Article:
Baxter, J., Cavalcanti, A., Gazda, M. et al. (1 more author) (2023) Testing using CSP models: time, inputs, and outputs. ACM Transactions on Computational Logic, 24 (2). 17. ISSN 1529-3785

https://doi.org/10.1145/3572837

Testing using CSP models: time, inputs, and outputs

JAMES BAXTER, University of York, UK
ANA CAVALCANTI, University of York, UK
MACIEJ GAZDA, University of Sheffield, UK
ROBERT M. HIERONS, University of Sheffield, UK

The existing testing theories for CSP cater for verification of interaction patterns (traces) and deadlocks, but not time. We address here refinement and testing based on a dialect of CSP, called *tock*-CSP, which can capture discrete time properties. This version of CSP has been of widespread interest for decades; recently, it has been given a denotational semantics, and model checking has become possible using a well established tool. Here, we first equip *tock*-CSP with a novel semantics for testing, which distinguishes input and output events: the standard models of (*tock*-)CSP do not differentiate them, but for testing this is essential. We then present a new testing theory for timewise refinement, based on novel definitions of test and test execution. Finally, we reconcile refinement and testing by relating timed ioco testing and refinement in *tock*-CSP with inputs and outputs. With these results, this paper provides, for the first time, a systematic theory that allows both timed testing and timed refinement to be expressed. An important practical consequence is that this ensures that the notion of correctness used by developers guarantees that tests pass when applied to a correct system and, in addition, faults identified during testing correspond to development mistakes.

CCS Concepts: • **Software and its engineering → Formal methods**; **Software testing and debugging**; • **Computing methodologies** → *Concurrent computing methodologies*.

Additional Key Words and Phrases: Model-based testing, exhaustive test set, process algebra, refinement

ACM Reference Format:
JAMES BAXTER, ANA CAVALCANTI, MACIEJ GAZDA, and ROBERT M. HIERONS. 2022. Testing using CSP models: time, inputs, and outputs. *ACM Trans. Comput. Logic* 1, 1 (November 2022), 42 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

CSP [43] is a well-established process algebra in use for more than thirty years. Extensive studies of its denotational, algebraic, and operational semantics, and of the formal correspondence between the various semantic definitions, and the availability of powerful model checkers have ensured interest in academia and industry [25, 47]. Model-based testing using CSP has also been studied in several semantic and technological settings [12, 14, 15, 29, 37, 44].

The focus of this paper is the black-box testing of a system under test (SUT) against a specification P written in a timed version of CSP, namely *tock*-CSP. Since testing is black-box, we can only check whether it conforms to P by interacting with the SUT and making observations. In testing, the observations made are sequences that include inputs,

Authors' addresses: JAMES BAXTER, University of York, York, UK, james.baxter@york.ac.uk; ANA CAVALCANTI, University of York, York, UK, ana.cavalcanti@york.ac.uk; MACIEJ GAZDA, University of Sheffield, Sheffield, UK, m.gazda@sheffield.ac.uk; ROBERT M. HIERONS, University of Sheffield, Sheffield, UK, r.hierons@sheffield.ac.uk.

outputs, the passing of time, and refusals: they are timed refusal traces. It is therefore necessary to define a timed refusal trace semantics for tock-CSP that appropriately deals with the differences between inputs and outputs. One of our novel contributions here is such a (compositional) semantics for testing, along with proofs that the semantics satisfies the healthiness conditions of the existing *tock*-CSP semantics [3].

To reason about test effectiveness, we make the assumption normally made in model-based testing that the SUT behaves like an *unknown* model Q that can be expressed using the same formalism as the specification. Another contribution here is a testing theory based on this assumption that justifies our novel testing approach. In this theory, testing involves comparing two tock-CSP processes: a specification P and the unknown process Q that represents the SUT. Comparisons are made by applying test cases to the SUT and checking that the observations made are allowed.

Although there has been work on testing from CSP models [12, 14, 15, 29, 37, 44], we are not aware of any work on testing using timed CSP models. A first issue is how time should be represented in models, with early works on CSP considering a continuous-time paradigm and the Timed CSP notation [21, 41, 46]. Work on tools for verification, however, has been based on *tock*-CSP [42], which uses a special event *tock* to encode the passage of discrete time. With *tock*-CSP we can use existing tools for reasoning, and can specify deadlines via timestops, that is, by refusing *tock*, as well as timeouts and time budgets. Use of *tock*-CSP is widespread, covering verification of robotics simulations [18], security properties [22], distributed adaptive systems [26], design of I/O controllers [30], and railway systems [27].

In all these areas of application, and, more generally, in the development of embedded control systems, use of testing for verification is prevalent. Such reactive systems interact with their environment through inputs and outputs. They typically operate through cycles defined by time periods in which they receive values from sensors (inputs), perform calculations, and then send values to actuators (outputs). Such systems are time sensitive, and can often be time critical. Several modelling notations for embedded control systems have a CSP-based semantics. Some examples are MATLAB Simulink [11, 48] and Stateflow [35, 49], which are de facto standards in the transport industry, and RoboChart and RoboSim [18, 36], which have been introduced to support the development of robotic systems and have a *tock*-CSP semantics. With the work presented here, we enable the use of models described in any of these notations, which are appropriate for verification by model checking and proof using CSP, also to support testing.

Although the existence of model checkers for CSP makes it possible to formally verify a CSP model against another, there is often also a need to carry out testing. This is particularly the case if we can only reason about the behaviour of the implementation through interacting with it: we do not, for example, have access to the source code. As said, however, we assume that there is an unknown CSP model that captures the behaviour of the SUT. This makes it possible to formally reason about the effectiveness of our proposed testing approach and, for example, prove that all tests lead to a valid verdict, and we identify all tests that would be needed to guarantee that all possible faults can be found.

Typically, a tester applies a *test case* that determines its behaviour. For example, a test case might indicate that the tester should start by applying an input i_1 and then apply a second input i_2 if output o_1 is observed in response to i_1. Test cases can be positive, in the sense that they attempt to verify the presence of allowed behaviours, or negative, if they attempt to find disallowed behaviours. Here, we consider negative testing, and so a test case is defined in terms of a behaviour not allowed by a CSP specification: if the tester observes this behaviour then the SUT must be faulty.

The testing theories for CSP are for untimed models [12, 15, 16, 40]. All these theories have some aspects in common: they embed the usual assumption in testing that models and systems are divergence free. This means that a CSP specification under consideration cannot follow an infinite sequence of internal (unobservable) events. This requirement is justified by the fact that in a model, divergence is taken as a mistake. Several CSP tools support the verification of divergence freedom [23]. In addition, in an SUT, divergence is observed in testing as a deadlock.

All of the testing theories consider two types of observations. The simplest record is a trace: a sequence of observable events. Traces can be enriched by recording refusals of sets X of observable events: the situation in which the SUT or model is in a stable state (it cannot change state through internal events) and cannot engage in any of the events from X. Such a refusal is typically observed in testing through the tester offering the events in X to the SUT and observing a deadlock, detected through a timeout. Under *traces refinement* the SUT is correct with respect to a CSP specification P if all traces of the SUT are also traces of P. Alternative forms of refinement (notions of correctness) consider also observations of refusals to identify allowed and forbidden deadlocks. All existing testing theories for CSP test for traces refinement and for deadlock separately to simplify definitions and proofs. All theories follow exactly the same approach to testing for traces refinements, in spite of adopting very different semantic models.

Here, we consider, for the first time, testing for *tock*-CSP. We show that, in this case, separate testing for traces and deadlock is still possible. Testing for traces refinement, however, is very different due to the special nature of the *tock* event, and testing for (intermediate allowed) deadlocks is simpler, because the tester can observe passage of time.

As a second contribution in this paper, we also enrich the *tock*-CSP denotational semantics to cater for inputs and outputs. In testing there is an asymmetry: the SUT controls outputs, while the tester controls inputs. So, the usual CSP approach, in which inputs and outputs are both treated as synchronisations is not practical. Using the new model presented here, we define input-output *tock*-CSP refinement, which we use as the notion of correctness (conformance relation) for testing. We identify a test set for a given CSP model and prove that it is exhaustive for input-output *tock*-CSP refinement. This means that every faulty implementation fails at least one test case in the set.

The denotational semantics of a *tock*-CSP process [3] has a number of properties that are not captured in the untimed semantics of CSP. The semantics of a *tock*-CSP process is given by a set of traces, each recording a sequence of events, including *tock*, and refusals. Refusals are observed before a *tock* event, and, possibly at the end of the trace. As a result, we cannot record the passage of time in an unstable state. So, internal events that do not require agreement with the environment are urgent; this ensures predictability and maximal progress.

When considering processes with inputs and outputs, we make the usual assumption that the tester (or environment) controls inputs, the SUT controls outputs, and the tester cannot block outputs produced by the SUT. As a result, the SUT cannot be made to deadlock if an output is enabled, that is, the tester cannot block enabled outputs. As a consequence, since the observation of a refusal involves observing deadlock, a tester cannot observe a refusal if the SUT is in a state in which it can produce an output. We therefore take the view that a divergence-free *tock*-CSP process is in a stable state if no internal events and also no outputs are enabled. These are quiescent states in which, for example, time can pass. We can, in addition, observe the refusal of inputs: models and implementations need not be input enabled, in contrast with many testing approaches. We adopt this notion of stability to define an input-output model for *tock*-CSP. We formalise its notion of timed traces and healthiness conditions, and calculate definitions for its operators.

Nondeterminism in the SUT can lead to the situation in which there are a number of possible results of applying a test case T to the SUT and so there is the additional problem of determining when a test case T has been applied a sufficient number of times. In order to handle possible nondeterminism in the SUT, we use the standard test hypothesis that corresponds to a fairness condition: we assume that there is some value k such that testing with a test case T a total of k times is guaranteed to lead to all possible behaviours (for T and SUT) being observed. The choice of k might depend on domain knowledge or criticality, and there is the potential to use probabilistic arguments.

Given the distinctive role of inputs and outputs in testing, there has been much interest in input-output transition systems (IOTSs) [51] and the ioco implementation relation [5, 55, 56]. In that context, observations are traces that record inputs, outputs, and quiescence. Several timed variants of ioco exist. The initial relation of Krichen and Tripakis, called

tioco, treated the passing of (discrete or continuous) time in a similar way to outputs, but did not consider quiescence as an observation [32]. In addition, at about the same time, an implementation relation called *rt-ioco*, equivalent to the Krichen and Tripakis tioco, was introduced by Larsen, Mikucionis and Nielsen [33]. Later, Schmaltz, and Tretmans produced a version of tioco in which quiescence is also considered [45].

Our third contribution is establishing the relationship between input-output *tock*-CSP refinement and the Schmaltz and Tretmans variant of tioco. We focus on this version of tioco because its consideration of quiescence makes it a stronger relation than the original Krichen and Tripakis version, and we are interested in whether our refinement relation is sufficiently strong and discriminating. We show that it is stronger than tioco.

Refinement is a conformance relation suitable for development, when we are dealing with models and code. On the other hand, ioco and its timed variants take into account the restricted observability of a testing experiment. It makes, however, no sense for developers and testers to use unrelated notions of conformance. The notion of correctness used by developers should ensure that the tests pass when applied to correct systems. Conversely, faults identified during testing should correspond to development mistakes. Here, for the first time, we identify how timed ioco-based testing can shed light on refinement proof, and how design by timed refinement may influence testing. Our work unifies these verification approaches, making it meaningful to collect and compare diverse evidence arising from both techniques.

In summary, we present here the first theory for model-based testing using a timed version of CSP, namely, *tock*-CSP, with a novel notion of test case. For that, we also present a new denotational semantics for *tock*-CSP that distinguishes inputs and outputs. Finally, we compare a strong version of tioco with our notion of conformance in *tock*-CSP, and show it can be used for consistent guidance during both development and testing.

This paper is structured as follows. Next, we present *tock*-CSP and its semantics. In Section 3, we present our *tock*-CSP semantics with inputs and outputs, and its refinement notion adopted in our testing theory. Section 4 describes IOLTS and the timed ioco variants, with Section 5 discussing the relationship between input-output *tock*-CSP refinement and the Schmaltz and Tretmans tioco. Testing is addressed in Section 6. We conclude in Section 7, where we also discuss related and future work. Appendix A presents definitions of the original *tock*-CSP semantics referenced here. Appendix B reproduces the definitions of all the semantic functions defined in this paper for ease of reference. A complete set of proofs for all lemmas and theorems is available in [2].

2 PRELIMINARIES: *tock*-CSP

Here, we describe one of the notations we use: *tock*-CSP. We describe the others, IOLTS and IOTS, in Section 4.

Systems and their components are modelled in CSP using processes that interact with each other and with their environment via events. In *tock*-CSP, the same approach is adopted, but, as mentioned, a special event *tock* marks the passage of time. Events are atomic and instantaneous. In any given model, Σ is the set of all declared events.

The process operators of *tock*-CSP are basically those of CSP. The behaviours defined by them, however, consider the special nature of the *tock* event. We explain the main operators below, and refer to [3] for a complete account. There, we can find additional examples and a formal semantics, also described below and reproduced in Appendix A.

A process that is ready to synchronise with the environment on an event a can be written using the prefixing operator $\rightarrow$ as follows $a \rightarrow P$. This process, after engaging on a, when the environment is ready, behaves like the process P. While the environment is not ready, $a \rightarrow P$ waits, and time may pass. So, any number of *tock* events may occur before a.

The processes **div**, **Stop**, and **Skip** diverge, deadlock, and terminate immediately. A special event $\checkmark$ marks termination, and cannot be used in process definitions. The set $\Sigma^{\checkmark}$ contains $\checkmark$ as well as all declared events, and $\Sigma^{\checkmark}_{tock}$ contains *tock* in addition. A process **Wait** n pauses for n time units before terminating; so, n occurrences of *tock* happen before a $\checkmark$.

Processes can also be combined in sequence $P; Q$, where, as usual, the behaviour is that of P, until it terminates, when Q takes over. Another core operator is external choice ($\Box$), which offers to the environment the possibility of choosing between processes, via an interaction on events that are initially available. While waiting for the choice, time may pass. So, *tock* events may occur, but they do not interfere with the choice. We provide an example.

Example 2.1. We present below a process *RD* that models a simple rescue drone and offers the environment a choice.The environment exercises its choice by interacting on either of the events *takeoff* or *turnoff*. If it chooses the event *takeoff*, then the behaviour of *turnoff* $\rightarrow$ **Skip** is no longer possible. Equally, if the environment chooses *turnoff*, the event *takeoff* is no longer available. The choice offered is based on *takeoff* and *turnoff*.

$$RD = takeoff \rightarrow \mathbf{Wait}\ 1;\ move \rightarrow found \rightarrow land \rightarrow \mathbf{Stop} \ \Box\ turnoff \rightarrow \mathbf{Skip}$$

After *takeoff*, *RD* pauses for 1 time unit (**Wait** 1), before offering the event *move*, and waiting for as long as it takes before *move* is accepted by the environment. The events offered subsequently are *found* and *land*, after which *RD* deadlocks (**Stop**) and then the only possible events are *tock*. If *turnoff* is chosen instead, *RD* terminates (**Skip**) and after a ✓, no other events take place. □

Another form of choice is internal ($\sqcap$), where the environment has no control.

Example 2.2. In the above example, instead of **Wait** 1, we might have a process **Wait** 1 $\sqcap$ **Wait** 2. This may capture, for example, the fact that the delay in the execution of the take off operation (represented by the event *takeoff*) depends on features of a specific drone. So, an implementation may pause for 1 or 2 time units before calling the operation represented by the event *move*. The environment has no possibility to interfere with this choice. □

A salient feature of *tock*-CSP is the possibility of defining deadlines. The deadline operator $P \blacktriangleright d$ defines a process that must terminate within d time units. This is a derived operator, which can be defined using a timestop: $\mathbf{Stop}_U$. This is a form of deadlock that does not allow time to pass: it timelocks, as no *tock* event can be recorded.

Example 2.3. We present below a process *RDL* that models the landing of the drone.

$$RDL = found \rightarrow ((land \rightarrow \mathbf{Skip}) \blacktriangleright 1);\ \mathbf{Stop}$$

In this example, once the target is *found*, the drone *RDL* must *land* in at most one time unit. □

Other operators are presented as needed. They include, for instance, timeout, parallelism, hiding, and renaming.

Originally, *tock*-CSP [42] was proposed as a form of using the standard models of CSP, notably, the stable-failures model, and its model checker FDR [25], without changes, to reason about time properties. In this context, however, the interpretation of *tock* as marking the passage of time has both positive and negative semantic consequences. On the positive side, it becomes simple to define processes that impose a deadline by refusing or controlling the passage of time. On the negative side, the semantics of some operators do not correspond to what may be expected. For instance, if *tock* is not recognised as a special event, an external choice can be resolved by passage of time.

To address some of these drawbacks, the most recent version of FDR supports the possibility of defining a timed section, where the operators have a semantics that is sensitive to the special nature of *tock*. For instance, implicitly, all parallel processes synchronise on *tock* to guarantee uniform passage of time across a system. None of the existing denotational semantics of CSP, however, can cater fully for the behaviour of (*tock*-CSP) processes in a timed section.

Over the years, a variety of semantic models have been proposed for *tock*-CSP [1, 34, 38]. None of them caters for deadlines, termination, Zeno behaviour, and the standard (failures-based) semantics within each time unit as expected

of *tock*-CSP processes. We, therefore, adopt the richer and recent ✓-tock model in [3]. As explained, in this approach, processes P are modelled by a set $tt[\![P]\!]$ of traces recording events and refusals. Formally, they are sequences whose elements are observations from the set *Obs* defined below of events from $\Sigma^{\checkmark}_{tock}$ or refusal sets from $\Sigma^{\checkmark}_{tock}$.

$$Obs ::= evt\langle\langle \Sigma^{\checkmark}_{tock} \rangle\rangle \mid ref\langle\langle \mathbb{P}\, \Sigma^{\checkmark}_{tock} \rangle\rangle$$

In examples, we often omit the type constructors *evt* and *ref* for brevity.

Example 2.4. The set $tt[\![RD]\!]$ includes the traces sketched below. The empty trace $\langle\rangle$ is in every set $tt[\![P]\!]$.

$$\langle\rangle, \quad \langle \Sigma^{\checkmark} \setminus \{takeoff, turnoff\}\rangle, \quad \langle \Sigma^{\checkmark} \setminus \{takeoff, turnoff, move\}\rangle, \quad \ldots, \quad \langle \emptyset \rangle,$$

Since there is no divergence, we can immediately observe the refusal of every event except *tock*, *takeoff*, and *turnoff*. If a set of events can be recorded as a refusal, so can all its subsets, including the empty set $\emptyset$ of refusals. In this sense, we have subset closure of refusals. In examples, we normally present just maximal refusals.

$$\langle takeoff\rangle, \quad \langle \Sigma^{\checkmark} \setminus \{takeoff, turnoff\}, tock\rangle, \quad \langle \Sigma^{\checkmark} \setminus \{takeoff, turnoff\}, tock, takeoff\rangle,$$
$$\langle \Sigma^{\checkmark} \setminus \{takeoff, turnoff\}, tock, \Sigma^{\checkmark} \setminus \{takeoff, turnoff\}, tock, \ldots\rangle, \quad \langle \Sigma^{\checkmark} \setminus \{takeoff, turnoff\}, tock, \ldots, takeoff\rangle,$$

The event *takeoff* can be observed immediately, or we can instead observe 1 time unit pass, recorded as a refusal followed by a *tock*. The event *takeoff* can again be observed after 1 time unit, that is, after an event *tock* and a refusal, because a *tock* is always preceded by a refusal. In fact, *takeoff* can be observed after any number of *tock* events. Passage of time does not resolve the choice, so after any number of *tock* events, neither *takeoff* or *turnoff* are refused.

$$\langle takeoff, \Sigma^{\checkmark}\rangle, \quad \langle takeoff, \Sigma^{\checkmark}, tock\rangle, \quad \langle takeoff, \Sigma^{\checkmark}, tock, move\rangle, \quad \langle takeoff, \Sigma^{\checkmark}, tock, \Sigma^{\checkmark} \setminus \{move\}\rangle,$$
$$\langle takeoff, \Sigma^{\checkmark}, tock, \Sigma^{\checkmark} \setminus \{move\}, tock, move\rangle, \quad \langle takeoff, \Sigma^{\checkmark}, tock, \Sigma^{\checkmark} \setminus \{move\}, tock, \ldots, move\rangle, \quad \ldots$$

After a *takeoff*, all events, except *tock* are refused because *RD* has to wait for 1 time unit before offering *move*. After that *tock*, *move* is no longer refused. It might happen immediately, or after any number of *tock* events. The sets $tt[\![P]\!]$ are prefix closed: if it includes a trace, it includes all its prefixes. In examples in the sequel, we normally elide prefixes.

$$\langle turnoff, \checkmark\rangle, \quad \langle \Sigma^{\checkmark} \setminus \{takeoff, turnoff\}, tock, turnoff, \checkmark\rangle, \quad \langle \Sigma^{\checkmark} \setminus \{takeoff, turnoff\}, tock, \ldots, turnoff, \checkmark\rangle$$

Once a *turnoff* takes place, *RD* terminates immediately and time is no longer recorded. □

Example 2.5. The set $tt[\![RDL]\!]$ includes the traces sketched below, which record the deadline.

$$\langle found, land, \Sigma^{\checkmark}, tock, \Sigma^{\checkmark}, tock, \ldots\rangle,$$
$$\langle found, \Sigma^{\checkmark} \setminus \{land\}, tock, land, \Sigma^{\checkmark}, tock, \Sigma^{\checkmark}, tock, \ldots\rangle, \langle found, \Sigma^{\checkmark} \setminus \{land\}, tock, \Sigma^{\checkmark}_{tock} \setminus \{land\}\rangle, \quad \ldots$$

Once *found* happens, *land* might happen immediately, and then *RDL* deadlocks, and so all events except *tock* are refused. Alternatively, *land* might happen after 1 time unit: after a *tock* associated to a refusal that does not include *land*, or *tock*. After *land* occurs, again, all events except *tock* are refused. After one *tock*, however, while *land* does not take place, another *tock* is refused, and so, cannot happen (until *land* does). Time cannot pass, as that violates the deadline. □

The set of sequences of *Obs* elements where refusals occur only before a *tock* or at the end, *tock* is not included in a refusal that precedes *tock*, and *tick* occurs only at the end, is called *TTTrace*. The traces in the sets in the range of $tt[\![\_]\!]$ belong to *TTTrace*. These sets also satisfy additional healthiness conditions. For example, $\langle\rangle$ is in all these sets.

