end{aligned}$$

This satisfies any reasonable untimed noninterference condition, but interpreted as Timed CSP it does not satisfy our timed ones for much the same reasons as above.

There are at least two ways one might set about putting this right:

(i) The whole ought to satisfy non-interference if we replace the *Medium* process above with a process that satisfies timed independence.
(ii) We could set out to find a solution which addresses the timing of the system as a whole.

The first of these represents sound design, and can reasonably be argued whenever (as in our case) the processes interacting with a non-interfering core are non-interacting (i.e. separated) parallel processes.

This can be realised by replacing *Medium* with the process derived in the previous section, and provided the system is actually constructed in this way this would lead to a secure system. It does not (at least composed in the way we did) lead to a system which satisfies the timed lazy independence property, because (like one plausible solution we discussed in the previous section) it fails to be quasi-deterministic even when H does nothing at all. It did, however, satisfy the weak noninterference condition $P \underset{H}{\|} STOP \sqsubseteq \mathcal{L}_H(P)$ as the earlier case.

The solution we found that looks at overall system design was based on the fact that we need the time it takes to transport a message from *Lois* to *Leah* to be deterministic and independent of H activity, and similarly the time from delivery at *Leah* until *Lois* next being able to send another. One way of achieving this is to allow the low processes in effect, the right to book a time at which their message will be delivered and ensure that the medium is not used by anything else at that time.

An easy way of enabling this is to modify the medium so that it accepts every message twice and only delivers on the second occasion.

$$BM = in?x?m \rightarrow in!x!m \rightarrow out!x!m \rightarrow BM$$

The assumption here is that we will ensure that the second input by BM from L will be at a time that depends deterministically on whatever action by *Lois* or *Leah* instigated it, even though the first input may not be.

We can achieve this by the following re-programming of the terminal processes above into Timed CSP, where D and D' are suitably chosen delays.

$$\begin{aligned}
TLoisT &= send.lois?x \rightarrow \\
&\quad ((in.lois!x \rightarrow SKIP) \,|||\, (WAIT(D);\ in.lois!x \rightarrow SKIP)); \\
&\quad out.lois.Ack \rightarrow TLoisT \\
TLeahT &= out.leah?x : T \rightarrow rec.leah!x \rightarrow \\
&\quad ((in.leah.Ack \rightarrow SKIP) \,|||\, (WAIT\ D;\ in.leah.Ack \rightarrow SKIP));\ TLeahT \\
THughT &= send.hugh?x \rightarrow in.hugh!x \rightarrow in.hugh!x \rightarrow \\
&\quad out.hugh.Ack \rightarrow WAIT\ D';\ THughT \\
THenryT &= out.henry?x : T \rightarrow rec.henry!x \rightarrow in.henry.Ack \rightarrow \\
&\quad in.henry.Ack \rightarrow WAIT\ D';\ THenryT
\end{aligned}$$

The point about this is that the delays $WAIT\ D'$ in the high-level processes must create the guarantee that within $D-1$ units of starting to try to communicate with the medium, they succeed. One example that works within our timing assumptions is $D = 4$, $D' = 1$.

The result then satisfies the full timed lazy independence specification.

The approach here differs from the one in the previous section in that there is no pre-arranged partition of the resource that the two levels share (i.e. the central medium process), but rather we ensure that the low level process can always get hold of enough of it relative to the reduced and delayed transmission that our model permits to it. The exact share of the central resource that L obtains is then concealed from L itself.

An interesting consequence of this style is that no offer made to L is ever withdrawn in this construction. So in fact whenever the timing is chosen to make the abstracted system quasi-deterministic it is also deterministic. In this case, as observed earlier, we can infer the continuous result from the discrete one without doubling the metronome.

9 Conclusions

We have shown in this paper how definitions of noninterference previously developed for untimed CSP can be adapted to Timed CSP. In doing so we have given new insights into the structures of both the discrete and continuous timed failures models and in particular their refinement-maximal members.

We have developed the idea that, where refinement corresponds to reduction of nondeterminism, specifying that the low-level abstraction is *maximally refined* (i.e. as deterministic as possible) is the right specification of noninterference in some circumstances, including the timed models.

Ouaknine's theory of *digitisation* for Timed CSP has been generalised, and we were able to show that in order to establish continuous-time noninterference for integer Timed CSP, it is sufficient to do so for a discrete model in which the time step *tock* is 0.5 of the units used to represent delay in the program under consideration.

We were able to create FDR checks which decide the above condition for finite-state processes, and applied them to some simple case studies. From these case studies we can conclude that timed noninterference can be decided, at least on small examples, quickly and efficiently.

It was impressed on us, in creating case studies, that creating Timed CSP systems that act deterministicially or quasi-deterministically is not always easy. This should not be surprising when a number of self-timed (as opposed to clock-driven) systems interact, but is of course an issue when our noninterference condition expects us to eliminate most or all nondeterminism from systems with no high-level behaviour (formalised as $P \underset{H}{\parallel} STOP$).

We have showed how a weak noninterference specification (in fact identical in structure to the fault tolerance specification proposed in [22]) can apply in such circumstances. We have found it able to capture timing channels that exist in systems, but unfortunately there are situations where information flow will not be captured. Further practical research is needed on how frequently the strong noninterference specification has to be weakened in this way. This might necessitate further theoretical work in understanding for which sorts of system it, or some variant, might be sufficient. A good alternative, investigated in [8, 9], might be timed variants of Forster's Local Noninterference (LNI) conditions [4, 25]. Further research would be needed to understand these over continuous Timed CSP and to implement timed versions of these in FDR. Some theory akin to those of [13, 12] may also be possible, though the question of what one must be ignorant of seems rather less tangible in the world of process algebras as opposed to models based on assignable state.