As said, the standard models of CSP and *tock*-CSP, including the model described above, do not distinguish between input and output events. As a syntactic abbreviation, we can write, for example, a prefixing $in?x \rightarrow P$ to describe a process that takes an input x through a channel in. This is, however, just a shorthand for an external choice over events $in.v$, for every value v of the type of in. An event $in.v$ is composed, but, like any other event, requires synchronisation. We can also write $out!v$ for a composed event $out.v$. It is used to indicate that $out.v$ is meant to be an output of a value v through a channel out. Again, in the standard semantic models, $out.v$ is just like any other event.

Distinctively, CSP and *tock*-CSP are process algebras for refinement. Each model defines a refinement relation. In each case, refinement holds between a specification process P and an implementation process Q when the behaviour of Q is a subset of that of P. For *tock*-CSP, ✓-tock refinement $P \sqsubseteq Q$ requires $tt[\![Q]\!] \subseteq tt[\![P]\!]$. This means that the interactions and deadlocks (refusals) of Q are possible for P, in the same order, and at the same time unit. Refinement allows, however, reduction of nondeterminism, since there may be behaviours of P that are not present in Q.

One of our goals is to study the relationship between this notion of refinement, in the context of the novel *tock*-CSP model that caters for inputs and outputs, and timed ioco, discussed in Section 4.

3 INPUTS AND OUTPUTS IN *tock*-CSP

Practical testing techniques require notions of input and output. Here, first, in Section 3.1, we define the input-output ✓-tock model and its healthiness conditions, and in Section 3.2, we define input-output ✓-tock refinement and explore some of its properties. Next, in Section 3.3 we define an input-output ✓-tock model for the *tock*-CSP operators by calculation from the definitions in Section 3.1 and their original ✓-tock semantics (in [3] and Appendix A).

3.1 Input-output ✓-tock model

To distinguish inputs and outputs, we capture the fact that, as already said, a process is stable when it cannot engage in internal events or outputs. Therefore, when stable, a process can refuse all outputs. This leads to the definition below of the set $iott^O[\![P]\!]$ of input-output ✓-tock traces for a process P and set O of output events. We divide Σ into disjoint sets I and O of inputs and outputs; ✓ and *tock* are neither inputs nor outputs.

Definition 3.1. $iott^O[\![P]\!] \mathrel{\widehat{=}} \{\rho : TTrace \mid addOuts^O(\rho) \in tt[\![P]\!]\}$

The trace $addOuts^O(\rho)$, defined inductively below, records exactly the same events e as in the given trace ρ, but its refusals $X \cup O$ are a superset of the refusals X in ρ that includes all outputs O. The set $iott^O[\![P]\!]$ includes the traces ρ for which $addOuts^O(\rho)$ is in $tt[\![P]\!]$. So, a trace ρ is included in $iott^O[\![P]\!]$ only if all the refusals X that it records are in stable states, where all outputs are refused. As an aside, we note that, because $addOuts^O(\rho)$ is a trace of $tt[\![P]\!]$, subset closure of refusals means that ρ is in $tt[\![P]\!]$ as well. So, $iott^O[\![P]\!] \subseteq tt[\![P]\!]$.

$$addOuts^O(\langle\rangle) = \langle\rangle$$
$$addOuts^O(\langle ref\ X\rangle \frown \rho) = \langle ref\ (X \cup O)\rangle \frown addOuts^O(\rho)$$
$$addOuts^O(\langle evt\ e\rangle \frown \rho) = \langle evt\ e\rangle \frown addOuts^O(\rho)$$

To simplify our notation, in the sequel we leave the extra parameter O of *addOuts* implicit. Above, we define that the empty trace is unaffected by *addOuts*. For a trace formed by the singleton trace $\langle ref\ X\rangle$ concatenated (operator $\frown$) with another trace ρ, the resulting trace has an initial refusal $X \cup O$ instead, followed by the result of the recursive application of *addOuts* to ρ. If the singleton trace has an event, the definition is similar, but the event is not changed.

Example 3.2. We reproduce below the process *RD* from Example 2.4.

$$RD = takeoff \rightarrow \mathbf{Wait}\ 1;\ move \rightarrow found \rightarrow land \rightarrow \mathbf{Stop} \ \Box\ turnoff \rightarrow \mathbf{Skip}$$

We take the outputs to be *takeoff*, *move*, and *land* (representing interactions with actuators of the drone that control its flight), with *found* and *turnoff* as inputs representing a sensor able to identify a target and a command to turn off the drone. In this case, some of the traces of $iott^O[[RD]]$ are as follows.

$\langle\,\rangle$,
$\langle takeoff, \Sigma^{\checkmark}, tock, move\rangle$,
$\langle takeoff, \Sigma^{\checkmark}, tock, move, found\rangle$, $\langle takeoff, \Sigma^{\checkmark}, tock, move, \Sigma^{\checkmark} \setminus \{found\}, tock, found\rangle, \ldots$
$\langle turnoff\rangle, \langle turnoff, \checkmark\rangle$

We can no longer observe a refusal before a *takeoff* or *turnoff* because, since *takeoff* is an output, *RD* is not stable at the start. This is captured by the fact that $\langle O\rangle$ is not a trace in $tt[[RD]]$, and so not included in $iott^O[[RD]]$. Once *takeoff* takes place, we have stability due to the **Wait** 1, and so can observe refusals. Afterwards, again, since *move* is an output, we cannot observe a refusal before it occurs. Like internal events, outputs are urgent, because, since a refusal cannot be observed, neither can *tock*. As illustrated here, however, we can model behaviours in which outputs take time to be computed by explicit uses of **Wait** processes. Once *move* takes place, since *found* is an input, it can be observed immediately, or after any number of *tock* events. This is captured, for example, by $\langle takeoff, \Sigma^{\checkmark}, tock, move, found\rangle$.

Instead of a *takeoff* event, we may have a *turnoff* at the start. After that, *RD* terminates immediately. Now *turnoff* must be immediate, or the choice is resolved in favour of the output *takeoff*. □

In the input-output ✓-tock model, imposing deadlines on outputs is unnecessary. As mentioned, since we can observe passage of time (*tock*) only after a refusal, and we cannot observe a refusal if an output is available, outputs are urgent, and so any deadline is redundant. This is illustrated by the following example.

Example 3.3. We recall the process *RDL* that models the landing of the drone.

$$RDL = found \rightarrow ((land \rightarrow \mathbf{Skip}) \blacktriangleright 1);\ \mathbf{Stop}$$

If *land* is an output, and *found* an input, $iott^O[[RDL]]$ includes the trace $\langle \Sigma^{\checkmark} \setminus \{found\}, tock, found, land\rangle$. This records the observation that *found* happens after one time unit. Since all outputs belong to $\Sigma^{\checkmark} \setminus \{found\}$, the initial refusals are stable. The set $iott^O[[RDL]]$ does not, however, include $\langle found, \Sigma^{\checkmark} \setminus \{land\}, tock, land\rangle$, which records the occurrence of *land* after 1 time unit, because $\langle found, \Sigma^{\checkmark}, tock, land\rangle$ is not a trace of *RDL*. For a similar reason, $iott^O[[RDL]]$ does not include $\langle found, \Sigma^{\checkmark} \setminus \{land\}, tock, \Sigma^{\checkmark}_{tock} \setminus land\rangle$. So, once *found* happens, *land* is urgent, because a refusal and, therefore, a *tock* cannot be observed before *land*. □

Deadlines on inputs, however, are still useful.

Example 3.4. Below, we present a model of the landing that imposes a deadline on finding the target instead.

$$RDFL = ((found \rightarrow \mathbf{Skip}) \blacktriangleright 1);\ (land \rightarrow \mathbf{Stop})$$

In this example, the trace $\langle \Sigma^{\checkmark} \setminus \{found\}, tock, \Sigma \setminus \{found\}\rangle$, recording the deadline via the refusal of *tock*, is in both $tt[[RDFL]]$ and $iott^O[[RDFL]]$ since $\Sigma^{\checkmark} \setminus \{found\}$ and $\Sigma \setminus \{found\}$ include all outputs. □

As said, the set *TTTrace* of well-formed ✓-*tock* traces is the subset of sequences of *Obs* (seq *Obs*) satisfying additional restrictions. Namely, ✓ can occur only at the end of a trace, a refusal can occur only at the end of the trace or before a

tock event, and every *tock* event must be preceded by a refusal that does not include *tock*.

$$\begin{aligned} TTTrace == \; & \{\rho : \mathrm{seq}\, Obs \mid \forall\, i : \mathrm{dom}\, \rho \bullet \\ & \quad (i < \#\rho \Rightarrow \rho\, i \neq evt\, \checkmark) \wedge \\ & \quad (i < \#\rho \wedge \rho\, i \in \mathrm{ran}\, ref \Rightarrow \rho\, (i+1) = evt\, tock) \wedge \\ & \quad (\rho\, i = evt\, tock \Rightarrow i > 1 \wedge \rho\, (i-1) \in \mathrm{ran}\, ref \wedge tock \notin (ref^{\sim})\, (\rho\, (i-1))) \\ & \} \end{aligned}$$

We use the mathematical notation of Z in our formal definitions [57], and explain any unusual operators as needed. Sequences are indexed from 1. The size of a sequence ρ is given by $\#\rho$. The operator ran defines the range of a function (such as the type constructors *evt* and *ref*) or the set of elements of a sequence. For a function f, we use $f^{\sim}$ for its inverse. So, above $(ref^{\sim})\, (\rho\, (i-1))$ is the set that defines the refusal in position $i-1$ of the trace ρ.

For every process P and set of output events O, the set $iott^O[\![P]\!]$ satisfies the healthiness conditions of the ✓-tock model, which we describe below. The proof of this result can be found in [2]. Here, first of all, we reproduce the definition of the type *TTTrace* of traces used in the definitions of the sets $tt[\![P]\!]$ and $iott^O[\![P]\!]$.

Processes are subsets of *TTTrace* and are required to satisfy four healthiness conditions. The first, **TT0** simply requires that a ✓-tock process P must have a nonempty set of traces. As said, they all have at least the empty trace.

$$\mathbf{TT0}(P) \qquad P \neq \emptyset$$

The second, **TT1**, requires that P is prefix and subset closed, specified in terms of a combined prefix and subset relation $\precsim$, under which $\rho \precsim \sigma$ if ρ can be formed from a prefix of σ by replacing some or all of the refusals in it with subsets.

$$\mathbf{TT1}(P) \qquad \rho \precsim \sigma \wedge \sigma \in P \Rightarrow \rho \in P$$

A formal definition for $\rho \precsim \sigma$ is provided in Appendix A.

The third healthiness condition, **TT2**, specifies that, wherever a refusal X occurs, a set of events Y that cannot happen at that point can be added to it. This ensures that every event that cannot be performed is refused. In the definition of **TT2** below, the set Y is characterised by being disjoint from the set of events e that can occur immediately at the point where X is observed as recorded by an extended trace $\rho \frown \langle evt\, e \rangle$. For *tock*, the extension needs to include the refusal X as well, since *tock* can occur only after a refusal. The operator $\frown$ is sequence concatenation.

$$\begin{aligned} \mathbf{TT2}(P) \qquad & \rho \frown \langle ref\, X \rangle \frown \sigma \in P \wedge \\ & Y \cap \{e : \Sigma^{\checkmark}_{tock} \mid (e \neq tock \wedge \rho \frown \langle evt\, e \rangle \in P) \vee (e = tock \wedge \rho \frown \langle ref\, X, evt\, tock \rangle \in P)\} = \emptyset \\ & \quad \Rightarrow \rho \frown \langle ref\, (X \cup Y) \rangle \frown \sigma \in P \end{aligned}$$

Finally, the fourth healthiness condition, **TT3**, specifies that wherever a refusal X occurs, there is a corresponding trace with ✓ added to the refusal. This ensures that ✓ is always refused when the process is stable. As a consequence, when ✓ happens, its record in traces always shows that it happens unstably.

$$\mathbf{TT3}(P) \qquad \rho \frown \langle ref\, X \rangle \frown \sigma \in P \Rightarrow \rho \frown \langle ref\, (X \cup \{\checkmark\}) \rangle \frown \sigma \in P$$

The input-output ✓-tock model characterised by $iott^O[\![P]\!]$ satisfies the extra healthiness condition **TT4** defined below.

It similar to **TT3**(P), but captures the instability of outputs, rather than termination.

TT4(P) $\quad \rho \frown \langle ref\ X \rangle \frown \sigma \in P \Rightarrow \rho \frown \langle ref\ (X \cup O) \rangle \frown \sigma \in P$

TT4 requires that the whole set of outputs can be added to any refusal. So, if any of them happens, the record shows instability like for ✓. The following theorem proves that the input-output ✓-tock model satisfies **TT4**.

Theorem 3.5. *If $tt[[P]]$ satisfies the healthiness conditions of the ✓-tock model then* **TT4**$(iott^O[[P]])$.

Proof.

$\rho \frown \langle ref\ X \rangle \frown \sigma \in iott^O[[P]]$

$\Rightarrow addOuts(\rho \frown \langle ref\ X \rangle \frown \sigma) \in tt[[P]]$ [definition of $iott^O[[P]]$]

$\Rightarrow addOuts(\rho) \frown \langle ref\ (X \cup O) \rangle \frown addOuts(\sigma) \in tt[[P]]$ [definition of *addOuts*]

$\Rightarrow addOuts(\rho) \frown addOuts(\langle X \cup O \rangle) \frown addOuts(\sigma) \in tt[[P]]$ [idempotence of $\cup$]

$\Rightarrow addOuts(\rho \frown \langle X \cup O \rangle \frown \sigma) \in tt[[P]]$ [property of *addOuts*]

$\Rightarrow \rho \frown \langle ref\ (X \cup O) \rangle \frown \sigma \in iott^O[[P]]$ [definition of $iott^O[[P]]$]

□

We next study refinement based on behaviour characterised by $iott^O[[P]]$.

3.2 Input-output ✓-tock refinement

The refinement relation $\sqsubseteq_{IOTT}$ can be defined in the natural way adopted in all CSP models: subset inclusion.

Definition 3.6 (Input-output ✓-tock refinement). $P \sqsubseteq_{IOTT} Q \mathrel{\widehat{=}} iott^O[[Q]] \subseteq iott^O[[P]]$

It is reassuring that ✓-tock refinement ensures input-output ✓-tock refinement.

Theorem 3.7. $P \sqsubseteq Q \Rightarrow P \sqsubseteq_{IOTT} Q$.

Proof.

$P \sqsubseteq Q = tt[[Q]] \subseteq tt[[P]]$ [definition of $\sqsubseteq$]

$\Rightarrow \{\rho : TTTrace \mid addOuts(\rho) \in tt[[Q]]\} \subseteq \{\rho : TTTrace \mid addOuts(\rho) \in tt[[P]]\}$ [property of sets]

$= iott^O[[Q]] \subseteq iott^O[[P]]$ [definition of $iott^O[[\_]]$]

$= P \sqsubseteq_{IOTT} Q$ [definition of $\sqsubseteq_{IOTT}$]

□

As usual for the richer notions of refinement in CSP, it allows for reduction of nondeterminism, but does not require elimination. Moreover, since the sets of traces representing a process are prefix closed (**TT1**), all partial observations of a behaviour characterised by a trace are regarded as acceptable.

In the input-output model, a refusal cannot be observed if an output is possible (see **TT4**). Further, a refusal cannot be observed if ✓ is possible. As a result, wherever a refusal X is possible, so is the refusal $X \cup O \cup \{✓\}$. Given that

refusals are downwardly closed, this suggests that we can characterise the set of possible input-output ✓-tock traces of a process in terms of those in which all refusals contain $O \cup \{✓\}$ as a subset.

In what follows, we define a function $iott^O_M[\![TT]\!]$ that characterises the subset of such traces for a given set of input-output ✓-tock traces TT. We then show that the input-output semantics $iott^O[\![P]\!]$ of a process and $\sqsubseteq_{IOTT}$ can be defined using $iott^O_M[\![\_]\!]$. This alternative semantic function is useful in proofs.

The definition of $iott^O_M[\![TT]\!]$ uses the function *addTick*, whose definition presented next is similar to that of *addOuts*.

$$addTick(\langle\rangle) = \langle\rangle$$
$$addTick(\langle ref\ X\rangle \frown \rho) = \langle ref\ (X \cup \{✓\})\rangle \frown addTick(\rho)$$
$$addTick(\langle evt\ e\rangle \frown \rho) = \langle evt\ e\rangle \frown addTick(\rho)$$

The ✓-tock trace $addTick(\rho)$ differs from ρ just in that ✓ is in all its refusals.

Definition 3.8. Given a healthy set TT of ✓-tock traces,

$$iott^O_M[\![TT]\!] \mathrel{\widehat{=}} \{\rho : \mathrm{ran}\ addTick \mid addOuts(\rho) \in TT \bullet addOuts(\rho)\}$$

The function *addOuts* is the identity on all elements of $iott^O_M[\![tt[\![P]\!]]\!]$ (see proof in [2]). The theorem below uses this result to establish a relationship between $iott^O[\![P]\!]$ and $iott^O_M[\![tt[\![P]\!]]\!]$. Briefly, $iott^O[\![P]\!]$ can be obtained by downward closure with respect to ✓-tock trace prefixing $\lesssim$ of $iott^O_M[\![tt[\![P]\!]]\!]$.

Theorem 3.9.

$$iott^O[\![P]\!] = \{\rho : TTTrace \mid \exists\, \rho_1 : iott^O_M[\![tt[\![P]\!]]\!] \bullet \rho \lesssim \rho_1\}$$

Proof. *Case* ($\Rightarrow$).

$\rho \in iott^O[\![P]\!]$

$\Rightarrow addOuts(\rho) \in tt[\![P]\!]$ [definition of $iott^O[\![P]\!]$]

$\Rightarrow addOuts(addTick(\rho)) \in tt[\![P]\!]$ [$tt[\![P]\!]$ is **TT3**, and commutativity of *addTick* and *addOut*]

$\Rightarrow addOuts(addTick(\rho)) \in iott^O_M[\![tt[\![P]\!]]\!]$ [$addOuts(addTick(\rho)) \in \mathrm{ran}\ addTick$ and definition of $iott^O_M[\![P]\!]$]

$= addOuts(addTick(\rho)) \in iott^O_M[\![tt[\![P]\!]]\!] \wedge \rho \lesssim addOuts(addTick(\rho))$ [property of *addOuts*, *addTick*, and $\lesssim$]

$\Rightarrow \exists\, \rho_1 : iott^O_M[\![tt[\![P]\!]]\!] \bullet \rho \lesssim \rho_1$ [predicate calculus]

$= \rho \in \{\rho : TTTrace \mid \exists\, \rho_1 : iott^O_M[\![tt[\![P]\!]]\!] \bullet \rho \lesssim \rho_1\}$ [property of set comprehension]

Case ($\Leftarrow$).

$\rho \in \{\rho : TTTrace \mid \exists\, \rho_1 : iott^O_M[\![tt[\![P]\!]]\!] \bullet \rho \lesssim \rho_1\}$

$= \exists\, \rho_1 : iott^O_M[\![tt[\![P]\!]]\!] \bullet \rho \lesssim \rho_1$ [property of set comprehension]

$\Rightarrow \exists\, \rho_1 : iott^O_M[\![tt[\![P]\!]]\!] \bullet addOuts(\rho) \lesssim addOuts(\rho_1)$ [property of *addOuts*]

$\Rightarrow \exists\, \rho_1 : iott^O_M[\![tt[\![P]\!]]\!] \bullet addOuts(\rho) \lesssim \rho_1$ [*addOuts* is the identity]

$\Rightarrow \exists\, \rho_1 : \{\rho_2 : \mathrm{ran}\ addTick \mid addOuts(\rho_2) \in tt[\![P]\!] \bullet addOuts(\rho_2)\} \bullet addOuts(\rho) \lesssim \rho_1$ [definition of $iott^O_M[\![\_]\!]$]

$\Rightarrow \exists\, \rho_1 : \{\rho_2 : TTTrace \mid addOuts(\rho_2) \in tt[\![P]\!] \bullet addOuts(\rho_2)\} \bullet addOuts(\rho) \lesssim \rho_1$ [property of sets]

$= \exists\, \rho_1, \rho_2 : TTTrace \bullet addOuts(\rho_2) \in tt[[P]] \land \rho_1 = addOuts(\rho_2) \land addOuts(\rho) \lesssim \rho_1$ [property of sets]

$\Rightarrow addOuts(\rho) \in tt[[P]]$ [$tt[[P]]$ is **TT1**]

$= \rho \in iott^O[[P]]$ [definition of $iott^O[[P]]$]

□

The model characterised by $iott^O[[P]]$ is more natural than that defined by $iott^O_M[[tt[[P]]]]$, since $iott^O[[P]]$ records all experiments that can be carried out in observing P. For reasoning, however, $iott^O_M[[TT]]$ can be useful because, when it is applied to a healthy set TT of traces, it keeps the traces in the range of *addTick* and *addOuts*. The next theorem establishes that input-output ✓-tock refinement can be characterised using $iott^O_M[[\_]]$.

Theorem 3.10. $P \sqsubseteq_{IOTT} Q \Leftrightarrow iott^O_M[[tt[[Q]]]] \subseteq iott^O_M[[tt[[P]]]]$

Proof. *Case (⇒).*

$P \sqsubseteq_{IOTT} Q$

$= iott^O[[Q]] \subseteq iott^O[[P]]$ [definition of $\sqsubseteq_{IOTT}$]

$\Rightarrow \{\rho : TTTrace \mid addOuts(\rho) \in tt[[Q]]\} \subseteq \{\rho : TTTrace \mid addOuts(\rho) \in tt[[P]]\}$ [definition of $iott^O[[Q]]$]

$\Rightarrow \{\rho : \mathrm{ran}\, addTick \mid addOuts(\rho) \in tt[[Q]] \bullet addOuts(\rho)\}$ [property of sets and function application]
$\quad \subseteq \{\rho : \mathrm{ran}\, addTick \mid addOuts(\rho) \in tt[[P]] \bullet addOuts(\rho)\}$

$\Rightarrow iott^O_M[[tt[[Q]]]] \subseteq iott^O_M[[tt[[P]]]]$ [definition of $iott^O_M[[\_]]$]

Case (⇐).

$iott^O_M[[tt[[Q]]]] \subseteq iott^O_M[[tt[[P]]]]$

$\Rightarrow \{\rho : TTTrace \mid \exists\, \rho_1 : iott^O_M[[tt[[Q]]]] \bullet \rho \leq_{RT} \rho_1\}$ [property of sets]
$\quad \subseteq \{\rho : TTTrace \mid \exists\, \rho_1 : iott^O_M[[tt[[P]]]] \bullet \rho \leq_{RT} \rho_1\}$

$\Rightarrow iott^O[[Q]] \subseteq iott^O[[P]]$ [Theorem 3.9]

$= P \sqsubseteq_{IOTT} Q$ [definition of $\sqsubseteq_{IOTT}$]

□

We use $iott^O_M[[tt[[P]]]]$ extensively in calculating, based on Definition 3.1, an input-output ✓-tock semantics for the CSP operators based on their definitions in [3]. This is the topic of the next section.

3.3 Operators and recursion

Using Definition 3.1, we can calculate the input-output ✓-tock traces of *tock*-CSP processes in terms of their ✓-tock traces. A summary of the definitions for all process operators is in Tables 1 and 2. The calculations are in [2]. The non-standard trace operators, such as *tocks*, *fTock*, and others, used in these definitions are in Appendix A.

The semantics of divergence, termination, deadlock, timestop, and delay are unaffected, since these constructs carry out no communication, and, therefore, have no added instabilities due to possible outputs. To illustrate the calculation of the sets $iott^O[[P]]$, we present below the result for a timestop $\mathbf{Stop}_U$, used to define deadlines in *tock*-CSP. The only traces of $\mathbf{Stop}_U$ are the empty trace and singletons $\langle ref\ X \rangle$ containing an arbitrary refusal X.

Table 1. $iott^{O}[[\_]]$ model of CSP processes

Process P	$iott^{O}[[P]]$
div	$\{\langle\rangle\}$
Skip	$\{\langle\rangle, \langle evt\ \checkmark\rangle\}$
Stop	$tocks\,\Sigma^{\checkmark} \cup \{\rho : tocks\,\Sigma^{\checkmark};\ X : \mathbb{P}\,\Sigma^{\checkmark} \bullet \rho \frown \langle ref\ X\rangle\}$
$\mathbf{Stop}_U$	$\{\langle\rangle\} \cup \{X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet \langle ref\ X\rangle\}$
Wait n	$\{\rho : tocks\,\Sigma^{\checkmark} \mid \#(\rho \upharpoonright \{evt\ tock\}) \leq n\}$ $\cup\{\rho : tocks\,\Sigma^{\checkmark};\ X : \mathbb{P}\,\Sigma^{\checkmark} \mid \#(\rho \upharpoonright \{evt\ tock\}) < n \bullet \rho \frown \langle ref\ X\rangle\}$ $\cup\{\rho : tocks\,\Sigma^{\checkmark} \mid \#(\rho \upharpoonright \{evt\ tock\}) = n \bullet \rho \frown \langle evt\ \checkmark\rangle\}\}$
$e \rightarrow P$	$\{\rho : TTTrace \mid e \notin O \wedge \rho \in tocks\,(\Sigma^{\checkmark} \setminus \{e\})\}$ $\cup\{\rho : tocks\,(\Sigma^{\checkmark} \setminus \{e\});\ X : \mathbb{P}(\Sigma^{\checkmark} \setminus \{e\}) \mid e \notin O \bullet \rho \frown \langle ref\ X\rangle\}$ $\cup\{\rho_1 : tocks\,(\Sigma^{\checkmark} \setminus \{e\});\ \rho_2 : iott^{O}[[P]] \mid e \notin O \wedge e \neq tock \bullet \rho_1 \frown \langle evt\ e\rangle \frown \rho_2\}$ $\cup\{\rho : iott^{O}[[P]] \mid e \in O \bullet \langle evt\ e\rangle \frown \rho\}$ $\cup\{\rho_1 : tocks\,\Sigma^{\checkmark};\ X : \mathbb{P}\,\Sigma^{\checkmark};\ \rho_2 : iott^{O}[[P]] \mid e = tock \bullet \rho_1 \frown \langle ref\ X, evt\ tock\rangle \frown \rho_2\}$
$P \sqcap Q$	$iott^{O}[[P]] \cup iott^{O}[[Q]]$
$P \,\Box\, Q$	$\{\rho_1 : tocks\,\Sigma^{\checkmark}_{tock};\ \rho_2, \rho_3, \rho_4 : TTTrace \mid$ $\rho_1 \frown \rho_2 \in iott^{O}[[P]] \wedge \rho_1 \frown \rho_3 \in iott^{O}[[Q]] \wedge$ $(\forall \rho_5 : tocks\,\Sigma^{\checkmark}_{tock} \bullet \rho_5 \lesssim \rho_1 \frown \rho_2 \Rightarrow \rho_5 \lesssim \rho_1) \wedge$ $(\forall \rho_5 : tocks\,\Sigma^{\checkmark}_{tock} \bullet \rho_5 \lesssim \rho_1 \frown \rho_3 \Rightarrow \rho_5 \lesssim \rho_1) \wedge$ $(\forall X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \mid \rho_2 = \langle ref\ X\rangle \bullet (\exists Y : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet \rho_3 = \langle ref\ Y\rangle \wedge X \setminus \{tock\} = Y \setminus \{tock\})) \wedge$ $(\forall X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \mid \rho_3 = \langle ref\ X\rangle \bullet (\exists Y : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet \rho_2 = \langle ref\ Y\rangle \wedge X \setminus \{tock\} = Y \setminus \{tock\})) \wedge$ $(\rho_4 = \rho_1 \frown \rho_2 \vee \rho_4 = \rho_1 \frown \rho_3)$ $\bullet\ \rho_4$ $\}$
$P;\ Q$	$\{\rho_1 : iott^{O}[[P]] \mid \neg\,(\exists \rho_2 : TTTrace \bullet \rho_1 = \rho_2 \frown \langle evt\ \checkmark\rangle)\}$ $\cup\{\rho_1, \rho_2 : TTTrace \mid \rho_1 \frown \langle evt\ \checkmark\rangle \in iott^{O}[[P]] \wedge \rho_2 \in iott^{O}[[Q]] \bullet \rho_1 \frown \rho_2\}$

Theorem 3.11. $iott^{O}[[\mathbf{Stop}_U]] = \{\langle\rangle\} \cup \{X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet \langle ref\ X\rangle\}$

Proof.

$iott^{O}[[\mathbf{Stop}_U]]$

$= \{\rho : TTTrace \mid addOuts(\rho) \in \{\langle\rangle\} \cup \{X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet \langle ref\ X\rangle\}\}$ [definitions of $iott^{O}[[\mathbf{Stop}_U]]$ and $tt[[\mathbf{Stop}_U]]$]

$= \{\rho : TTTrace \mid addOuts(\rho) = \langle\rangle \vee (\exists X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet addOuts(\rho) = \langle ref\ X\rangle)\}$ [property of sets]

$= \{\rho : TTTrace \mid \rho = \langle\rangle \vee (\exists X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet \rho = \langle ref\ X\rangle)\}$ [definition of $addOuts$ and $O \subseteq \Sigma^{\checkmark}_{tock}$]

$= \{\langle\rangle\} \cup \{X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet \langle ref\ X\rangle\}$ [property of sets]

□

Table 2. $iott^O[[\_]]$ model of CSP processes - continuation

Process P	$iott^O[[P]]$
$P \triangle Q$	$\{\rho_1 : TTTrace;\ \rho_2 : iott^O[[Q]] \mid \rho_1 \frown \langle evt\ \checkmark\rangle \in iott^O[[P]] \land fTock\,\rho_1 = \rho_2 \bullet \rho_1 \frown \langle evt\ \checkmark\rangle\}$ $\cup\ \{\rho_1, \rho_2 : TTTrace;\ X, Y, Z : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \mid$ $\rho_1 \frown \langle ref\ X\rangle \in iott^O[[P]] \land \rho_2 \frown \langle ref\ Y\rangle \in iott^O[[Q]] \land$ $fTock\,\rho_1 = \rho_2 \land Z \subseteq X \cup Y \land X \setminus \{tock\} = Y \setminus \{tock\} \land$ $\bullet\ \rho_1 \frown \langle ref\ Z\rangle$ $\}$ $\cup\ \{\rho_1 : iott^O[[P]];\ \rho_2, \rho_3 : TTTrace \mid$ $(\neg\ \exists \phi : \text{seq}\ Obs \bullet \rho_1 = \phi \frown \langle evt\ \checkmark\rangle) \land (\neg\ \exists \phi : \text{seq}\ Obs;\ X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet \rho_1 = \phi \frown \langle ref\ X\rangle)$ $\land$ $fTock\,\rho_1 = \rho_2 \land \rho_2 \frown \rho_3 \in iott^O[[Q]] \land (\neg\ \exists \phi : \text{seq}\ Obs;\ X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet \rho_1 = \langle ref\ X\rangle \frown \phi)$ $\bullet\ \rho_1 \frown \rho_3$ $\}$
$P \triangle_d Q$	$\{\rho_1 : iott^O[[P]] \mid \#(\rho_1 \upharpoonright \{evt\ tock\}) < d\}$ $\cup\ \{\rho_1 : iott^O[[P]];\ \rho_2 : iott^O[[Q]];\ \phi : \text{seq}\ Obs \mid$ $\#(\rho_1 \upharpoonright \{evt\ tock\}) = d \land ((d = 0 \land \rho_1 = \langle\rangle) \lor (d > 0 \land \rho_1 = \phi \frown \langle evt\ tock\rangle)) \bullet \rho_1 \frown \rho_2$ $\}$
$P \,[\![X]\!]\, Q$	$\bigcup\{\rho_1 : iott^O[[P]];\ \rho_2 : iott^O[[Q]] \bullet (\rho_1 \,[\![X]\!]^T\, \rho_2)\}$
$P \setminus X$	$\bigcup\{\rho : iott^O[[P]] \bullet hideTrace\,X\,\rho\}$
$P[[f]]$	$\bigcup\{\rho : iott^O[[P]] \bullet renameTrace\,f\,\rho\}$

If e is not an output ($e \notin O$) or if e is *tock*, a prefixing $e \rightarrow P$, like in the original semantics, contributes traces with e as well as events *tock* and their associated refusals. If, however, e is an output, the only traces that we can have are of the form $\langle e\rangle \frown \rho$, where ρ is an input-output trace of P. This reflects the fact that an output is unstable and so urgent.

Example 3.12. The traces in $iott^O[[takeoff \rightarrow \mathbf{Wait}(1)]]$ are the prefixes of $\langle takeoff, \Sigma^{\checkmark}, tock, \checkmark\rangle$. □

Internal and external choice, sequence, interrupt, timeout, and hiding are unaffected by the presence of input and outputs. For further illustration, we include below the calculation of the traces of a timeout $P \triangle_d Q$, which behaves like the process P, until d time units have passed, when Q takes over. We recall that all calculations are in [2]. In words, the traces of a timeout $P \triangle_d Q$ include those of P (that is, from $iott^O[[P]]$) for which the number of *tock* events is less than d. For a sequence ϕ, the filtering $\phi \upharpoonright S$ defines the sequence obtained from ϕ by keeping just the elements in the set S, and $\#\phi$ is the number of elements of ϕ. Additionally, $P \triangle_d Q$ has traces formed from traces ρ_1 of P with exactly d occurrences of *tock* followed by a trace ρ_2 of Q. If d is 0, then ρ_1 must be the empty trace, since Q takes over immediately, before any events of P take place. If d is greater than 0, then ρ_1 finishes on the last *tock* event, because, again, when the deadline is over, Q starts immediately before P can engage in any more events.

Theorem 3.13.

$$
\begin{aligned}
iott^{\mathcal{O}}[\![P \vartriangle_d Q]\!] = \; & \{\rho_1 : iott^{\mathcal{O}}[\![P]\!] \mid \#(\rho_1 \upharpoonright \{evt\ tock\}) < d\} \cup \\
& \{\rho_1 : iott^{\mathcal{O}}[\![P]\!];\ \rho_2 : iott^{\mathcal{O}}[\![Q]\!];\ \phi : \mathrm{seq}\ Obs \mid \\
& \quad \#(\rho_1 \upharpoonright \{evt\ tock\}) = d \wedge ((d = 0 \wedge \rho_1 = \langle\rangle) \vee (d > 0 \wedge \rho_1 = \phi \frown \langle evt\ tock\rangle)) \bullet \rho_1 \frown \rho_2 \\
& \}
\end{aligned}
$$

Proof. We rely here on Theorem 3.10.

$$
\begin{aligned}
& iott^{\mathcal{O}}_M[\![tt[\![P \vartriangle_d Q]\!]]\!] \\
& = iott^{\mathcal{O}}_M[\![\quad \{\rho_2 : tt[\![P]\!] \mid \#(\rho_2 \upharpoonright \{evt\ tock\}) < d\} \cup && \text{[definition of } tt[\![P \vartriangle_d Q]\!]\text{]} \\
& \qquad \{\rho_2 : tt[\![P]\!];\ \rho_3 : tt[\![Q]\!];\ \phi : \mathrm{seq}\ Obs \mid \\
& \qquad\quad \#(\rho_2 \upharpoonright \{evt\ tock\}) = d \wedge ((d = 0 \wedge \rho_2 = \langle\rangle) \vee (d > 0 \wedge \rho_2 = \phi \frown \langle evt\ tock\rangle)) \bullet \rho_2 \frown \rho_3 \\
& \qquad \} \ ]\!] \\
& = \{\rho_1 : \mathrm{ran}\ addTick \mid addOuts(\rho_1) \in && \text{[definition of } iott^{\mathcal{O}}[\![\_]\!]\text{]} \\
& \qquad \{\rho_2 : tt[\![P]\!] \mid \#(\rho_2 \upharpoonright \{evt\ tock\}) < d\} \cup \\
& \qquad \{\rho_2 : tt[\![P]\!];\ \rho_3 : tt[\![Q]\!];\ \phi : \mathrm{seq}\ Obs \mid \\
& \qquad\quad \#(\rho_2 \upharpoonright \{evt\ tock\}) = d \wedge ((d = 0 \wedge \rho_2 = \langle\rangle) \vee (d > 0 \wedge \rho_2 = \phi \frown \langle evt\ tock\rangle)) \bullet \rho_2 \frown \rho_3 \\
& \qquad \} \\
& \quad \bullet addOuts(\rho_1) \\
& \} \\
& = \{\rho_1 : \mathrm{ran}\ addTick \mid && \text{[property of sets]} \\
& \qquad addOuts(\rho_1) \in tt[\![P]\!] \wedge \#(addOuts(\rho_1) \upharpoonright \{evt\ tock\}) < d \ \vee \\
& \qquad \exists \rho_2 : tt[\![P]\!];\ \rho_3 : tt[\![Q]\!];\ \phi : \mathrm{seq}\ Obs \bullet \\
& \qquad\quad \#(\rho_2 \upharpoonright \{evt\ tock\}) = d \wedge ((d = 0 \wedge \rho_2 = \langle\rangle) \vee (d > 0 \wedge \rho_2 = \phi \frown \langle evt\ tock\rangle)) \wedge \\
& \qquad\quad addOuts(\rho_1) = \rho_2 \frown \rho_3 \\
& \quad \bullet addOuts(\rho_1) \\
& \} \\
& = \{\rho_1 : \mathrm{ran}\ addTick \mid && \text{[idempotence of } addOuts \text{ and } \rho_2 \text{ and } \rho_3 \text{ in ran } addOuts\text{]} \\
& \qquad addOuts(addOuts(\rho_1)) \in tt[\![P]\!] \wedge \#(addOuts(\rho_1) \upharpoonright \{evt\ tock\}) < d \ \vee \\
& \qquad \exists \rho_2 : TTTrace;\ \rho_3 : TTTrace;\ \phi : \mathrm{seq}\ Obs \bullet \\
& \qquad\quad addOuts(\rho_2) \in tt[\![P]\!] \wedge addOuts(\rho_3) \in tt[\![Q]\!] \wedge \\
& \qquad\quad \#(\rho_2 \upharpoonright \{evt\ tock\}) = d \wedge ((d = 0 \wedge \rho_2 = \langle\rangle) \vee (d > 0 \wedge \rho_2 = \phi \frown \langle evt\ tock\rangle)) \wedge \\
& \qquad\quad addOuts(\rho_1) = \rho_2 \frown \rho_3 \\
& \quad \bullet addOuts(\rho_1) \\
& \}
\end{aligned}
$$

$$
\begin{array}{ll}
= & \{\, \rho_1 : \mathrm{ran}\ addTick \mid \\
& \quad addOuts(\rho_1) \in iott^O[\![P]\!] \land \#(addOuts(\rho_1) \upharpoonright \{evt\ tock\}) < d \ \lor \\
& \quad \exists \rho_2 : iott^O[\![P]\!];\ \rho_3 : iott^O[\![Q]\!];\ \phi : \mathrm{seq}\ Obs \bullet \\
& \qquad \#(\rho_2 \upharpoonright \{evt\ tock\}) = d \land ((d = 0 \land \rho_2 = \langle\rangle) \lor (d > 0 \land \rho_2 = \phi \frown \langle evt\ tock\rangle)) \land \\
& \qquad addOuts(\rho_1) = \rho_2 \frown \rho_3 \\
& \quad \bullet\ addOuts(\rho_1) \\
& \}
\end{array}
\qquad [\text{definition of } iott^O[\![\_]\!]]
$$

$$
\begin{array}{ll}
= & \{\, \rho_1 : \mathrm{ran}\ addTick \mid addOuts(\rho_1) \in \\
& \quad \{\rho_2 : iott^O[\![P]\!] \mid \#(\rho_2 \upharpoonright \{evt\ tock\}) < d\} \cup \\
& \quad \{\rho_2 : iott^O[\![P]\!];\ \rho_3 : iott^O[\![Q]\!];\ \phi : \mathrm{seq}\ Obs \mid \\
& \qquad \#(\rho_2 \upharpoonright \{evt\ tock\}) = d \land ((d = 0 \land \rho_2 = \langle\rangle) \lor (d > 0 \land \rho_2 = \phi \frown \langle evt\ tock\rangle)) \bullet \rho_2 \frown \rho_3 \\
& \quad \} \\
& \quad \bullet\ addOuts(\rho_1) \\
& \}
\end{array}
\qquad [\text{properties of sets}]
$$

$$
\begin{array}{ll}
= & iott^O_M[\![\ \ \{\rho_2 : iott^O[\![P]\!] \mid \#(\rho_2 \upharpoonright \{evt\ tock\}) < d\} \cup \\
& \qquad \{\rho_2 : iott^O[\![P]\!];\ \rho_3 : iott^O[\![Q]\!];\ \phi : \mathrm{seq}\ Obs \mid \\
& \qquad\quad \#(\rho_2 \upharpoonright \{evt\ tock\}) = d \land ((d = 0 \land \rho_1 = \langle\rangle) \lor (d > 0 \land \rho_2 = \phi \frown \langle evt\ tock\rangle)) \bullet \rho_2 \frown \rho_3 \\
& \qquad \}\ \ ]\!]
\end{array}
\qquad [\text{definition of } iott^O_M[\![\_]\!]]
$$

□

Uses of parallelism and renaming have to satisfy well-formedness conditions to ensure that we can define a congruence to give their semantics. Below, we define and justify these conditions.

We have to restrict the use of the parallel operator $P \,[\![\, X \,]\!]\, Q$ to forbid synchronisation on outputs. $P \,[\![\, X \,]\!]\, Q$ defines a process whose behaviour is characterised by the parallel execution of the processes P and Q synchronising on events in the set X. If an event e is an input in P and Q, then it is an input in the parallelism; there is no issue. If e is an output in P and Q, then it is an output in the parallelism. We then require that e is not in X.

Example 3.14. We consider $E1 = out1 \rightarrow \mathbf{Stop} \sqcap out2 \rightarrow \mathbf{Stop}$ and $E2 = out1 \rightarrow \mathbf{Stop} \,\Box\, out2 \rightarrow \mathbf{Stop}$. If $out1$ and $out2$ are outputs, $E1$ and $E2$ have the same input-output refusal traces. This is because, since their initial states are unstable, all their traces start with an event. We cannot differentiate the forms of choice due to lack of stability. We, therefore, expect that $E3 = out1 \rightarrow \mathbf{Stop} \,[\![\, \{out1, out2\} \,]\!]\, E1$ and $E4 = out1 \rightarrow \mathbf{Stop} \,[\![\, \{out1, out2\} \,]\!]\, E2$ also have the same traces. In $E3$, however, we have a possible stability: if $E1$ resolves the choice to $out2 \rightarrow \mathbf{Stop}$, then we have a deadlock. In this case, the traces are all those whose events are *tock* and whose refusals are subsets of $\Sigma^{\checkmark}$ (see semantics of **Stop** in Table 1). The same stability, however, is not possible for $E4$. □

So, we define that in a well formed parallelism, X does not include outputs.

A process $P[[f]]$, defined in terms of a process P by renaming in accordance to a function f from events to events,

behaves as P, except that every occurrence of an event e in P is replaced with the event $f(e)$.