We believe that noninterference analysis will become increasingly important thanks to the advent of Cloud computing, in which software and data belonging to multiple parties use common hardware. When two applications, one of which may be specifically designed for gathering information, are sharing an implementation platform, it will be necessary for security to show that information cannot leak from one to the other.

In addition to the type of conditions presented in this paper that ban information flow completely, there is also the need for ones that bound the capacity of any channel from high to low. Of course in a timed context we have the possibility of measuring this in bits per second.

Resources

FDR can be downloaded from `http://www.cs.ox.ac.uk/projects/concurrency-tools/`. Version 2.94 contains all the features used in this paper such as the Timed CSP implementation. Example files for FDR containing examples and case studies from this paper can be found at ???.

Notation of timed traces and failures

The following notation, used in this paper, is common to the literature of Timed CSP.

$s \upharpoonright t$ the sequence of all (timed) events in the trace s up to *and including* those at t.
$\aleph \upharpoonright t = \aleph \cap (\Sigma \times [0, t))$ the refusals in $\aleph$ up to and *not including* those at t.
$end(s)$ the last time appearing in s, or 0 if $s = \langle\rangle$.
$begin(s)$ the first time appearing in s, or ∞ if $s = \langle\rangle$.

References

1. P.G. Allen, *A comparison of non-interference and non-deducibility using CSP*, Proc. CSFW 1991 (IEEE).
2. P.J. Armstrong, G. Lowe, J Ouaknine and A.W. Roscoe, *Model-checking Timed CSP*, Forthcoming 2012 (H. Barringer festschift).
3. R. Focardi and R. Gorrieri, *A classification of security properties for process algebras*, Journal of Computer Security **3**, 1994.
4. R. Forster, *Noninterference properties for nondeterministic processes*, Oxford University DPhil Thesis, 1999.
5. J.A. Goguen and J. Meseguer, *Security policies and security Models*, Proc of IEEE Symposium on Security and Privacy, 1982.
6. J. Graham-Cumming, *The formal development of secure systems*, Oxford University DPhil Thesis 1992.
7. T.A. Henzinger, Z. Manna, and A. Pnueli, *What good are digital clocks? In Proceedings of the Nineteenth International* Colloquium on Automata, Languages, and Programming (ICALP 92), volume 623, pages 545-558. Springer LNCS, 1992.
8. Huang Jian, *Extending non-interference properties to the timed world*, Oxford University D.Phil thesis, 2010.
9. Huang Jian and A.W. Roscoe, *Extending non-interference properties to the timed world*, Proc ACM SAC. 2006.
10. R.S. Lazić, *A semantic study of data independence with applications to model checking*, Oxford University DPhil Thesis 1999.
11. G. Lowe and J. Ouaknine, *On Timed Models and Full Abstraction*, ENTCS 155, pp 497-519, 2006.
12. A.K. McIver and C.C. Morgan, *The thousand-and-one cryptographers*, Reflections on the work of C.A.R. Hoare, Springer 2010.
13. C.C. Morgan, *The Shadow Knows: Refinement of Ignorance in Sequential Programs*, Proc MPC 2006 LNCS 4014.
14. J. Ouaknine, *Discrete analysis of continuous behaviour in real-time concurrent systems*, Oxford University D.Phil thesis, 2001.
15. J. Ouaknine, *Digitisation and full abstraction for dense-time model checking*, TACAS Springer LNCS, 2002.
16. J. Ouaknine and J.B. Worrell, *Timed CSP = Closed Timed epsilon-automata*, Nordic Journal of Computing, **10**, 2003.
17. G.M. Reed, *A uniform mathematical theory for real-time distributed computing*, Oxford University D.Phil thesis, 1988.
18. G.M. Reed and A.W. Roscoe, *A timed model for communicating sequential processes*, Theoretical Computer Science **58**, 249-261, 1988.
19. G.M. Reed and A.W. Roscoe, *The timed failures-stability model for CSP*, Theoretical Computer Science **211**, 85-127, 1999.
20. A.W. Roscoe, *Model checking CSP*, in 'A classical mind: essays in honour of C.A.R. Hoare', Prentice Hall, 1994.

21. A.W. Roscoe, *CSP and determinism in security modelling*, Proc of IEEE Symposium on Security and Privacy, 1995.
22. A.W. Roscoe, *The theory and practice of concurrency* Prentice Hall, 1997.
23. A.W. Roscoe, *Confluence thanks to extensional determinism*, ENTCS **162**, pp305-309, 2006.
24. A.W. Roscoe, *Understanding concurrent Systems*, Springer, 2010.
25. A.W. Roscoe, R. Forster and G.M. Reed, *The successes and failures of behavioural models*, Millennial Perspectives in Computer Science, Palgrave 1999.
26. A.W. Roscoe, P.J. Hopcroft and P. Armstrong, *Fairness analysis through priority*, forthcoming 2012.
27. A.W. Roscoe, J.C.P. Woodcock and L. Wulf, *Non-interference through determinism,* Journal of Computer Security. Vol. 4, no. 1, pp. 27-53. 1996.
28. P.Y.A. Ryan, *A CSP formulation of non-interference and unwinding*, Cipher Winter 1991, IEEE Press.
29. S.A. Schneider, *Concurrent and real-time systems: the CSP approach*, Wiley, 2000.