Example 3.15.

$$RD[[id \oplus \{turnoff \mapsto off, move \mapsto mv\}]] = takeoff \rightarrow \mathbf{Wait}\ 1;\ mv \rightarrow found \rightarrow land \rightarrow \mathbf{Stop} \,\Box\, off \rightarrow \mathbf{Skip}$$

Here, id is the identity function on events, and $f \oplus g$ is the overriding operator that defines the function that maps x to $g(x)$, if x is in the domain of g, and, otherwise, maps x to $f(x)$. So the renaming function in this example maps all events to themselves, expect for *turnoff* and *move*, which are mapped to *off* and mv in the function $\{turnoff \mapsto off, move \mapsto mv\}$. □

In *tock*-CSP, there is an assumption that the renaming function f is total, and *tock* and ✓ are not renamed. (Considering relational renaming is a straightforward generalisation.) For our model, we assume in addition that outputs are renamed to outputs, and, therefore, inputs to inputs. In this way, the instabilities arising from outputs of $P[[f]]$ are the same as those of P, and we can indeed define the semantics of $P[[f]]$ in terms of that of P.

The function *renameTrace* $f\ \rho$ used in the definition of the semantics of $P[[f]]$ (see Table 2 and Appendix A) applies the renaming function f to each event of ρ. For the refusals X in ρ, *renameTrace* $f\ \rho$ identifies all sets of events Y, such that, via renaming of its events, we obtain X (and there may be more than one such set Y if f is not injective).

Example 3.16. In the renaming below, two input events $inp1$ and $inp2$ are renamed to a single input event inp.

$$E1 = out \rightarrow \mathbf{Wait}\ 1;\ (inp1 \rightarrow \mathbf{Skip} \,\Box\, inp2 \rightarrow \mathbf{Skip}) \quad \text{and} \quad E2 = E1[[id \oplus \{inp1 \mapsto inp, inp2 \mapsto inp\}]]$$

As it might be expected, the process described by this renaming can be defined as $out \rightarrow \mathbf{Wait}\ 1;\ inp \rightarrow \mathbf{Skip}$. Accordingly, $E1$ has the following trace: $\langle out, \Sigma^{\checkmark}, tock, \{out, \checkmark\}, tock, inp1, \checkmark\rangle$. In this scenario, the input $inp1$ takes place after one additional time unit following the **Wait** 1. A trace of the renamed process $E2$ is $\langle out, \Sigma^{\checkmark}, tock, \{out, \checkmark\}, tock, inp, \checkmark\rangle$. We note that the renaming function maps *out* and ✓ to themselves. Another trace includes, instead of $\Sigma^{\checkmark}$, the refusal $\{inp, out, tock\}$ because applying relational image of $\{inp, out, \checkmark\}$ through the inverse of the renaming function gives $\Sigma^{\checkmark}$. The inverse of the renaming function maps inp to itself, and to $inp1$ and $inp2$. □

A recursive process $P = F(P)$ is defined by a function F from processes to processes described using the process operators of *tock*-CSP. Its semantics of P is given by the greatest (with respect to $\sqsubseteq_{IOTT}$) fixed point of F, given by $iott^O[[P]] = \bigcup\{n : \mathbb{N} \bullet F^n(iott^O[[\mathbf{div}]])\}$, where F^n is defined by the repeated application of F n times.

Given our model for *tock*-CSP with inputs and outputs, we next compare its refinement relation to tioco.

4 INPUT OUTPUT LABELLED TRANSITION SYSTEMS (IOLTS)

Most works on testing from a formal state-based model reason about a labelled transition system that represents the operational semantics of the original model. In addition, it is normal to distinguish between input and output events, since, as said, they play very different roles in testing, with the tester controlling inputs and the SUT controlling outputs. Typically, the name of an input starts with '?' and that of an output starts with '!'. In addition, it is often assumed that the tester cannot block outputs, and the implementation cannot block inputs.

A labelled transition system whose events are partitioned into inputs and outputs is called an input-output labelled transition system (IOLTS) [53]. An IOLTS M can be represented by a tuple (I, O, Q, q_0, h) in which I is the set of input events, O is the set of output events, Q is the set of states, $q_0 \in Q$ is the initial state, and h is the transition relation of type $Q \times (I \cup O \cup \{\tau\}) \times Q$. Here, τ represents a silent (internal) event. If $(q_1, e, q_2) \in h$ then (q_1, q_2) is a transition of

M, denoting it is possible for M to move from state q_1 to state q_2 with event e (if $e \in I \cup O$) or without any event being observed (if $e = \tau$). There has been significant interest in testing using an IOLTS [4, 6, 19, 20, 28, 39, 50–55].

Timed IOLTS (TIOLTS) extend IOLTS by allowing transitions that denote durations from a set D, capturing the passing of (typically discrete) time [53]. These events, which represent durations, are not inputs to the system (since they are not controlled by the tester) and also are not outputs (since they are not controlled by the SUT). Sometimes, it might also be possible for a tester to observe quiescence, that is, the SUT being in a state where it cannot produce an output or change state without receiving an input. Quiescence is usually represented by a new symbol δ, where $\delta \notin I \cup O \cup D \cup \{\tau\}$. Quiescence is a special type of refusal, which represents the refusal of all outputs, and is normally observed through a timeout. A tester observes *suspension traces*, which are sequences of events in either $I \cup O \cup D$ (if quiescence cannot be observed) or $I \cup O \cup D \cup \{\delta\}$ (if quiescence can be observed).

An IOLTS is input-enabled if for each state q and input $?i$, there is at least one state q' that can be reached from q through a sequence of internal transitions such that there is a transition of the form $(q', ?i, q'')$. If an IOLTS is input-enabled, then it is an input- output transition system (IOTS). This definition extends naturally to TIOLTS and TIOTS; we adopt the same definition and do not allow a transition representing passage of time to be considered an internal transition. Frequently, works on testing from an TIOLTS assume that the SUT is input-enabled but the specification does not have to be input-enabled [53]. As a result, testing can be seen as a procedure in which one is testing an implementation that behaves like an unknown TIOLTS N, where N has the same input and output sets as the specification TIOLTS M. Most works also assume that the processes are divergence free since, normally, testing cannot distinguish between divergence and deadlock. In this paper we assume that processes are divergence free but implementations do not have to be input-enabled, allowing the development of a more general testing theory that can be applied to systems where, for example, sensors can be disabled.

When testing from an IOLTS, a popular conformance relation is ioco [51–53], under which, if σ is a suspension trace of the specification and σ occurs in testing of the SUT, then any next event (output or quiescence) produced by the SUT must be one allowed by the specification (after σ). As mentioned, the ioco relation was extended to timed ioco (tioco). We define below the Schmaltz and Tretmans version of tioco, which uses the following notation for a TIOLTS M.

- Given a suspension trace σ, the suspension traces of TIOLTS M *after* σ are those that can be produced by M after σ. We therefore have that σ_1 is a suspension trace of M *after* σ if, and only if, $\sigma \frown \sigma_1$ is a suspension trace of M.
- $Out(M)$ is the set of observations from $O \cup D \cup \{\delta\}$ that M can initially perform. As a result, $e \in O \cup D \cup \{\delta\}$ is in $Out(M)$ if, and only if, $\langle e \rangle$ is a suspension trace of M.

We can now define tioco [45].

Definition 4.1. Given TIOLTS M and TIOTS N with the same sets of inputs and outputs, N conforms to M under tioco if, and only if, for all σ, if σ is a suspension trace of M then $Out(N \text{ after } \sigma) \subseteq Out(M \text{ after } \sigma)$.

This is the notion of tioco that we compare to refinement in our input-output model for *tock*-CSP presented in the previous section. We note that although the focus of this paper is on testing from models with discrete time, tioco can also be used for modelling and reasoning about continuous time.

As an aside, we observe that tioco is rather different from trace inclusion because of the way in which it handles inputs that are not enabled in the specification. Under trace inclusion, if the specification has a trace ϕ and, for input $?i$, does not have a trace $\phi \frown \langle ?i \rangle$, then a correct implementation is not allowed to have the trace $\phi \frown \langle ?i \rangle$. In contrast, under tioco, if the specification has a suspension trace σ, but not $\sigma \frown \langle ?i \rangle$, not only is $\sigma \frown \langle ?i \rangle$ an allowed behaviour of an implementation but, in addition, all behaviours are allowed after $\sigma \frown \langle ?i \rangle$.

5 INPUT-OUTPUT *tock*-CSP REFINEMENT AND IOCO WITH TIME

In this section, we define suspension traces for processes P using $iott^O[\![P]\!]$ to characterise tioco in the context of CSP. With that, we establish that $\sqsubseteq_{IOTT}$ is stronger than Schmaltz and Tretmans tioco (Definition 4.1).

Precisely, in this section we show that if $P \sqsubseteq_{IOTT} Q$, then Q conforms to P under tioco (Theorem 5.9). Moreover, we show that there are processes P and Q related by tioco for which $P \sqsubseteq_{IOTT} Q$ does not hold (Theorem 5.10). Together, these results establish that $\sqsubseteq_{IOTT}$ is strictly stronger than tioco for input-enabled implementations. We restrict here attention to input-enabled implementations because tioco is only defined for such implementations. Input-output ✓-tock refinement does not have such a restriction as stated in Definition 3.6.

To define a suspension-traces model for CSP, we consider the set $Strace^O$ of valid suspension traces for output events in O whose definition from [17] we reproduce below. Here, the set $\Sigma^\delta = \Sigma^\checkmark_{tock} \cup \{\delta\}$ of events in scope includes an extra special event δ that represents quiescence like the observation of the same name in an a TIOLTS.

Definition 5.1.

$$Strace^O == \{\sigma : \operatorname{seq} \Sigma^\delta \mid \forall\, i : 1 \,..\, \#\sigma - 1 \bullet \sigma\, i = \delta \Rightarrow \sigma\,(i+1) \notin O\}$$

This set includes the sequences σ of events in $\Sigma^\checkmark_{tock}$ and δ, such that, a δ is never followed by an output. This is required because, if we can observe stability, recorded by δ, then an output cannot be possible. As shown in [15], to study inclusion of sets of suspension traces, it is enough to consider sets ST that satisfy the healthiness condition **ST** below.

ST $\quad \sigma \in ST \Rightarrow \neg\, (\langle \delta, \delta \rangle) \text{ in } \sigma$

We write σ_1 in σ_2 when the sequence σ_1 occurs contiguously in the sequence σ_2. In the context of ioco and tioco, it is normal to allow the recording of quiescence to be repeated. Here, the subsequence $\langle \delta, \delta \rangle$ essentially provides the same information as the subsequence δ; both simply denote the process being in a state where it cannot produce output or change state without first receiving input. **ST** ensures that such redundant records are not included.

We now define a function st that characterises a suspension trace corresponding to a ✓-tock trace.

Definition 5.2.

$$st : TTTrace \to Strace^O$$

$$\begin{aligned}
&\forall\, e : \Sigma^\checkmark_{tock};\ X : \mathbb{P}\, \Sigma^\checkmark_{tock};\ \rho : TTTrace \bullet \\
&\quad st\, \langle\rangle = \langle\rangle \land st\, (\langle e \mathrel{vt} e \rangle \frown \rho) = \langle e \rangle \frown st\, \rho \\
&\quad st\, (\langle ref\ X \rangle \frown \rho) = \langle\rangle \land \neg\, (O \cup \{\checkmark\} \subseteq X) \lor st\, (\langle ref\ X \rangle \frown \rho) = \langle \delta \rangle \frown st\, \rho \land O \cup \{\checkmark\} \subseteq X
\end{aligned}$$

This function removes refusals that do not include all outputs and ✓ and replaces any refusal that contains all outputs and ✓ by δ. We observe that, by **TT3**, st could be defined to replace a refusal that contains all outputs by δ (that is, not require that the refusal contains ✓ as well). We would indeed obtain the same set of suspension traces when the function is applied to the traces of a healthy set. The above definition, however, slightly simplifies some proofs.

It is worth briefly commenting on $st\, (\langle ref\ X \rangle \frown \rho)$ when $\neg\, (O \cup \{\checkmark\} \subseteq X)$. Here, the refusal X does not establish stability under the input-output model since either ✓ or some outputs are not in X and so may be enabled. As a result, $\langle ref\ X \rangle \frown \rho$ does not correspond to a suspension trace. If it happens to be the case that a process is stable, then it has another trace $\langle ref\ (X \cup O \cup \{\checkmark\}) \rangle \frown \rho$ for which we obtain a suspension trace that records its stability and events.

We now define the set of timed suspension traces of a process.

Definition 5.3. $tstraces[[P]] \mathrel{\widehat=} st(\!| \ iott^O[[P]] \ |\!)$.

Here, $R(\!| \ S \ |\!)$ is the relational image of a set S through the relation (or function, in particular) R.

The following lemma establishes that every set of suspension traces defined by $tstraces[[\_]]$ is healthy. Omitted proofs of this and other results presented in this section can be found in [2].

Lemma 5.4. *$tstraces[[P]]$ is **ST**-healthy.*

The traces in $tstraces[[P]]$ also satisfy another property: δ is followed by *tock*. This strengthens the normal requirement that quiescence cannot be followed by an output, and holds for every trace σ characterised by an application of st.

Lemma 5.5. *For every ρ in $TTTrace$, for every $i : 1 \mathinner{..} \#(st\ \rho) - 1$, if $(st\ \rho)\ i = \delta$ then $(st\ \rho)\ (i + 1) = tock$.*

We call a trace that is in the range of st, and therefore satisfies the above property, a timed suspension trace.

The Schmaltz and Tretmans version of tioco can be expressed as follows in terms of timed suspension traces.

$$Q \text{ tioco } P \mathrel{\widehat=} \forall \sigma : tstraces[[P]] \bullet Out(Q \ after \ \sigma) \subseteq Out(P \ after \ \sigma)$$

Here, for a process P, $Out(P)$ denotes the set of events from $O \cup \{tock, \delta\}$ that start a timed suspension trace of P. In addition, $P \ after \ \sigma$ is the process whose timed suspension traces σ_1 are such that $\sigma \frown \sigma_1$ is a timed suspension trace of P. By a property of $\subseteq$, the above definition of tioco can be rewritten to the following.

$$Q \text{ tioco } P \mathrel{\widehat=} \forall \sigma : tstraces[[P]];\ e : O \cup \{\delta, tock\} \bullet e \in Out(Q \ after \ \sigma) \Rightarrow e \in Out(P \ after \ \sigma)$$

For a process R, we have that e is in $Out(R \ after \ \sigma)$, if, and only if, R can move via σ to a state in which e can be observed. This is the case if, and only if, $\sigma \frown \langle e \rangle \in tstraces[[R]]$. So, $e \in Out(R \ after \ \sigma)$ if, and only if, $\sigma \frown \langle e \rangle \in tstraces[[R]]$. Based on this, we obtain the definition below of Q tioco P in terms of $tstraces[[P]]$ and $tstraces[[Q]]$.

Definition 5.6. For an arbitrary process P and an input-enabled process Q,

$$Q \text{ tioco } P \mathrel{\widehat=} \forall \sigma : tstraces[[P]];\ e : O \cup \{\delta, tock\} \bullet \sigma \frown \langle e \rangle \in tstraces[[Q]] \Rightarrow \sigma \frown \langle e \rangle \in tstraces[[P]]$$

Timed suspension traces that are **ST**-healthy, and so do not contain $\langle \delta, \delta \rangle$ as a subsequence, are similar to input-output ✓-tock traces. We now define, for a timed suspension trace σ, the corresponding ✓-tock trace $tt(\sigma)$.

$$tt : Strace^O \rightarrow TTTrace$$

$$\forall a : \Sigma;\ \sigma : Strace^O \bullet$$
$$tt(\langle\rangle) = \langle\rangle \land tt(\langle e \rangle \frown \sigma) = \langle evt\ e \rangle \frown tt(\sigma) \land tt(\langle \delta \rangle \frown \sigma) = \langle ref\ (O \cup \{\checkmark\}) \rangle \frown tt(\sigma)$$

This simply replaces each occurrence of δ with the refusal set $O \cup \{\checkmark\}$. We now establish that tt and st are related as expected: they form a Galois connection between $TTTraces$, ordered by $\lesssim$, and timed suspension traces with equality.

Theorem 5.7. $st(tt(\sigma)) = \sigma \quad and \quad tt(st(\rho)) \lesssim \rho$

For a ✓-tock trace ρ, $tt(st(\rho))$ need not be the same as ρ, because the application of st to ρ removes any refusals that do not contain O and ✓ and also all information regarding the refusal of inputs.

Interestingly, $iott^O[[P]]$ and $iott^O_M[[P]]$ define the same sets of suspension traces through st.

Theorem 5.8. $st(\!|\ iott^O[[P]]\ |\!) = st(\!|\ iott^O_M[[tt[[P]]]]\ |\!)$

We now show that input-output ✓-tock refinement implies tioco.

Theorem 5.9. *Given processes P and Q such that Q is input-enabled, $P \sqsubseteq_{IOTT} Q \Rightarrow Q$ tioco P.*

Proof.

$P \sqsubseteq_{IOTT} Q$

$\Rightarrow iott^O[[Q]] \subseteq iott^O[[P]]$ [definition of $\sqsubseteq_{IOTT}$]

$\Rightarrow st(\!|\ iott^O[[Q]]\ |\!) \subseteq st(\!|\ iott^O[[P]]\ |\!)$ [property of relational image]

$\Rightarrow tstraces[[Q]] \subseteq tstraces[[P]]$ [definition of $tstraces[[\_]]$]

$\Rightarrow \forall \sigma' : Straces \bullet \sigma' \in tstraces[[Q]] \Rightarrow \sigma' \in tstraces[[P]]$ [property of set inclusion]

$\Rightarrow \forall \sigma : tstraces[[P]];\ e : O \cup \{\delta, tock\} \bullet \sigma \frown \langle e \rangle \in tstraces[[Q]] \Rightarrow \sigma \frown \langle e \rangle \in tstraces[[P]]$

[substitution of $\sigma \frown \langle e \rangle$ for σ']

$\Rightarrow Q$ tioco P [definition of tioco]

□

The proof above does not use the hypothesis that Q is input-enabled explicitly, but it is needed because tioco is defined only for input-enabled implementations. Below we show that, even for these implementations, tioco is weaker.

Theorem 5.10. *There are P and Q such that Q tioco P, but not $P \sqsubseteq_{IOTT} Q$.*

Proof. As an example, we can take as the specification P the process $\mathbf{Stop}_U$ and let Q be any input-enabled implementation that can produce an output *out* in response to some urgent input *in* but that cannot produce an output before first receiving an input. For example, we can have $Q = (in \rightarrow \mathbf{Skip}) \blacktriangleright 0;\ out \rightarrow \ldots$, where we impose a deadline 0 on *in*, to make it urgent, before allowing the output *out*. Any input-enabled process can follow *out*. The set $tstraces[[\mathbf{Stop}_U]]$ includes $\langle\rangle$ and $\langle\delta\rangle$. Under tioco we only need to consider the behaviour of Q after the empty sequence, because there are no traces of Q of the form $\langle\delta\rangle \frown \langle e\rangle$ (see Definition 4.1). So, no restrictions arise from tioco for $\sigma = \langle\delta\rangle$. In addition, under tioco we only need to consider the outputs of Q, *tock*, and quiescence after the empty sequence. In our example, there is no such event, because the input of Q is urgent. So, we have that Q tioco P as required. However, $P \sqsubseteq_{IOTT} Q$ does not hold since the implementation Q has input-output ✓-tock traces that are not input-output ✓-tock of P (for example, any that involves the input *in* followed by the output *out*). □

We note that, in the above example, all that P (that is, $\mathbf{Stop}_U$) can do is deadlock, but Q can exhibit any behaviour after $\langle in, out\rangle$. As a result, we argue that it is natural to expect that Q is regarded as being a faulty implementation of P, something that is not the case if we adopt tioco as the notion of correctness.

We now know that timed input-output refinement is strictly stronger than tioco for input-enabled implementations, which is the main result from this section. We next, in Theorem 5.13, consider the case where P and Q are both input-enabled. Formally, P is input enabled if, and only if, all refusals in all its traces are subsets of $O \cup \{\checkmark\}$.

We can strengthen Theorem 5.7 for input-enabled processes.

Theorem 5.11. *For an input-enabled process P, if $\rho \in iott^{O}_{M}[[tt[[P]]]]$, then $tt(st(\rho)) = \rho$.*

Proof.

$\rho \in iott^{O}_{M}[[tt[[P]]]]$

$\Rightarrow \forall\, 1\,..\,\#\rho;\ X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \mid \rho\, i = ref\ X \bullet X = O \cup \{\checkmark\}$

[P is input enabled and definitions of $iott^{O}_{M}[[P]]$, *addTick*, and *addOuts*]

$\Rightarrow \forall\, 1\,..\,\#\rho;\ X : \mathbb{P}\,\Sigma^{\checkmark}_{tock};\ e : \Sigma^{\checkmark}_{tock} \bullet (\rho\, i = ref\ X \Rightarrow X = O \cup \{\checkmark\} \wedge st(\rho)\, i = \delta) \wedge (\rho\, i = evt\ e \Rightarrow st(\rho)\, i = e)$

[definition of st]

$\Rightarrow \forall\, 1\,..\,\#\rho;\ X : \mathbb{P}\,\Sigma^{\checkmark}_{tock};\ e : \Sigma^{\checkmark}_{tock} \bullet$ [definition of tt]
$(\rho\, i = ref\ X \Rightarrow X = O \cup \{\checkmark\} \wedge tt(st(\rho))\, i = ref\ (O \cup \{\checkmark\})) \wedge (\rho\, i = evt\ e \Rightarrow tt(st(\rho))\, i = evt\ e)$

$\Rightarrow tt(st(\rho)) = \rho$ [property of sequences]

□

In the proof of Theorem 5.13, we use the lemma below regarding tioco; the corresponding result for ioco is known [55].

Lemma 5.12. *Given input-enabled processes P and Q, Q tioco $P \Leftrightarrow tstraces[[Q]] \subseteq tstraces[[P]]$.*

Proof. The right-to-left direction is immediate from the definition of tioco and so we focus on the left-to-right direction. We assume that P and Q are input-enabled and use proof by contradiction.

Q tioco $P \wedge \neg\, (tstraces[[Q]] \subseteq tstraces[[P]])$

$\Rightarrow Q$ tioco $P \wedge \exists\, \sigma_1 : Strace \bullet \sigma_1 \in tstraces[[Q]] \setminus tstraces[[P]]$ [property of sets]

$\Rightarrow Q$ tioco $P \wedge \exists\, \sigma_1, \sigma_2 : Strace;\ e : \Sigma^{\checkmark}_{tock} \cup \{\delta\} \bullet$ [by **TT0** and **TT1**, $\langle\rangle \in tstraces[[P]] \cap tstraces[[Q]]$]
$\sigma_1 = \sigma_2 \frown \langle e \rangle \wedge \sigma_1 \in tstraces[[Q]] \setminus tstraces[[P]] \wedge \sigma_2 \in tstraces[[P]]$

$\Rightarrow Q$ tioco $P \wedge \exists\, \sigma_1, \sigma_2 : Strace;\ e : \Sigma \cup \{tock, \delta\} \bullet$ [$e \neq \checkmark$ since Q is input-enabled]
$\sigma_1 = \sigma_2 \frown \langle e \rangle \wedge \sigma_1 \in tstraces[[Q]] \setminus tstraces[[P]] \wedge \sigma_2 \in tstraces[[P]]$

$\Rightarrow Q$ tioco $P \wedge \exists\, \sigma_2 : Strace;\ e : \Sigma \cup \{tock, \delta\} \bullet$ [predicate calculus]
$\sigma_2 \frown \langle e \rangle \in tstraces[[Q]] \setminus tstraces[[P]] \wedge \sigma_2 \in tstraces[[P]]$

$\Rightarrow Q$ tioco $P \wedge \exists\, \sigma_2 : Strace;\ e : \Sigma \cup \{tock, \delta\} \bullet$ [property of sets]
$\sigma_2 \frown \langle e \rangle \in tstraces[[Q]] \wedge \sigma_2 \frown \langle e \rangle \notin tstraces[[P]] \wedge \sigma_2 \in tstraces[[P]]$

$\Rightarrow \exists\, \sigma_2 : Strace;\ e : \Sigma \cup \{tock, \delta\} \bullet e \notin O \cup \{tock, \delta\} \wedge$ [definition of tioco]
$\sigma_2 \frown \langle e \rangle \in tstraces[[Q]] \wedge \sigma_2 \frown \langle e \rangle \notin tstraces[[P]] \wedge \sigma_2 \in tstraces[[P]]$

$\Rightarrow \exists\, \sigma_2 : Strace;\ e : \Sigma \setminus (O \cup \{tock, \delta\}) \bullet \sigma_2 \frown \langle e \rangle \notin tstraces[[P]] \wedge \sigma_2 \in tstraces[[P]]$ [predicate calculus]

This contradicts P being input-enabled as required. □

Finally, we prove that $\sqsubseteq_{IOTT}$ is identical to tioco if all processes are input-enabled.

Theorem 5.13. *Given input-enabled processes P and Q, $P \sqsubseteq_{IOTT} Q \Leftrightarrow Q$ tioco P.*

Proof. The left to right implication is given by Theorem 5.9. So, we assume that Q tioco P and prove that $P \sqsubseteq_{IOTT} Q$.

Q tioco P

$\Rightarrow tstraces[\![Q]\!] \subseteq tstraces[\![P]\!]$ [Lemma 5.12]

$\Rightarrow st(\!|\ iott^{O}[\![Q]\!]\ |\!) \subseteq st(\!|\ iott^{O}[\![P]\!]\ |\!)$ [definition of *tstraces*]

$\Rightarrow st(\!|\ iott^{O}_{M}[\![tt[\![Q]\!]]\!]\ |\!) \subseteq st(\!|\ iott^{O}_{M}[\![tt[\![P]\!]]\!]\ |\!)$ [Theorem 5.8]

$\Rightarrow tt(\!|\ st(\!|\ iott^{O}_{M}[\![tt[\![Q]\!]]\!]\ |\!)\ |\!) \subseteq tt(\!|\ st(\!|\ iott^{O}_{M}[\![tt[\![P]\!]]\!]\ |\!)\ |\!)$ [property of relational image]

$\Rightarrow iott^{O}_{M}[\![tt[\![Q]\!]]\!] \subseteq iott^{O}_{M}[\![tt[\![P]\!]]\!]$ [Theorem 5.11]

$\Rightarrow P \sqsubseteq_{IOTT} Q$ [Lemma 3.10]

□

To summarise, we have that input-output ✓-tock refinement is stronger than the Schmaltz and Tretmans version of tioco (and so also the Krichen and Tripakis version) and is equivalent to it if the specification is input-enabled. So, if an implementation fails a test according to tioco, we know that the implementation is not valid under timed input-output ✓-tock refinement. This is essential to unify the development and testing activities.

We now consider how we can test for input-output ✓-tock traces refinement.

6 TESTING AND INPUT-OUTPUT *tock*-CSP

We now present a testing theory for *tock*-CSP: a generative definition of test cases, definitions of test execution and test verdict, and a characterisation of exhaustive test sets. We provide formal definitions and proofs of soundness and exhaustiveness, which guarantee that the tests in the suite are sound and enough to establish conformance. The theory is generic in that it can also be used for checking standard ✓-tock refinement and traces refinement.

The formalisation, but not the application, of our theory is based on processes. Formally, a test execution involves two processes: a process Q representing a candidate implementation for the SUT, and a test case T, a process that attempts to elicit a specific erroneous behaviour when composed in parallel with Q. The process that represents Q can be described using any of the *tock*-CSP operators, so that nondeterminism can be specified explicitly (via $\sqcap$) or implicitly (arising for the combined semantics of the operators), but as said Q is assumed to be divergence free. Explicit nondeterminism $P \sqcap Q$ defines that the SUT can make a choice, independently from the tester, to behave as either P or Q. Implicit nondeterminism arises normally from parallelism, or even from an external choice such as $a \rightarrow P \,\Box\, a \rightarrow Q$, where the processes $a \rightarrow P$ and $a \rightarrow Q$ in choice offer the same event a to the tester, which then cannot identify a particular process in the choice via interaction with that event ($a \rightarrow P \,\Box\, a \rightarrow Q$ is not equal to $a \rightarrow (P \,\Box\, Q)$).

A process representing the composition of an SUT and a test process and is called a test execution. A notion of an implementation failing a test formally captures how the fail verdict for erroneous behaviour is identified through a test execution. For a given conformance relation, we specify a set of tests that is sound (that is, no correct SUT can fail) and exhaustive (that is, every fault can be identified by some test).

Here, we define the above mentioned notions of test case, test execution, and failure for our input-output ✓-tock model, but observe that the definitions also apply for the standard ✓-tock semantics (Section 6.1). Accordingly, in Section 6.2 we provide test suites for both the input-output ✓-tock semantics and the standard ✓-tock semantics. Finally, in the latter part of this section, we also consider semantics induced by traces of events only (Section 6.3).

6.1 Testing theory: setup

We first describe how test execution and the notion of an SUT failing a test are defined. For that, we rely on the fact that a test case uses special verdict events $V = \{pass, fail, inc\}$. The verdict of the events *pass* and *fail* is obvious, and *inc* indicates an inconclusive verdict: the test execution did not manage to drive the SUT to the end of the trace considered in the test case. We recall that behaviour characterised by a prefix of a valid trace is valid according to the notion of correctness captured by (input-output ✓-tock) refinement, so it is not appropriate to give a *fail* verdict. Moreover, in defining test execution, we use one more special event *ticktest*, specific to our theory for *tock*-CSP, to handle termination. This event is issued once the SUT finishes, and can be observed by the test case.

Both verdict events and *ticktest* are assumed to be fresh, that is, not occurring in any process under consideration apart from the test case and test execution processes. For simplicity, we assume that V and Σ are disjoint, but that $ticktest \in \Sigma$. In a test execution, all events, except the verdict events in V and *tock*, are hidden. The last verdict event of the trace of the execution process provides the verdict of the experiment.

Formally, we define the execution of a test case T against an SUT Q as follows.

Definition 6.1.

$$Execution(Q, T) \mathrel{\widehat{=}} ((Q;\ ticktest \rightarrow_U \mathbf{Stop}_U) \,[\![\, \Sigma \,]\!]\, T) \setminus \Sigma$$

We sequentially compose the SUT process Q with a prefixing for the event *ticktest* followed by a deadlock, so that the test process T can detect termination of Q through the occurrence of *ticktest*. We use here an untimed version $\rightarrow_U$ of the prefixing operator, which does not allow time to pass. So, if Q terminates, the signalling of *ticktest* is urgent. This prefixing operator can be defined using other *tock*-CSP operators as follows: $e \rightarrow_U P = e \rightarrow P \,\Box\, \mathbf{Stop}_U$.

The parallel composition synchronises on all events in Σ, including *ticktest*. These events are also hidden, so that, as said, only the passage of time and the verdict events are visible. $Execution(Q, T)$ is defined in the standard ✓-tock model, since T needs to observe all events in Σ, including the outputs. So, the synchronisation set in the parallelism needs to include all outputs. We recall, however, that well-formed parallelisms in the input-output ✓-tock model cannot have outputs in their synchronisation sets. We, therefore, adopt ✓-tock in the theory here.

The verdict of a test execution is given by the final verdict event appearing before the execution of the test deadlocks. A formal definition for the verdict, which characterises the failure verdict, is provided below.

Definition 6.2.

$$Q\ fails\ T \mathrel{\widehat{=}} \exists \rho : TTTrace \bullet \rho \frown \langle evt\ fail, ref\ (\Sigma^{\checkmark}_{tock} \cup V)\rangle \in tt[\![Execution(Q, T)]\!]$$

Since $Execution(Q, T)$ is defined using the ✓-tock model, $Q\ fails\ T$ is intrinsically characterised in that model. As usual, deadlock is characterised by a refusal of all events, including *tock* and the verdict events.

Example 6.3. Here, we consider an example using a test that can be generated from the *RD* process in Example 3.2. A forbidden trace (disallowed behaviour) of *RD* is $\langle takeoff, \checkmark\rangle$, since termination can happen after *turnoff*, but not *takeoff*.

In our theory, the test for this trace (as formally defined in the sequel) is specified as a process as follows.

$$NT = inc \rightarrow_U takeoff \rightarrow_U pass \rightarrow_U ticktest \rightarrow_U fail \rightarrow_U \mathbf{Stop}_U$$

Intuitively, for any SUT Q, in $Execution(Q, NT)$, the test NT raises the *inc* verdict, and then attempts to drive Q to engage in *takeoff*. If that succeeds, the verdict becomes *pass*, as a valid trace is observed. Proceeding, there is, however, the possibility of observing *ticktest*, indicating that Q terminated. In this case, the final verdict is *fail* as NT then deadlocks.

□

We define test cases via an operator $T_{tt}(\rho)$ that maps a ✓-tock trace ρ to the corresponding test-(case) process. Here, ρ ranges over minimal traces forbidden by the specification, that is, every proper prefix of ρ is a valid trace – hence only the final observation is forbidden. A test case $T_{tt}(\rho)$ drives the SUT through ρ, until it reaches the final observation; it can detect if an SUT admits ρ when they are composed into a test execution as described above.

T_{tt} is defined inductively for all traces in *TTTrace*. There are five cases, covering the possible forms of a *TTTrace*.

Case (1) is for the empty trace $\langle\rangle$. It is included for technical convenience, as a base case for the inductive definition.

$$(1) \quad T_{tt}(\langle\rangle) = fail \rightarrow_U \mathbf{Stop}_U$$

Since every process has the empty trace in its semantics, we never test for $\langle\rangle$ on its own, but only as a suffix of a nontrivial forbidden trace. As shown above, the test $T_{tt}(\langle\rangle)$ always yields verdict *fail*. This captures the fact that the verdict of a test execution that manages to drive the SUT to the end of the trace that defines its test is *fail*.

The other verdict events are used by the tests corresponding to longer traces. As formalised below, in general terms, as the test succeeds in driving the SUT along the trace, the (potentially intermediate) verdicts are *inc*.

Case (2) is that of a refusal X at the end of the trace: so a forbidden refusal. In that case the test first outputs a verdict *fail*, which can be potentially overridden by a *pass*, signalling that X is not refused. After *fail*, all events in X, except ✓, alongside the special event *ticktest* are offered, and, if accepted, *pass* is raised. In the context of our test execution, the event *ticktest* always signals that the SUT has just terminated – hence no refusal could have been observed.

$$(2) \quad T_{tt}(\langle ref\ X\rangle) = fail \rightarrow_U \left(\begin{array}{l} \Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow_U pass \rightarrow_U \mathbf{Stop}_U \\ \Box \\ ticktest \rightarrow_U pass \rightarrow_U \mathbf{Stop}_U \end{array} \right)$$

Case (3) covers a trace ending with a forbidden termination ✓. First, the verdict *pass* is output, which can potentially be overridden by a *fail*, if the SUT does terminate. We recall that, according to the definition of $Execution(Q, T)$, the SUT terminates if, and only if, the first component of the parallel composition performs the *ticktest* event. Hence if a synchronisation on *ticktest* is offered, the test case proceeds to the base process $T_{tt}(\langle\rangle)$, which outputs the *fail* verdict; otherwise the test case deadlocks and the last verdict *pass* becomes the verdict of the experiment.

$$(3) \quad T_{tt}(\langle evt\ \checkmark\rangle) = pass \rightarrow_U ticktest \rightarrow_U T_{tt}(\langle\rangle)$$

Case (4) covers traces $\langle evt\ e\rangle \frown \rho$ starting with an event e different from *tock*. A non-failing verdict is output first: depending on whether the entire correct trace prefix has been executed, that is, on whether ρ is empty or not, the verdict is *pass* or *inc*. The initial event e is then offered. This event may not be accepted by the SUT, in which case the process deadlocks and the non-failing verdict just output becomes the verdict of the experiment. Otherwise e is performed, and

we proceed to the test corresponding to the remainder of the trace where a different verdict may be given.

$$(4) \quad T_{tt}(\langle evt\ e\rangle \frown \rho) = \begin{cases} pass \rightarrow_U e \rightarrow_U T_{tt}(\rho) \text{ if } \rho = \langle\rangle, e \neq tock \\ inc \rightarrow_U e \rightarrow_U T_{tt}(\rho) \text{ if } \rho \neq \langle\rangle, e \neq tock \end{cases}$$

In the next example, we illustrate the application of cases (1), (3), and (4).

Example 6.4. Here, we consider the test *NT* presented in Example 6.3. It is the process $T_{tt}(\langle takeoff, \checkmark\rangle)$, defined using the forbidden trace $\langle takeoff, \checkmark\rangle$ of *RD* from Example 3.2. Applying the definitions above for $T_{tt}(\rho)$, we get the process that raises the *inc* verdict and then *takeoff* (case (4)). If that succeeds, the verdict becomes *pass* (case (3)). Proceeding, there is the possibility of observing *ticktest*, when the final verdict is *fail* and the test deadlocks (case (2)). □

The final case (5) handles traces $\langle X, tock\rangle \frown \rho$ starting with a refusal X immediately followed by *tock*. The refusal X is a correct observation (forbidden refusals are handled by case (2)). The test first outputs a non-failing verdict *pass* or *inc*, depending on whether ρ is empty or not. The SUT is then tested for the presence of the refusal X like in case (2). If any event in X or termination is observed, then the test deadlocks and the non-failing verdict stands. In this case, the SUT has not been driven to the end of the trace for the test. A timeout, after 1 time unit, however, allows for time to pass. If, however, an SUT timelocks (because of a deadline), then the test execution deadlocks, again keeping the non-failing verdict. On the other hand, if *tock* is observed, then we proceed with the test corresponding to the trace suffix ρ.

$$(5) \quad T_{tt}(\langle ref\ X, evt\ tock\rangle \frown \rho) = \begin{cases} pass \rightarrow_U \left(\begin{array}{l} \Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow \mathbf{Stop}_U \\ \Box \\ ticktest \rightarrow \mathbf{Stop}_U \end{array} \right) \triangle_1\ T_{tt}(\rho) \quad \text{if } \rho = \langle\rangle \\ inc \rightarrow_U \left(\begin{array}{l} \Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow \mathbf{Stop}_U \\ \Box \\ ticktest \rightarrow \mathbf{Stop}_U \end{array} \right) \triangle_1\ T_{tt}(\rho) \quad \text{if } \rho \neq \langle\rangle \end{cases}$$

As for case (4), this covers two forms of trace: one where ρ is empty and one where it is not. In both cases, the refusal of the set X of events is observed by offering all events that are in $X \setminus \{\checkmark\}$ and *ticktest*. If an SUT Q can perform an event $e \in X \setminus \{\checkmark\} \cup \{ticktest\}$ then the composition of Q and the test case in $Execution(Q, T_{tt}(\langle ref\ X, evt\ tock\rangle \frown \rho))$ can perform e and then deadlock with a final verdict of either *pass* or *inc*. Further, because all events in Σ, which includes *ticktest*, are hidden in $Execution(Q, T_{tt}(\langle ref\ X, evt\ tock\rangle \frown \rho))$, e becomes an internal event and so is urgent. As a result, time cannot pass until Q and the test case perform such an event e, and therefore there is no possibility of a timeout. $Execution(Q, T_{tt}(\langle ref\ X, evt\ tock\rangle \frown \rho))$ then deadlocks. This ensures that, if Q can engage in an event $e \in X \setminus \{\checkmark\} \cup \{ticktest\}$, the test execution deadlocks and so the test cannot proceed with any events from $T_{tt}(\rho)$. As a result, the verdict *fail* cannot occur and so Q passes this test case as expected.

Example 6.5. Going back to *RD* in Example 3.2 again, another forbidden trace is $\langle takeoff, \Sigma^{\checkmark}, tock, found\rangle$ since after *takeoff* and one time unit, an *RD* implementation must *move* before *found*. The test we get for this trace is as follows.

$$
\begin{aligned}
inc \rightarrow_U takeoff \rightarrow_U inc \rightarrow_U (\ & takeoff \rightarrow \mathbf{Stop}_U \\
& \Box \\
& move \rightarrow \mathbf{Stop}_U \\
& \Box \\
& found \rightarrow \mathbf{Stop}_U \\
& \Box \\
& land \rightarrow \mathbf{Stop}_U \\
& \Box \\
& turnoff \rightarrow \mathbf{Stop}_U \\
& \Box \\
& ticktest \rightarrow \mathbf{Stop}_U\ )\ \triangle_1\ pass \rightarrow_U found \rightarrow_U fail \rightarrow_U \mathbf{Stop}_U
\end{aligned}
$$

After a second *inc* event, reflecting the fact that the trace continues after the refusal $\Sigma^{\checkmark}$, all events from Σ and *ticktest* (in lieu of $\checkmark$) are offered in choice. If the SUT engages in any of these events, the test execution deadlocks. Time cannot pass, as *tock* is refused as well, and the test is never interrupted. In this case, the *inc* event gives the verdict. If, however, a *tock* happens, the choice is interrupted, and the verdict is now *pass*, but the *SUT* is offered *found*. If the SUT engages in *found*, the final verdict is *fail*, and that cannot change. □

The following theorem formally establishes that test processes obtained with the function $T_{tt}(\rho)$ work exactly as intended: for a given trace ρ and implementation Q, the test execution involving $T_{tt}(\rho)$ and Q interacting with each other yields verdict *fail* precisely when Q exhibits the trace ρ. It is important to note that, while we use this result in the next section to prove that a given test set is sound and complete, the following result is more general and shows that T_{tt} can be used as the basis of testing whenever we have a test generation technique that produces a set of disallowed traces. We return to this point in the final section, when we describe future work.

Theorem 6.6. *For any $\rho \in TTTrace$ and implementation Q*

$$Q\ fails\ T_{tt}(\rho) \ \Leftrightarrow\ \rho \in tt[\![Q]\!]$$

Proof. We define $\widehat{Q} \mathrel{\widehat{=}} Q;\ ticktest \rightarrow_U STOP_U$, the process on the left-hand side of the parallelism in $Execution(Q, T)$. Due to the semantics of hiding, a trace $\rho_h \in tt[\![Execution(Q, T)]\!] = tt[\![(\widehat{Q} \,[\![\, \Sigma \,]\!]\, T) \setminus \Sigma]\!]$ has a corresponding trace ρ_v where no events are hidden, that is, such that $\rho_v \in tt[\![\widehat{Q} \,[\![\, \Sigma \,]\!]\, T]\!] \ \wedge\ \rho_h \in hideTrace\, \Sigma\, \rho_v$. Importantly, we can identify such a trace ρ_v whose refusals all subsume Σ, that is, so that ρ_v is Σ-saturated. We say that a trace ρ is Y-saturated, written $satref(\rho, Y)$, if every refusal in ρ subsumes Y, that is, $satref(\rho, Y) \Leftrightarrow \forall\, i : \mathrm{dom}\ \rho \mid \rho\, i \in \mathrm{ran}\ ref \bullet Y \subseteq ref^{\sim}(\rho\, i)$. We call a trace ρ_v satisfying these properties a *visible counterpart* of ρ_h.

Furthermore, since all events of $\widehat{Q}$ are in Σ, given a trace $\rho \in tt[\![\widehat{Q} \,[\![\, \Sigma \,]\!]\, T]\!]$, there is a corresponding trace of T with the same event sequence. Formally, this is a trace ρ_T such that $\rho_T \in tt[\![T]\!]$ and $\rho \sim_{evt} \rho_T$. This captures the fact that the projections of ρ and ρ_T to ran *evt* are equal. Formally, $\rho \sim_{evt} \rho' \ \Leftrightarrow\ \#\rho = \#\rho' \wedge \forall\, i : \mathrm{dom}\ \rho \mid \rho\, i \in \mathrm{ran}\ evt \bullet \rho\, i = \rho'\, i$. We call a trace ρ_T satisfying such properties a *test-component counterpart* of ρ_h.

We now proceed with the proof by structural induction on ρ. For simplicity, we omit the constructor functions *evt*

and *ref* when the context makes it clear whether we refer to an event or a refusal.

Case $\langle\rangle$ ($\Rightarrow$). From **TT0**, $\langle\rangle \in tt[\![Q]\!]$ always holds.

Case $\langle\rangle$ ($\Leftarrow$). We show that $\langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![(\widehat{Q} \,[\![\, \Sigma \,]\!]\, fail \rightarrow_U \mathbf{Stop}_U) \setminus \Sigma]\!]$. It follows directly from the semantics of hiding and the following two results proved below. First, we have the following.

$\langle \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![\widehat{Q} \,[\![\, \Sigma \,]\!]\, \mathbf{Stop}_U]\!]$ [property of parallelism and $\mathbf{Stop}_U$]

$\Rightarrow \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![\widehat{Q} \,[\![\, \Sigma \,]\!]\, fail \rightarrow_U \mathbf{Stop}_U]\!]$ [property of parallelism and prefixing]

In addition, we can make the following observation.

$\langle \Sigma^{\checkmark}_{tock} \cup V\rangle \in hideTrace\, \Sigma\, \langle \Sigma^{\checkmark}_{tock} \cup V\rangle$ [definition of *hideTrace* and $\Sigma^{\checkmark}_{tock} \cup V \subseteq \Sigma^{\checkmark}_{tock} \cup V$]

$\Rightarrow \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in hideTrace\, \Sigma\, \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle$ [definition of *hideTrace* and $fail \notin \Sigma$]

Case $\langle\checkmark\rangle$ *(*$\Rightarrow$*).* Above, we do not rely on properties of Q, so for any P, we have $\langle fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \in tt[\![P \,[\![\, \Sigma \,]\!]\, T_{tt}(\langle\rangle)]\!]$. For a $\rho_h : TTTrace$ so that $\rho_h \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![(\widehat{Q} \,[\![\, \Sigma \,]\!]\, T_{tt}(\langle\checkmark\rangle)) \setminus \Sigma]\!]$, we consider its visible counterpart ρ_v.

$\rho_v \frown \langle fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \in tt[\![(\widehat{Q} \,[\![\, \Sigma \,]\!]\, pass \rightarrow_U ticktest \rightarrow_U T_{tt}(\langle\rangle))]\!]$

$\Leftrightarrow\ \rho_v \frown \langle fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \in tt[\![(\widehat{Q} \,[\![\, \Sigma \,]\!]\, pass \rightarrow_U ticktest \rightarrow_U fail \rightarrow_U \mathbf{Stop}_U]\!]$

From the semantics of $\rightarrow_U$ we deduce that ρ_v must be $\langle pass, ticktest\rangle$. We then can proceed as follows. Here and in what follows, we use the notation $\alpha(P)$ to refer to the set of events used by the process P.

$\langle pass, ticktest, fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \in tt[\![\widehat{Q} \,[\![\, \Sigma \,]\!]\, pass \rightarrow_U ticktest \rightarrow_U T_{tt}(\langle\rangle)]\!]$

$\Leftrightarrow\ \langle ticktest, fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \in tt[\![\widehat{Q} \,[\![\, \Sigma \,]\!]\, ticktest \rightarrow_U T_{tt}(\langle\rangle)]\!]$ [$pass \notin \alpha(\widehat{Q})$]

$\Leftrightarrow\ \langle fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \in tt[\![\widehat{Q} \text{ after } ticktest \,[\![\, \Sigma \,]\!]\, T_{tt}(\langle\rangle)]\!]$ [$ticktest \in \alpha(\widehat{Q})$]

$\Leftrightarrow\ tt[\![\widehat{Q} \text{ after } ticktest]\!] \neq \emptyset$

[($\Rightarrow$) by a property of parallelism: $\widehat{Q}$ after *ticktest* must have a singleton refusal trace at least.]

[($\Leftarrow$) follows from $\langle fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \in tt[\![P \,[\![\, \Sigma \,]\!]\, T_{tt}(\langle\rangle)]\!]$ for any P.]

$\Leftrightarrow\ \langle ticktest\rangle \in tt[\![\widehat{Q}]\!]$ [definition of after]

$\Leftrightarrow\ \langle ticktest\rangle \in tt[\![Q;\ ticktest \rightarrow_U \mathbf{Stop}_U]\!]$ [definition of $\widehat{Q}$]

$\Leftrightarrow\ \langle\checkmark\rangle \in tt[\![Q]\!]$ [semantics of sequential composition and $ticktest \notin \alpha(Q)$]

Case $\langle\checkmark\rangle$ *(*$\Leftarrow$*).* From $\langle\checkmark\rangle \in tt[\![Q]\!]$, we get $\langle pass, ticktest, fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \in tt[\![\widehat{Q} \,[\![\Sigma]\!]\, T_{tt}(\langle\checkmark\rangle)]\!]$, as shown above for the case ($\Rightarrow$). Since $\langle pass, fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \in hideTrace\, \Sigma\, \langle pass, ticktest, fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle$, by the definition of *hideTrace*, we obtain $\langle pass, fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \in tt[\![(\widehat{Q} \,[\![\, \Sigma \,]\!]\, T_{tt}(\langle\checkmark\rangle)) \setminus \Sigma]\!]$.

Case $\langle X\rangle$ *(*$\Rightarrow$*).* For a $\rho_h \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![(\widehat{Q} \,[\![\, \Sigma \,]\!]\, T_{tt}(\langle X\rangle)) \setminus \Sigma]\!]$, we consider its visible counterpart ρ_v, and

its test-component counterpart $\rho_T \frown \langle fail, X_T\rangle \in tt[\![T_{tt}(\langle X\rangle)]\!]$. By definition of $T_{tt}(\langle X\rangle)$, we have the following.

$$\rho_T \frown \langle fail, X_T\rangle \in tt[\![fail \rightarrow_U ((\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow_U pass \rightarrow_U \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow_U pass \rightarrow_U \mathbf{Stop}_U)]\!]$$

From the semantics of $\rightarrow_U$ and $\Box$, and since *fail* is not in X, we can deduce that $\rho_T = \langle\rangle$. This, combined with

$$\rho_v \frown \langle fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \sim_{evt} \rho_T \frown \langle fail, X_T\rangle = \langle fail, X_T\rangle$$

yields $\rho_v = \langle\rangle$ and so $\langle fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle$ belongs to

$$tt[\![\widehat{Q} \,[\![\, \Sigma \,]\!]\, (fail \rightarrow_U ((\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow_U pass \rightarrow_U \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow_U pass \rightarrow_U \mathbf{Stop}_U))]\!]$$

By the semantics of parallelism, since *fail* does not belong to the synchronisation set Σ, we obtain the following.

$$\langle(\Sigma^{\checkmark}_{tock} \cup V)\rangle \in tt[\![\widehat{Q} \,[\![\, \Sigma \,]\!]\, ((\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow_U pass \rightarrow_U \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow_U pass \rightarrow_U \mathbf{Stop}_U)]\!]$$

Hence there must be refusals $\langle X^L\rangle \in tt[\![\widehat{Q}]\!]$ and

$$\langle X^R\rangle \in tt[\![((\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow_U pass \rightarrow_U \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow_U pass \rightarrow_U \mathbf{Stop}_U)]\!]$$

such that $\Sigma^{\checkmark}_{tock} \cup V = X^L \cup X^R$. We observe that $X \setminus \{\checkmark\} \cap X^R = \emptyset$ because the right-hand parallel process initially offers an external choice that includes all events in $X \setminus \{\checkmark\}$. It therefore must be the case that $X \setminus \{\checkmark\} \subseteq X^L$. Moreover, because of **TT3**, we can choose an X^L that contains $\checkmark$. Therefore, we can deduce that $X \subseteq X^L$. Since refusals of a process are downward-closed due to **TT1**, we thus finally obtain $\langle X\rangle \in tt[\![\widehat{Q}]\!]$.

Case $\langle X\rangle$ *(*$\Leftarrow$*).* From $\langle X\rangle \in tt[\![Q]\!]$, since no trace of Q contains *ticktest* or the events in V, using **TT2** we get $\langle X \cup V \cup \{ticktest\}\rangle \in tt[\![Q]\!]$. In addition, $\langle fail, (\Sigma^{\checkmark}_{tock} \setminus (X \cup \{ticktest\})) \cup V\rangle$ is in the following set.

$$tt[\![T_{tt}(\langle X\rangle)]\!] = tt[\![fail \rightarrow_U ((\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow_U pass \rightarrow_U \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow_U pass \rightarrow_U \mathbf{Stop}_U)]\!]$$

This follows from the semantics of $\rightarrow_U$ and $\Box$ and **TT2**. We now proceed to show that

$$\langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![(\widehat{Q} \,[\![\, \Sigma \,]\!]\, T_{tt}(\langle X\rangle)) \setminus \Sigma]\!]$$

This is a consequence of the following two results, and the semantics of parallelism and hiding.

(1) $\langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in \langle X \cup V \cup \{ticktest\}\rangle \,[\![\, \Sigma \,]\!]^T\, \langle fail, (\Sigma^{\checkmark}_{tock} \setminus (X \cup \{ticktest\})) \cup V\rangle$. Since $X \cup \{ticktest\} \subseteq \Sigma$, then $(X \cup V \cup \{ticktest\}) \setminus \Sigma^{\checkmark}_{tock} = V = ((\Sigma^{\checkmark}_{tock} \setminus (X \cup \{ticktest\})) \cup V) \setminus \Sigma^{\checkmark}_{tock}$. Moreover, we observe that $((X \cup V \cup \{ticktest\}) \cup (\Sigma^{\checkmark}_{tock} \setminus (X \cup \{ticktest\})) \cup V) = \Sigma^{\checkmark}_{tock} \cup V$, hence by the definition of $\_ \,[\![\,]\!]^T \_$, we obtain $\langle \Sigma^{\checkmark}_{tock} \cup V\rangle \in \langle X \cup V \cup \{ticktest\}\rangle \,[\![\, \Sigma \,]\!]^T\, \langle(\Sigma^{\checkmark}_{tock} \setminus (X \cup \{ticktest\})) \cup V\rangle$. So, the definition of $\_ \,[\![\,]\!]^T \_$ give us (1).

(2) $\langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in hideTrace\, \Sigma\, \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle$. We note that $fail \notin \Sigma$ and $\Sigma \subseteq \Sigma^{\checkmark}_{tock} \cup V$, so the result follows from the definition of $hideTrace\, \Sigma\, \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle$.

We note that $\Sigma^{\checkmark}_{tock}$ is the synchronisation set of the parallelism, with $\checkmark$ and *tock* added.

Case $\langle X, tock\rangle \frown \rho$ *(*$\Rightarrow$*).* We need to prove $\langle X, tock\rangle \frown \rho \in tt[\![Q]\!]$, using the induction hypothesis, namely, for all Q, Q *fails* $T_{tt}(\rho) \;\Leftrightarrow\; \rho \in tt[\![Q]\!]$. For a $\rho_h : TTTrace$ so that $\rho_h \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![(\widehat{Q} \,[\![\, \Sigma \,]\!]\, T_{tt}(\langle X, tock\rangle \frown \rho) \setminus \Sigma]\!]$, we consider its visible counterpart ρ_v and a test-component counterpart ρ_T so that $\rho_T \frown \langle fail, X_T\rangle \in tt[\![T_{tt}(\langle X, tock\rangle \frown \rho)]\!]$. The definition of $T_{tt}(\langle X, tock\rangle \frown \rho)$ gives us the following.

$$\rho_T \frown \langle fail, X_T\rangle \in tt[\![inc \rightarrow_U ((\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow \mathbf{Stop}_U) \,\triangle_1\, (T_{tt}(\rho))]\!]$$

From the semantics of $\rightarrow_U$ and $\triangle_1$, there are traces ρ_1 and ρ_2 and refusal X_2 such that

$$\rho_T \frown \langle fail, X_T\rangle = \langle inc\rangle \frown \rho_1 \frown \langle X_2, tock\rangle \frown \rho_2 \frown \langle fail, X_T\rangle$$

where $\rho_2 \frown \langle fail, X_T\rangle \in tt[\![T_{tt}(\rho)]\!]$, $\rho_1 \frown \langle X_2, tock\rangle \in tt[\![(\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow \mathbf{Stop}_U)]\!]$, and $\#(\rho_1 \upharpoonright \{tock\}) = 0$. For any set E, we have $\rho_1 \frown \langle X, tock\rangle \in tt[\![\Box\, e : E \bullet e \rightarrow \mathbf{Stop}_U]\!] \;\Rightarrow\; \rho_1 \in tocks(\Sigma \setminus E)$ since the choice admits no further *tock* events after an event e in E. This and $\#(\rho_1 \upharpoonright \{tock\}) = 0$ mean that $\rho_1 = \langle\rangle$. Hence $\rho_T \frown \langle fail, X_T\rangle = \langle inc, X_2, tock\rangle \frown \rho_2 \frown \langle fail, X_T\rangle$. Since $\rho_v \frown \langle fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \sim_{evt} \rho_T \frown \langle fail, X_T\rangle$, then ρ_v is of the form $\langle inc, X_3, tock\rangle \frown \rho_3$, for some X_3 and ρ_3. So, we have that $\langle inc, X_3, tock\rangle \frown \rho_3 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle$ is in

$$tt[\![\widehat{Q} \,[\![\, \Sigma \,]\!]\, (inc \rightarrow_U ((\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow \mathbf{Stop}_U) \,\triangle_1\, T_{tt}(\rho))]\!]$$

From $inc \notin \alpha(\widehat{Q})$, the set of events used in $\widehat{Q}$, we obtain $\langle X_3, tock\rangle \frown \rho_3 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle$ is in

$$tt[\![\widehat{Q} \,[\![\, \Sigma \,]\!]\, (((\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow \mathbf{Stop}_U) \,\triangle_1\, T_{tt}(\rho))]\!]$$

By the semantics of parallelism, this means that there must be traces

$$\langle X_3^L, tock\rangle \frown \rho^L \in tt[\![\widehat{Q}]\!]$$
$$\langle X_3^R, tock\rangle \frown \rho^R \in [\![((\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow \mathbf{Stop}_U) \,\triangle_1\, T_{tt}(\rho)]\!]$$

such that (1) $\langle X_3\rangle \in \langle X_3^L\rangle \,[\![\, \Sigma \,]\!]^T\, \langle X_3^R\rangle$, and (2) $\rho_3 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in \rho^L \,[\![\, \Sigma \,]\!]^T\, \rho^R$. From (1), the definition of trace parallelism ($\_ \,[\![\, \Sigma \,]\!]^T\, \_$) gives $X_3 = X_3^L \cup X_3^R$. We observe that $\Sigma \subseteq X_3$ (as X_3 occurs in a Σ-saturated trace); we therefore have $\Sigma \subseteq X_3^L \cup X_3^R$. Since $(X \setminus \{\checkmark\}) \cap X_3^R = \emptyset$ because the right-hand parallel process initially offers an external choice that includes all events in $X \setminus \{\checkmark\}$. S, it must be the case that $X \setminus \{\checkmark\} \subseteq X_3^L$. Moreover, because of **TT3**, we can choose an X_3^L that contains $\checkmark$. Therefore, we can deduce that $X \subseteq X_3^L$. From (2) it follows that $\rho_3 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle$ is in

$$tt[\![\widehat{Q} \,\mathsf{after}\, \langle X_3^L, tock\rangle \,[\![\, \Sigma \,]\!]\, (((\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow \mathbf{Stop}_U) \,\triangle_1\, T_{tt}(\rho)) \,\mathsf{after}\, \langle X_3^R, tock\rangle]\!]$$

For a process P and a trace ρ, we have $tt[\![P \,\mathsf{after}\, \rho_1]\!] \;\widehat{=}\; \{\rho_2 : TTTrace \mid \rho_1 \frown \rho_2 \in tt[\![P]\!] \bullet \rho_2\}$. In addition,

$$(((\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow \mathbf{Stop}_U) \,\triangle_1\, T_{tt}(\rho)) \,\mathsf{after}\, \langle X_3^R, tock\rangle = T_{tt}(\rho)$$

because the interruption happens after exactly one *tock*, when then $T_{tt}(\rho)$ takes over. Hence

$$\rho_3 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![\widehat{Q} \,\mathsf{after}\, \langle X_3^L, tock\rangle \,[\![\, \Sigma \,]\!]\, T_{tt}(\rho)]\!]$$

From this and the induction hypothesis, we obtain $\rho \in tt[\![Q \,\mathsf{after}\, \langle X_3^L, tock\rangle]\!]$. Hence from the definition of after we finally obtain $\langle X_3^L, tock\rangle \frown \rho \in tt[\![Q]\!]$. Since as noted above $X \subseteq X_3^L$, by **TT1** we get $\langle X, tock\rangle \frown \rho \in tt[\![Q]\!]$.

Case $\langle X, tock\rangle \frown \rho$ ($\Leftarrow$). From $\langle X, tock\rangle \frown \rho \in tt[\![Q]\!]$, we get $\rho \in tt[\![Q \,\mathsf{after}\, \langle X, tock\rangle]\!]$. By the induction hypothesis, there is a trace such that $\rho_h \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![(\widehat{Q} \,\mathsf{after}\, \langle X, tock\rangle \,[\![\, \Sigma \,]\!]\, T_{tt}(\rho)) \setminus \Sigma]\!]$. For its visible counterpart $\rho_v \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![\widehat{Q} \,\mathsf{after}\, \langle X, tock\rangle \,[\![\Sigma]\!]\, T_{tt}(\rho)]\!]$ we have $\rho_h \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in hideTrace\, \Sigma\, (\rho_v \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle)$. We define $T_{ver} = T_{tt}(\langle X, tock\rangle \frown \rho) \,\mathsf{after}\, \langle inc\rangle$ if $\rho \neq \langle\rangle$ or $T_{ver} = T_{tt}(\langle X, tock\rangle \frown \rho) \,\mathsf{after}\, \langle pass\rangle$, if $\rho = \langle\rangle$. In both cases, from the definition of $T_{tt}(\rho)$, we have $T_{ver} = ((\Box\, e : X \setminus \{\checkmark\} \bullet e \rightarrow \mathbf{Stop}_U) \,\Box\, ticktest \rightarrow \mathbf{Stop}_U) \,\triangle_1\, T_{tt}(\rho)$. For every ρ_1 in $tt[\![(\widehat{Q} \,\mathsf{after}\, \langle X, tock\rangle) \,[\![\, \Sigma \,]\!]\, T_{tt}(\rho)]\!]$, we have $\langle \Sigma, tock\rangle \frown \rho_1 \in tt[\![\widehat{Q} \,[\![\, \Sigma \,]\!]\, T_{ver}]\!]$. This is because X is refused before *tock* in $\widehat{Q}$ and $\Sigma \setminus (\{tock, ticktest\} \cup X)$ are refused in T_{ver}. So, $\Sigma \setminus \{ticktest\}$ is a refusal of the parallelism. In addition, *ticktest* is not an event of $\widehat{Q}$, so it is in particular also refused before a *tock* in $\widehat{Q}$. So, by **TT2**, we have the refusal Σ.

With this result, by taking ρ_1 to be $\rho_v \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle$, we obtain $\langle \Sigma, tock\rangle \frown \rho_v \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![\, \widehat{Q} \,[\![\Sigma]\!]\, T_{ver}]\!]$. Using the definitions of T_{ver} and parallelism, we obtain $\langle ver, \Sigma, tock\rangle \frown \rho_v \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![\, \widehat{Q} [\![\Sigma]\!]\, T_{tt}(\langle X, tock\rangle \frown \rho)]\!]$, where *ver* stands for either *inc* or *pass*. From the definition of *hideTrace*, we have $hideTrace\ \Sigma\ (\langle ver, \Sigma, tock\rangle) \neq \emptyset$. Moreover, as previously noted, ρ_v is Σ-saturated, so $hideTrace\ \Sigma\ (\rho_v \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle) \neq \emptyset$. Together, these give

$$\exists\, \rho_2 : TTTrace \bullet \rho_2 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![(\widehat{Q}\ [\![\ \Sigma\ ]\!]\ T_{tt}(\langle X, tock\rangle \frown \rho)) \setminus \Sigma]\!]$$

So, $Q\ fails\ T_{tt}(\langle X, tock\rangle \frown \rho)$, by the definition of *fails*.

Case $\langle e\rangle \frown \rho$ ($\Rightarrow$). We need to prove $\langle e\rangle \frown \rho \in tt[\![Q]\!]$, using the induction hypothesis $Q\ fails\ T_{tt}(\rho) \Leftrightarrow \rho \in tt[\![Q]\!]$ for every Q. For a $\rho_h \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![(\widehat{Q}\ [\![\ \Sigma\ ]\!]\ T_{tt}(\langle e\rangle \frown \rho) \setminus \Sigma]\!]$, we consider its visible counterpart $\rho_v \frown \langle fail, (\Sigma^{\checkmark}_{tock} \cup V)\rangle \in tt[\![(\widehat{Q}[\![\Sigma]\!]\, T_{tt}(\langle X, tock\rangle \frown \rho)]\!]$ and test-component counterpart $\rho_T \frown \langle fail, X_T\rangle \in tt[\![\, T_{tt}(\langle e\rangle \frown \rho)]\!]$. From the definition of T_{tt}, we have $\rho_T \frown \langle fail, X_T\rangle \in tt[\![\, inc \rightarrow_U e \rightarrow_U T_{tt}(\rho)]\!]$. Since the trace contains the event *fail*, which is used in $T_{tt}(\rho)$ only, the trace must have a suffix from $tt[\![\, T_{tt}(\rho)]\!]$. From the semantics of $\rightarrow_U$, we can deduce that there is a ρ_1 such that $\rho_T = \langle inc, e\rangle \frown \rho_1$ and $\rho_1 \frown \langle fail, X_T\rangle \in tt[\![\, T_{tt}(\rho)]\!]$. Since $\rho_v \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \sim_{evt} \rho_T \frown \langle fail, X_T\rangle$, then ρ_v is of the form $\langle inc, e\rangle \frown \rho_3$, for some ρ_3. We therefore have

$\langle inc, e\rangle \frown \rho_3 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![\, \widehat{Q}\ [\![\ \Sigma\ ]\!]\ (inc \rightarrow_U e \rightarrow_U T_{tt}(\rho))]\!]$

$\Rightarrow \langle e\rangle \frown \rho_3 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![\, \widehat{Q}\ [\![\ \Sigma\ ]\!]\ (e \rightarrow_U T_{tt}(\rho))]\!]$ [semantics of parallelism: *inc* is not in Σ]

$\Rightarrow \rho_3 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![\, \widehat{Q}\ \text{after}\ \langle e\rangle\ [\![\ \Sigma\ ]\!]\ T_{tt}(\rho)]\!]$

[semantics of parallelism: e is in Σ and $T_{tt}(\rho) = (e \rightarrow_U T_{tt}(\rho))$ after $\langle e\rangle$]

$\Rightarrow \exists\, \rho_4 : TTrace \bullet \rho_4 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![(\widehat{Q}\ \text{after}\ \langle e\rangle\ [\![\ \Sigma\ ]\!]\ T_{tt}(\rho)) \setminus \Sigma]\!]$

[$hideTrace\ \Sigma\ (\rho_3 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle) \neq \emptyset$ due to $satref(\rho_3, \Sigma)$]

$\Rightarrow \rho \in tt[\![Q\ \text{after}\ \langle e\rangle]\!]$ [induction hypothesis]

$\Rightarrow \langle e\rangle \frown \rho \in tt[\![Q]\!]$ [definition of after]

Case $\langle e\rangle \frown \rho$ ($\Leftarrow$).

$\langle e\rangle \frown \rho \in tt[\![Q]\!]$

$\Rightarrow \rho \in tt[\![Q\ \text{after}\ \langle e\rangle]\!]$ [definition of after]

$\Rightarrow \exists\, \rho_h : TTTrace \bullet \rho_h \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in tt[\![(\widehat{Q}\ \text{after}\ \langle e\rangle\ [\![\ \Sigma\ ]\!]\ T_{tt}(\rho)) \setminus \Sigma]\!]$ [induction hypothesis]

$\Rightarrow \exists\, \rho_h, \rho_v : TTTrace \bullet$ [semantics of hiding]
$\rho_v \in tt[\![\, \widehat{Q}\ \text{after}\ \langle e\rangle\ [\![\ \Sigma\ ]\!]\ T_{tt}(\rho)]\!] \wedge satref(\rho_v, \Sigma) \wedge \rho_h \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in hideTrace\ (\Sigma)\ \rho_v$

$\Rightarrow \exists\, \rho_h, \rho_v : TTTrace \bullet$ [semantics of parallelism: $e \in \Sigma$]
$\langle e\rangle \frown \rho_v \frown \in tt[\![\, \widehat{Q}\ [\![\ \Sigma\ ]\!]\ e \rightarrow_U T_{tt}(\rho)]\!] \wedge satref(\rho_v, \Sigma) \wedge \rho_h \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in hideTrace\ (\Sigma)\ \rho_v$

$\Rightarrow \exists\, \rho_h, \rho_v : TTTrace \bullet$ [semantics of parallelism: $inc \notin \Sigma$]
$\langle inc\rangle \frown \langle e\rangle \frown \rho_v \in tt[\![\, \widehat{Q}\ [\![\ \Sigma\ ]\!]\ inc \rightarrow_U e \rightarrow_U T_{tt}(\rho)]\!] \wedge$
$satref(\rho_v, \Sigma) \wedge \rho_h \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V\rangle \in hideTrace\ (\Sigma)\ \rho_v$

$$\Rightarrow \quad \exists\, \rho_h, \rho_v : TTTrace \bullet \qquad \text{[definition of } T_{tt}\text{]}$$
$$\langle inc \rangle \frown \langle e \rangle \frown \rho_v \in tt[\![\widehat{Q} \,|[\, \Sigma \,]|\, T_{tt}(\langle e \rangle \frown \rho)]\!] \wedge satref\,(\rho_v, \Sigma) \wedge \rho_h \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V \rangle \in hideTrace\,(\Sigma)\,\rho_v$$

Since $\rho_h \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V \rangle \in hideTrace\,(\Sigma)\,\rho$, and $hideTrace\,(\Sigma)\,(\langle inc, e \rangle) = \{\langle inc \rangle\} \neq \emptyset$, then by definition of *hideTrace*, there is a ρ_1 such that $\rho_1 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V \rangle \in hideTrace\,(\Sigma)\,(\langle inc, e \rangle \frown \rho)$. The semantics of the hiding operator thus gives $\rho_1 \frown \langle fail, \Sigma^{\checkmark}_{tock} \cup V \rangle \in tt[\![\widehat{Q} \,|[\, \Sigma \,]|\, T_{tt}(\langle e \rangle \frown \rho) \setminus \Sigma]\!]$.

□

The above theorem is the main result in this section, used next to establish exhaustiveness of our test suites.

6.2 Test suites: semantics based on ✓-tock traces

As the final ingredient of our testing theory, we need to provide collections of tests that are complete for our refinement relations of interest. Since our tests correspond to individual traces, a test suite can be identified with a collection of traces. The general idea is to include traces disallowed by the specification, while keeping the test suite as small as possible. To avoid redundancies, we therefore include only traces that are minimal with respect to the prefix order $\lesssim$.

Thus, given a specification P, we first define the "abstract" test suites (that is, collections of traces) for P with respect to the input-output and standard ✓-tock refinement as follows.

Definition 6.7.

$$\mathsf{TS}^{\mathcal{O}}_{iott}(P) \mathrel{\widehat{=}} min_{\lesssim}\{\rho : TTTrace \mid \rho \notin iott^{\mathcal{O}}[\![P]\!] \bullet addOuts^{\mathcal{O}}(\rho)\}$$
$$\mathsf{TS}_{tt}(P) \mathrel{\widehat{=}} min_{\lesssim}\{\rho : TTTrace \mid \rho \notin tt[\![P]\!] \bullet \rho\}$$

We observe that these sets may be infinite and there then remains the problem of choosing some appropriate finite subsets to use in testing. This is a topic that we touch upon under our discussion of future work in Section 7.

With the next example, we illustrate why minimality is of interest.

Example 6.8. We consider again *RD* from Example 3.2, and the trace $\rho_1 = \langle takeoff, \Sigma^{\checkmark}, tock, found, \Sigma^{\checkmark}, tock \rangle$. Since, as explained in Example 6.5, the prefix $\rho_2 = \langle takeoff, \Sigma^{\checkmark}, tock, found \rangle$ of ρ_1 is forbidden, so is its extension ρ_1. If the test for the shorter trace ρ_2 fails, there is no need to execute the test for ρ_1. On the other hand, if the test for ρ_2 passes, the test for ρ_1 is inconclusive. This is guaranteed because, if the test for ρ_2 passes, it means that the test execution stops before the *found* event. In this case, the test for the ρ_1 deadlocks at that same point, where the verdict is *inc*. In either case, the test for the longer ρ_1 does not add any information: it is useless. □

In addition, to test for input-output ✓-tock refinement, we require output-saturated traces.

Example 6.9. We consider again *RD* from Example 3.2, and the trace $\rho_3 = \langle \emptyset \rangle$, which is forbidden, since the output *takeoff* is possible at the start, and is minimal with respect to $\lesssim$. The test $T_{tt}(\rho_3)$ raises a *fail* verdict and then offers only *ticktest* as a possibility for the *SUT* (see case (2) in the definition of $T_{tt}(\rho)$). If the SUT does not terminate, the *fail* verdict stands. This is not sound if the SUT can provide an output, for example, *takeoff*, as specified. □

The corresponding test suites can now be defined in a straightforward way.

Definition 6.10. $Exhaust^{\mathcal{O}}_{iott}(P) \mathrel{\widehat{=}} \{\rho \in \mathsf{TS}^{\mathcal{O}}_{iott}(P) \bullet T_{tt}(\rho)\}$ and $Exhaust_{tt}(P) \mathrel{\widehat{=}} \{\rho \in \mathsf{TS}_{tt}(P) \bullet T_{tt}(\rho)\}$

Exhaustiveness, and, therefore, soundness, is established by the following theorem.

Theorem 6.11. *For any specification P, the test suites $Exhaust^{O}_{iott}(P)$ and $Exhaust_{tt}(P)$ are exhaustive for, respectively, input-output and standard ✓-tock refinement.*

$$P \not\sqsubseteq_{IOTT} Q \Leftrightarrow \exists\, T : Exhaust^{O}_{iott}(P) \bullet Q \text{ fails } T$$
$$P \not\sqsubseteq_{TT} Q \Leftrightarrow \exists\, T : Exhaust_{tt}(P) \bullet Q \text{ fails } T$$

Proof. We show below the proof for $\sqsubseteq_{IOTT}$.

$$P \not\sqsubseteq_{IOTT} Q$$
$$\Leftrightarrow \exists\, \rho_1 : TTTrace \bullet \rho_1 \notin iott^{O}[\![P]\!] \wedge \rho_1 \in iott^{O}[\![Q]\!] \quad \text{[definition of } \sqsubseteq_{TT}\text{]}$$
$$\Leftrightarrow \exists\, \rho_1 : TTTrace \bullet \quad \text{[definition of } iott\text{]}$$
$$\rho_1 \notin \{\rho_1 : TTTrace \mid addOuts^{O}(\rho_1) \in tt[\![P]\!]\} \wedge \rho_1 \in \{\rho_1 : TTTrace \mid addOuts^{O}(\rho_1) \in tt[\![Q]\!]\}$$
$$\Leftrightarrow \exists\, \rho_1 : TTTrace \bullet \quad \text{[property of sets]}$$
$$\rho_1 \in \{\rho_1 : TTTrace \mid addOuts^{O}(\rho_1) \notin tt[\![P]\!]\} \wedge \rho_1 \in \{\rho_1 : TTTrace \mid addOuts^{O}(\rho_1) \in tt[\![Q]\!]\}$$
$$\Leftrightarrow \exists\, \rho_2 : TTTrace \bullet \rho_2 \in \{\rho_1 : TTTrace \mid addOuts^{O}(\rho_1) \notin tt[\![P]\!] \bullet addOuts^{O}(\rho_1)\} \wedge \rho_2 \in tt[\![Q]\!]$$
$$\text{[property of sets]}$$
$$\Leftrightarrow \exists\, \rho_3 : TTTrace \bullet \rho_3 \in min_{\lesssim}\{\rho : TTTrace \mid addOuts^{O}(\rho) \notin tt[\![P]\!] \bullet addOuts^{O}(\rho)\} \wedge \rho_3 \in tt[\![Q]\!] \quad \textbf{[TT1]}$$
$$\Leftrightarrow \exists\, \rho_3 : \mathsf{TS}^{O}_{iott}(P) \bullet \rho_3 \in tt[\![Q]\!] \quad \text{[definitions of } iott^{O}[\![P]\!] \text{ and } \mathsf{TS}^{O}_{iott}(P)\text{]}$$
$$\Leftrightarrow \exists\, \rho_3 : \mathsf{TS}^{O}_{iott}(P) \bullet Q \text{ fails } T_{tt}(\rho_3) \quad \text{[Theorem 6.6]}$$
$$\Leftrightarrow \exists\, T : Exhaust^{O}_{iott}(P) \bullet Q \text{ fails } T \quad \text{[definition of } Exhaust^{O}_{iott}(P)\text{]}$$

The proof for $\sqsubseteq_{TT}$ is similar, since Theorem 6.6 applies to traces ρ of $tt[\![Q]\!]$ as well.

□

It is important to observe that our tests deal with deadlines as illustrated by the following example.

Example 6.12. We consider as specification the process *RDFL* from Example 3.4 and the trace $\rho = \langle \emptyset, tock, \emptyset, tock \rangle$, which is forbidden for any refinement of *RDFL*, as it violates the deadline of at most one *tock* before the event *found* happens. We recall that, in this example, $O = \{takeoff, move, land\}$, hence we need to consider the trace $addOuts^{O}(\rho) = \langle \{takeoff, move, land\}, tock, \{takeoff, move, land\}, tock \rangle$. Its test case can be calculated as follows.

$$T_{tt}(addOuts^{O}(\rho))$$
$$= T_{tt}(\langle \{takeoff, move, land\}, tock, \{takeoff, move, land\}, tock \rangle)$$
$$= inc \rightarrow_U \begin{pmatrix} \Box\, e : O \bullet e \rightarrow \mathbf{Stop}_U \\ \Box \\ ticktest \rightarrow \mathbf{Stop}_U \end{pmatrix} \triangle_1\ T_{tt}(\langle \{takeoff, move, land\}, tock \rangle) \quad \text{[case (5), } t \neq \langle\rangle\text{]}$$
$$= inc \rightarrow_U \begin{pmatrix} \Box\, e : O \bullet e \rightarrow \mathbf{Stop}_U \\ \Box \\ ticktest \rightarrow \mathbf{Stop}_U \end{pmatrix} \triangle_1 \left(pass \rightarrow_U \begin{pmatrix} \Box\, e : O \bullet e \rightarrow \mathbf{Stop}_U \\ \Box \\ ticktest \rightarrow \mathbf{Stop}_U \end{pmatrix} \triangle_1\ T_{tt}(\langle\rangle) \right) \quad \text{[case (5), } t = \langle\rangle\text{]}$$

$$= inc \rightarrow_U \left(\begin{array}{l} \Box\, e : O \bullet e \rightarrow \mathbf{Stop}_U \\ \Box \\ ticktest \rightarrow \mathbf{Stop}_U \end{array} \right) \triangle_1 \left(pass \rightarrow_U \left(\begin{array}{l} \Box\, e : O \bullet e \rightarrow \mathbf{Stop}_U \\ \Box \\ ticktest \rightarrow \mathbf{Stop}_U \end{array} \right) \triangle_1 fail \rightarrow_U STOP \right) \qquad \text{[case (1)]}$$

We suppose now that we have the following implementation $RDFL_1 = found \rightarrow land \rightarrow \mathbf{Stop}$. This is not a correct refinement of *RDFL*, since it allows an arbitrary number of time units to pass before *found* happens; in particular, it exhibits the trace ρ. The erroneous behaviour can be discovered using the test case $T_{tt}(addOuts^O(\rho))$ as shown below.

$$Execution(RDFL_1, T_{tt}(addOuts^O(\rho)))$$
$$= ((RDFL_1;\ ticktest \rightarrow_U \mathbf{Stop}_U) \,[\![\Sigma]\!]\, T_{tt}(addOuts^O(\rho))) \setminus \Sigma$$

We need to show that there is at least one trace of the above process whose last event before deadlock is *fail*. We consider first the process without the hiding and proceed as follows.

$$(RDFL_1;\ ticktest \rightarrow_U \mathbf{Stop}_U) \,[\![\Sigma]\!]\, T_{tt}(addOuts^O(\rho))$$
$$= RDFL_1 \,[\![\Sigma]\!]\, T_{tt}(addOuts^O(\rho)) \qquad \text{[no trace of } RDFL_1 \text{ contains } \checkmark\text{]}$$

The semantics of parallel composition is defined using a parallel operator for traces $[\![\_]\!]^T$. We therefore need to indicate two traces ρ_1 and ρ_2 of the component processes such that their composition $\rho_1 \,[\![\Sigma]\!]^T\, \rho_2$ contains a trace with the desired property. From the semantics of CSP operators we obtain the following natural candidates.

$$\rho_1 = \langle O, tock, O, tock, \Sigma \rangle \in tt[\![RDFL_1]\!]$$
$$\rho_2 = \langle inc, \Sigma \setminus O, tock, pass, \Sigma \setminus O, tock, fail, \Sigma^{\checkmark}_{tock} \cup V \rangle \in tt[\![T_{tt}(addOuts^O(\rho))]\!]$$

Furthermore, using the definition of $[\![\_]\!]^T$, we can calculate a trace for the parallelism.

$$\langle inc, \Sigma, tock, pass, \Sigma, tock, fail, \Sigma^{\checkmark}_{tock} \cup V \rangle \in \rho_1 \,[\![\Sigma]\!]^T\, \rho_2$$
$$\Rightarrow \langle inc, \Sigma, tock, pass, \Sigma, tock, fail, \Sigma^{\checkmark}_{tock} \cup V \rangle \in tt[\![(RDFL_1;\ ticktest \rightarrow_U \mathbf{Stop}_U) \,[\![\Sigma]\!]\, T_{tt}(addOuts^O(\rho))]\!]$$

Since all events occurring in the above trace are not in Σ, and all the refusals subsume Σ, we finally obtain

$$\langle inc, \Sigma, tock, pass, \Sigma, tock, fail, \Sigma^{\checkmark}_{tock} \cup V \rangle \in ((RDFL_1;\ ticktest \rightarrow_U \mathbf{Stop}_U) \,[\![\Sigma]\!]\, T_{tt}(addOuts^O(\rho))) \setminus \Sigma$$

according to the semantics of hiding. Hence we can conclude that $RDFL_1$ *fails* $T_{tt}(addOuts^O(\rho))$. □

Although timelocks are purely specification devices used to capture deadlines, our theory can handle such specifications. It can also be used for a weaker conformance relation, namely, traces refinement, as discussed in the next section.

6.3 Event traces

Here, we briefly discuss a semantics characterised by traces containing only events in $\Sigma^{\checkmark}_{tock}$. This type of semantics is typically referred to in the literature as a trace semantics; in our context, for clarity and notational convenience, we use the term event traces. Formally, the set *ETrace* of event traces can be defined as follows.

Definition 6.13.

$$ETrace == \{\rho : \mathrm{seq}\ Obs \mid \mathrm{ran}\ \rho \subseteq \mathrm{ran}\ evt\}$$

We observe that *ETrace* is not a subset of *TTTrace*, as event traces in *ETrace* do not contain any refusals, whereas traces in *TTTrace* are required to have refusals preceding every *tock* event.

The presence of *tock* gives rise to a trace semantics, induced by the semantics defined by the ✓-tock model, that has more observational power than the standard trace semantics of CSP. Since a *tock* can happen in a stable state only, its occurrence implicitly entails observation of at least an empty refusal $\emptyset$. In the input-output setting, one can make an even stronger inference, as refusal of all outputs must have occurred before each *tock*.

The above observations suggest a straightforward translation operator, called et2iott below, from *ETrace* to *TTTrace*, yielding ✓-tock traces corresponding to a given event trace.

Definition 6.14.

$$\text{et2iott}^O(\langle\rangle) = \langle\rangle$$

$$\text{et2iott}^O(\langle e\rangle \frown \rho) = \begin{cases} e \frown \text{et2iott}^O(\rho) \text{ if } e \neq tock \\ \langle O, tock\rangle \frown \text{et2iott}^O(\rho) \text{ if } e = tock \end{cases}$$

We can now conveniently formalise the input-output event trace semantics in terms of the ✓-tock semantics.

Definition 6.15. $eiott^O[[P]] \mathrel{\widehat{=}} \{\rho \in ETrace \mid \text{et2iott}^O(\rho) \in iott[[P]]\}$

In the definition of exhaustive test suits for traces refinement, we can apply an approach similar to that used for ✓-tock refinement in the earlier part of this section. We first define the test suite in terms of traces for a specification P.

$$\text{TS}^O_{eiott}(P) \mathrel{\widehat{=}} min_{\lesssim}\{\rho \in ETrace \wedge \rho \notin eiott^O[[P]] \bullet \text{et2iott}^O(\rho)\}$$

The definition of the corresponding CSP test suite is as follows.

$$Exhaust^O_{eiott}(P) \mathrel{\widehat{=}} \{\rho \in \text{TS}^O_{eiott}(P) \bullet T_{tt}(\rho)\}$$

We can then easily show that the above test suite is exhaustive.

Example 6.16. We recall the previous Example 6.12. The property considered there – violation of a deadline of at most one *tock* before the event *found* – can be expressed using a syntactically simpler event trace, namely $\widehat{\rho} = \langle tock, tock\rangle$. The corresponding process in the test suite $Exhaust^O_{eiott}(RDFL)$ is $T_{tt}(\text{et2iott}^O(\widehat{\rho}))$, where $\text{et2iott}^O(\widehat{\rho}) = \langle O, tock, O, tock\rangle$. Since the latter trace is equal to $addOuts^O(\rho)$ from Example 6.12, the same test can be used in both examples, as expected. □

In summary, our tests can be used also to test for traces refinement only. While the weaker traces-refinement relation does not require all the tests required to test for refinement, the notion of test is the same.

7 CONCLUSIONS

We have presented the first testing theory for timewise refinement available in the literature. Other refinement relations that take inputs and outputs into account have been presented, but none of them deal with time. By considering time, and inputs and outputs in CSP, we have a theory of testing that can form the basis for practical testing.

Existing testing theories for CSP (and its variants) [13, 40] take advantage of a core theorem that shows how refinement can be expressed in terms of traces refinement and another *conf* relation concerned just with deadlocks. Those testing theories provide two definitions of test cases (for testing for traces refinement and for *conf*) and two exhaustive test sets. This is advantageous in terms of formalisation, since the test cases for the weaker relations are simpler. In the context of *tock*-CSP, however, tests for traces refinement are no longer simple, because of the special nature of *tock*, which does not represent an interaction. To deal with *tock*, we have to deal with refusals anyway. It is

for this reason that, here, we deal directly with ✓-tock traces and input-output ✓-tock-refinement and consider the tests for traces refinement as a special case. An advantage is that we then have a single suite of more powerful tests that check for the conformance of interactions, time, and deadlocks.

In addition, in [15], since there is no possibility of observing timeouts, we use prioritisation to check refusals: the SUT events are prioritised, and, if they do not happen, we then can issue a verdict event. In that context, we use prioritisation also to handle termination. For *tock*-CSP, we can use timeouts instead of prioritisation. We, therefore, adopt a simpler definition of test execution, and handle termination via an extra special event *ticktest*.

The work in [40] adopts the standard traces and failures semantics of CSP. For a finite non-terminating CSP model, finite optimal test suites for checking traces and failures refinement are presented, and their exhaustiveness is proven. The fault domains for which failure detection can be guaranteed are specified by means of normalised transition graphs representing the failures semantics of finite-state CSP processes adopted in a popular model checker [25]. The definition of finite test suites for input-output *tock*-CSP is part of our agenda for future work. Importantly, Theorem 6.6 shows that our testing theory can be used to test for any trace not allowed by the specification; it does not apply only to the (set of disallowed) traces described in this paper. As a result, our testing theory can be used with any test generation algorithm that identifies traces that are not allowed by the specification.

CSP and a timed version of ioco have been considered in [8], where the authors define a new conformance relation called csptio. Like in our work, and as usual in formal theories of testing, both the specification and the SUT are assumed to be described in CSP. In [8], however, a normal form is considered for the process descriptions to reflect, in particular, the cyclic paradigm of data-flow reactive systems. On the other hand, both discrete and dense time are considered by combining CSP and SMT-solving technology. The goal in [8] is not to adopt refinement as a conformance relation, as we do here, but to use CSP technology to reason about a relation inspired by ioco. This work has been taken forward to underpin testing techniques based on controlled natural language [7].

We have used *tock*-CSP extensively to give semantics to domain-specific modelling languages for robotics [9]. In particular, all our software modelling languages have a *tock*-CSP semantics. In our future work, we will use the testing theory presented here to justify test-generation approaches based on models written in these languages.

For example, previously, we have used mutation testing [10] to generate tests from models written in the control-software design language RoboChart. The approach essentially creates mutants of the specification by seeding faults and then uses a model-checker to find a behaviour of a mutant that is not allowed by the specification. The tests generated, however, do not take into account time or inputs and outputs. We will revisit our approach and our examples to consider their rich set of time properties. We will also create the infrastructure to execute the tests and provide verdicts as specified here. As noted above, once we have a behaviour of a mutant that is not allowed by the specification, Theorem 6.6 shows that our function T_{tt} can be used to generate a test for this disallowed behaviour.

Finally, it is worth noting that there are at least two issues related to efficiency of testing. First, the tests we use are essentially sequential and there is potential to use other types of tests, such as those in the form of trees, to improve efficiency. As an example, we consider two disallowed behaviours $\rho_1 = \rho \frown \langle o_1 \rangle \frown \rho_3$ and $\rho_2 = \rho \frown \langle o_2 \rangle \frown \rho_4$ that have a common prefix and then have different outputs. Using sequential tests, as described in this paper, if we are testing for ρ_1 and the output o_2 is produced by the SUT after ρ then testing terminates with an inconclusive verdict. Instead, one could combine the two tests so that if o_1 is produced after ρ then the test continues as defined for $T_{tt}(\rho_1)$ and if o_2 is produced after ρ then the test continues as defined for $T_{tt}(\rho_2)$. General optimisations such as this are a problem for future, and have been tackled in the context of CSP previously [12]. Second, recent work has shown how redundancies can be eliminated in a set of tests [24] and it should be possible to use this approach to further improve efficiency.

ACKNOWLEDGEMENT

The authors would like to thank the RoboStar team, and Pedro Ribeiro, in particular, for useful discussions. Ana Cavalcanti and James Baxter are funded by the UK EPSRC (Engineering and Physical Sciences Research Council) under Grants No EP/M025756/1 and EP/R025479/1, and by the Royal Academy of Engineering under Grant No CiET1718/45. Maciej Gazda and Robert M. Hierons are funded by the EPSRC, under Grant No EP/R025134/1.

REFERENCES

[1] P. Armstrong, G. Lowe, J. Ouaknine, and A. W. Roscoe. 2012. Model checking Timed CSP. In *Festschrift for Howard Barringer*.

[2] J. Baxter, A. L. C. Cavalcanti, M. Gazda, and R. Hierons. 2022. *Testing using CSP models: time, inputs, and outputs – Extended version.* Technical Report. RoboStar Centre on Software Engineering for Robotics. Available at `robostar.cs.york.ac.uk/publications/reports/BCGH22.pdf`.

[3] J. Baxter, P. Ribeiro, and A. L. C. Cavalcanti. 2021. Sound reasoning in tock-CSP. *Acta Informatica* (2021). https://doi.org/10.1007/s00236-020-00394-3 online April 2021.

[4] P. Bos, R. Janssen, and J. Moerman. 2019. n-Complete test suites for IOCO. *Software Quality Journal* 27, 2 (2019), 563–588.

[5] I. B. Bourdonov, A. Kossatchev, and V. V. Kuliamin. 2006. Formal Conformance Testing of Systems with Refused Inputs and Forbidden Actions. *Electronic Notes in Theoretical Computer Science* 164, 4 (2006), 83–96.

[6] E. Brinksma, L. Heerink, and J. Tretmans. 1998. Factorized Test Generation for Multi-Input/Output Transition Systems. In *11th IFIP Workshop on Testing of Communicating Systems*. Kluwer Academic Publishers, 67–82.

[7] G. Carvalho. 2016. *NAT2TEST: generating test cases from natural language requirements based on CSP.* Ph.D. Dissertation. Universidade Federal de Pernambuco. repositorio.ufpe.br/handle/123456789/17929

[8] G. Carvalho, A. C. A. Sampaio, and A. C. MotaAlexandre. 2013. A CSP Timed Input-Output Relation and a Strategy for Mechanised Conformance Verification. In *Formal Methods and Software Engineering*, L. Groves and J. Sun (Eds.). Springer Berlin Heidelberg, 148–164.

[9] A. L. C. Cavalcanti, W. Barnett, J. Baxter, G. Carvalho, M. C. Filho, A. Miyazawa, P. Ribeiro, and A. C. A. Sampaio. 2021. *RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing.* Springer International Publishing, 249–293. https://doi.org/10.1007/978-3-030-66494-7_9

[10] A. L. C. Cavalcanti, J. Baxter, R. M. Hierons, and R. Lefticaru. 2019. Testing Robots using CSP. In *Tests and Proofs*, D. Beyer and C. Keller (Eds.). Springer, 21–38. https://doi.org/doi.org/10.1007/978-3-030-31157-5_2

[11] A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. 2011. From Control Law Diagrams to Ada via *Circus*. *Formal Aspects of Computing* 23, 4 (2011), 465–512. https://doi.org/10.1007/s00165-010-0170-3

[12] A. L. C. Cavalcanti and M.-C. Gaudel. 2007. Testing for Refinement in CSP. In *9th International Conference on Formal Engineering Methods (Lecture Notes in Computer Science, Vol. 4789)*. Springer-Verlag, 151–170. https://doi.org/10.1007/978-3-540-76650-6_10

[13] A. L. C. Cavalcanti and M.-C. Gaudel. 2011. Testing for Refinement in *Circus*. *Acta Informatica* 48, 2 (2011), 97–147. https://doi.org/10.1007/s00236-011-0133-z

[14] A. L. C. Cavalcanti, M.-C. Gaudel, and R. M. Hierons. 2011. Conformance Relations for Distributed Testing based on CSP. In *IFIP International Conference on Testing Software and Systems (Lecture Notes in Computer Science)*, B. Wolff and F. Zaidi (Eds.). Springer-Verlag. https://doi.org/10.1007/978-3-642-24580-0_5

[15] A. L. C. Cavalcanti, R. Hierons, and S. Nogueira. 2020. Inputs and outputs in CSP: a model and a testing theory. *ACM Transactions on Computational Logic* (2020). https://doi.org/10.1145/3379508

[16] A. L. C. Cavalcanti and R. M. Hierons. 2013. Testing with Inputs and Outputs in CSP. In *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science, Vol. 7793)*. 359–374. https://doi.org/10.1007/978-3-642-37057-1_26

[17] A. L. C. Cavalcanti, R. M. Hierons, S. Nogueira, and A. C. A. Sampaio. 2016. A Suspension-Trace Semantics for CSP. In *International Symposium on Theoretical Aspects of Software Engineering*. 3–13. https://doi.org/10.1109/TASE.2016.9 Invited paper.

[18] A. L. C. Cavalcanti, A. C. A. Sampaio, A. Miyazawa, P. Ribeiro, M. Conserva Filho, A. Didier, W. Li, and J. Timmis. 2019. Verified simulation for robotics. *Science of Computer Programming* 174 (2019), 1–37. https://doi.org/doi.org/10.1016/j.scico.2019.01.004

[19] A. David, K. G. Larsen, S. Li, and B. Nielsen. 2008. A Game-Theoretic Approach to Real-Time System Testing. In *Design, Automation and Test in Europe*, Donatella Sciuto (Ed.). ACM, 486–491.

[20] A. David, K. G. Larsen, S. Li, and B. Nielsen. 2009. Timed Testing under Partial Observability. In *2nd International Conference on Software Testing Verification and Validation*. IEEE Computer Society, 61–70.

[21] J. Davies. 1993. *Specification and Proof in Real-time CSP.* Cambridge University Press.

[22] N. Evans and S. Schneider. 2000. Analysing time dependent security properties in CSP using PVS. In *European Symposium on Research in Computer Security*. Springer, 222–237.

[23] M. Conserva Filho, M. V. M. Oliveira, A. C. A. Sampaio, and A. L. C. Cavalcanti. 2016. Local Livelock Analysis of Component-Based Models. In *International Conference on Formal Engineering Methods (Lecture Notes in Computer Science, Vol. 10009)*. Springer, 279–295. https://doi.org/10.1007/978-3-319-47846-3_18

[24] M. Gazda and R. M. Hierons. 2021. Removing Redundant Refusals: Minimal Complete Test Suites for Failure Trace Semantics. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, 1–13.

[25] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. 2014. FDR3 - A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*. 187–201.

[26] T. Göthel and B. Bartels. 2015. Modular Design and Verification of Distributed Adaptive Real-Time Systems. In *Nature of Computation and Communication*, P. G. Vinh, E. Vassev, and M. Hinchey (Eds.). Springer International Publishing, 3–12.

[27] Y. Isobe, F. Moller, H. N. Nguyen, and M. Roggenbach. 2012. Safety and line capacity in railways–an approach in Timed CSP. In *International Conference on Integrated Formal Methods*. Springer, 54–68.

[28] C. Jard and T. Jéron. 2005. TGV: theory, principles and algorithms, A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer* 7, 4 (2005), 297–315.

[29] T. Kahsai, M. Roggenbach, and B.-H. Schlingloff. 2007. Specification-based testing for refinement. In *5th IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society, 237–246.

[30] S. A. Kharmeh, K. Eder, and D. May. 2011. A design-for-verification framework for a configurable performance-critical communication interface. In *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 335–351.

[31] M. Krichen. 2010. A Formal Framework for Conformance Testing of Distributed Real-Time Systems. In *Principles of Distributed Systems*, C. Lu, T. Masuzawa, and M. Mosbah (Eds.). Lecture Notes in Computer Science, Vol. 6490. Springer, 139–142.

[32] M. Krichen and S. Tripakis. 2004. Black-Box Conformance Testing for Real-Time Systems. In *11th International SPIN Workshop on Model Checking Software (Lecture Notes in Computer Science, Vol. 2989)*, S. Graf and L. Mounier (Eds.). Springer, 109–126.

[33] K. Larsen, M. Mikucionis, and B. Nielsen. 2005. Online Testing of Real-time Systems Using UPPAAL. In *Formal Approaches to Software Testing*, J. Grabowski and B. Nielsen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–94.

[34] G. Lowe and J. Ouaknine. 2006. On timed models and full abstraction. *Electronic Notes in Theoretical Computer Science* 155 (2006), 497–519.

[35] A. Miyazawa and A. L. C. Cavalcanti. 2012. Refinement-oriented models of Stateflow charts. *Science of Computer Programming* 77, 10-11 (2012), 1151–1177. https://doi.org/10.1016/j.scico.2011.07.007

[36] A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, J. Timmis, and J. C. P. Woodcock. 2019. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling* 18, 5 (2019), 3097–3149. https://doi.org/doi.org/10.1007/s10270-018-00710-z

[37] S. Nogueira, A. C. A. Sampaio, and A. C. Mota. 2014. Test generation from state based use case models. *Formal Aspects of Computing* 26, 3 (2014), 441–490.

[38] J. Ouaknine. 2001. *Discrete analysis of continuous behaviour in real-time concurrent systems*. Ph.D. Dissertation. Oxford University.

[39] S. L. C. Paiva, A. Simão, M. Varshosaz, and M. R. Mousavi. 2016. Complete IOCO test cases: a case study. In *7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. ACM, 38–44.

[40] J. Peleska, W. l. Huang, and A. L. C. Cavalcanti. 2019. Finite complete suites for CSP refinement testing. *Science of Computer Programming* 179 (2019), 1 – 23. https://doi.org/doi.org/10.1016/j.scico.2019.04.004

[41] G. M. Reed and A. W. Roscoe. 1988. A timed model for communicating sequential processes. *Theoretical Computer Science* 58 (1988), 249–261.

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

[43] A. W. Roscoe. 2011. *Understanding Concurrent Systems*. Springer.

[44] A. C. A. Sampaio, S. Nogueira, A. Mota, and Y. Isobe. 2014. Sound and mechanised compositional verification of input-output conformance. *Software Testing, Verification and Reliability* 24, 4 (2014), 289–319.

[45] J. Schmaltz and J. Tretmans. [n.d.]. On Conformance Testing for Timed Systems. In *6th International Conference on Formal Modeling and Analysis of Timed Systems (Lecture Notes in Computer Science, Vol. 5215)*. Springer, 250–264.

[46] S. Schneider. 2000. *Concurrent and Real-time Systems: The CSP Approach*. Wiley.

[47] J. Sun, Y. Liu, and J. S. Dong. 2008. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (Communications in Computer and Information Science, Vol. 17)*. Springer, 307–322.

[48] The MathWorks,Inc. [n.d.]. *Simulink*. The MathWorks,Inc. `www.mathworks.com/products/simulink`.

[49] The MathWorks,Inc. [n.d.]. *Stateflow and Stateflow Coder 7 User's Guide*. The MathWorks,Inc. `www.mathworks.com/products`.

[50] J. Tretmans. 1992. *A formal approach to conformance testing*. Ph.D. Dissertation. University of Twente, Enschede, The Netherlands.

[51] J. Tretmans. 1996. Test Generation with Inputs, Outputs, and Quiescence. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science, Vol. 1055)*. Springer-Verlag, 127–146.

[52] J. Tretmans. 1996. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software - Concepts and Tools* 17, 3 (1996), 103–120.

[53] J. Tretmans. 2008. Formal Methods and Testing. Springer-Verlag, Chapter Model Based Testing with Labelled Transition Systems, 1–38.

[54] J. Tretmans and E. Brinksma. 2003. TorX: Automated Model Based Testing. In *1st European Conference on Model-Driven Software Engineering*. 13–25.

[55] M. van der Bijl, A. Rensink, and J. Tretmans. 2004. Compositional Testing with ioco. In *Formal Approaches to Software Testing*, A. Petrenko and A. UlrichAndreas (Eds.). Lecture Notes in Computer Science, Vol. 2931. Springer, 86–100.

[56] M. Weiglhofer and B. Aichernig. 2010. Unifying Input Output Conformance. In *Unifying Theories of Programming*, A. Butterfield (Ed.). Lecture Notes in Computer Science, Vol. 5713. Springer, 181–201.

[57] J. C. P. Woodcock and J. Davies. 1996. *Using Z - Specification, Refinement, and Proof*. Prentice-Hall.

A DEFINITIONS FROM *tock*-CSP DENOTATIONAL SEMANTICS

Here, we reproduce the semantics of *tock*-CSP [3], including some operators used in the input-output semantics here.

Trace prefix relation

$$\_ \lesssim \_ : Obs \leftrightarrow Obs$$

$$\forall \sigma, \rho : \operatorname{seq} Obs;\ e : \Sigma^{\checkmark}_{tock};\ X, Y : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet$$
$$\langle\rangle \lesssim \rho \land (\sigma \lesssim \rho \Rightarrow \langle evt\ e\rangle \frown \sigma \lesssim \langle evt\ e\rangle \frown \rho) \land (\sigma \lesssim \rho \land X \subseteq Y \Rightarrow \langle ref\ X\rangle \frown \sigma \lesssim \langle ref\ Y\rangle \frown \rho)$$

Divergence and termination

$$tt[\![\mathbf{div}]\!] = \{\langle\rangle\}$$

$$tt[\![\mathbf{Skip}]\!] = \{\langle\rangle, \langle evt\ \checkmark\rangle\}$$

Timed deadlock

$$tt[\![\mathbf{Stop}]\!] = tocks\,\Sigma^{\checkmark} \cup \{\rho : tocks\,\Sigma^{\checkmark};\ X : \mathbb{P}\,\Sigma^{\checkmark} \bullet \rho \frown \langle ref\ X\rangle\}$$

$$tocks : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \to \mathbb{P}\ TTTrace$$

$$\forall X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet \langle\rangle \in tocks\,X \land (\forall \rho : tocks\,X;\ Y : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \mid Y \subseteq X \bullet \langle ref\ Y, evt\ tock\rangle \frown \rho \in tocks\,X)$$

Timestop

$$tt[\![\mathbf{Stop}_U]\!] = \{\langle\rangle\} \cup \{X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet \langle ref\ X\rangle\}$$

Delay

$$\begin{aligned} tt[\![\mathbf{Wait}\ n]\!] = \quad & \{\rho : tocks\,\Sigma^{\checkmark} \mid \#(\rho \upharpoonright \{evt\ tock\}) \leq n\} \\ & \cup \{\rho : tocks\,\Sigma^{\checkmark};\ X : \mathbb{P}\,\Sigma^{\checkmark} \mid \#(\rho \upharpoonright \{evt\ tock\}) < n \bullet \rho \frown \langle ref\ X\rangle\} \\ & \cup \{\rho : tocks\,\Sigma^{\checkmark} \mid \#(\rho \upharpoonright \{evt\ tock\}) = n \bullet \rho \frown \langle evt\ \checkmark\rangle\} \end{aligned}$$

Prefixing

$$\begin{aligned} tt[\![e \to P]\!] = \quad & tocks\,(\Sigma^{\checkmark} \setminus \{e\}) \\ & \cup \{\rho_1 : tocks\,(\Sigma^{\checkmark} \setminus \{e\});\ X : \mathbb{P}(\Sigma^{\checkmark} \setminus \{e\}) \bullet \rho_1 \frown \langle ref\ X\rangle\} \\ & \cup \{\rho_1 : tocks\,(\Sigma^{\checkmark} \setminus \{e\});\ \rho_2 : tt[\![P]\!] \mid e \neq tock \bullet \rho_1 \frown \langle evt\ e\rangle \frown \rho_2\} \\ & \cup \{\rho_1 : tocks\,\Sigma^{\checkmark};\ X : \mathbb{P}\,\Sigma^{\checkmark};\ \rho_2 : tt[\![P]\!] \mid e = tock \bullet \rho_1 \frown \langle ref\ X, evt\ tock\rangle \frown \rho_2\} \end{aligned}$$

Internal choice

$$tt[\![P \sqcap Q]\!] = tt[\![P]\!] \cup tt[\![Q]\!]$$

External choice

$$
\begin{aligned}
tt[\![P \,\Box\, Q]\!] = \; & \{\rho_1 : tocks\, \Sigma^{\checkmark}_{tock};\ \rho_2, \rho_3, \rho_4 : TTTrace \mid \\
& \quad \rho_1 \frown \rho_2 \in tt[\![P]\!] \land \rho_1 \frown \rho_3 \in tt[\![Q]\!] \land \\
& \quad (\forall \rho_5 : tocks\, \Sigma^{\checkmark}_{tock} \bullet \rho_5 \text{ prefix } \rho_1 \frown \rho_2 \Rightarrow \rho_5 \text{ prefix } \rho_1) \land \\
& \quad (\forall \rho_5 : tocks\, \Sigma^{\checkmark}_{tock} \bullet \rho_5 \text{ prefix } \rho_1 \frown \rho_3 \Rightarrow \rho_5 \text{ prefix } \rho_1) \land \\
& \quad (\forall X : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet \rho_2 = \langle ref\ X \rangle \Rightarrow \exists Y : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet \rho_3 = \langle ref\ Y \rangle \land X \setminus \{tock\} = Y \setminus \{tock\}) \land \\
& \quad (\forall X : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet \rho_3 = \langle ref\ X \rangle \Rightarrow \exists Y : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet \rho_2 = \langle ref\ Y \rangle \land X \setminus \{tock\} = Y \setminus \{tock\}) \land \\
& \quad (\rho_4 = \rho_1 \frown \rho_2 \lor \rho_4 = \rho_1 \frown \rho_3) \\
& \bullet \rho_4 \\
& \}
\end{aligned}
$$

Sequence

$$
\begin{aligned}
tt[\![P;\ Q]\!] = \; & \{\rho_1 : tt[\![P]\!] \mid \neg\, (\exists \rho_2 : TTTrace \bullet \rho_1 = \rho_2 \frown \langle evt\ \checkmark \rangle)\} \\
& \cup \{\rho_1, \rho_2 : TTTrace \mid \rho_1 \frown \langle evt\ \checkmark \rangle \in tt[\![P]\!] \land \rho_2 \in tt[\![Q]\!] \bullet \rho_1 \frown \rho_2\}
\end{aligned}
$$

Interrupt

$$
\begin{aligned}
& tt[\![P \,\triangle\, Q]\!] = \\
& \quad \{\rho_1 : TTTrace;\ \rho_2 : tt[\![Q]\!] \mid \rho_1 \frown \langle evt\ \checkmark \rangle \in tt[\![P]\!] \land fTock\, \rho_1 = \rho_2 \bullet \rho_1 \frown \langle evt\ \checkmark \rangle\} \\
& \quad \cup \; \{\rho_1, \rho_2 : TTTrace;\ X, Y, Z : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \mid \\
& \qquad \rho_1 \frown \langle ref\ X \rangle \in tt[\![P]\!] \land \rho_2 \frown \langle ref\ Y \rangle \in tt[\![Q]\!] \land fTock\, \rho_1 = \rho_2 \land Z \subseteq X \cup Y \land X \setminus \{tock\} = Y \setminus \{tock\} \\
& \qquad \bullet \rho_1 \frown \langle ref\ Z \rangle \\
& \quad \} \\
& \quad \cup \; \{\rho_1 : tt[\![P]\!];\ \rho_2, \rho_3 : TTTrace \mid \\
& \qquad (\neg\ \exists \phi : \text{seq } Obs \bullet \rho_1 = \phi \frown \langle evt\ \checkmark \rangle) \land (\neg\ \exists \phi : \text{seq } Obs;\ X : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet \rho_1 = \phi \frown \langle ref\ X \rangle) \land \\
& \qquad fTock\, \rho_1 = \rho_2 \land \rho_2 \frown \rho_3 \in tt[\![Q]\!] \land (\neg\ \exists \phi : \text{seq } Obs;\ X : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet \rho_3 = \langle ref\ X \rangle \frown \phi) \\
& \qquad \bullet \rho_1 \frown \rho_3 \\
& \quad \}
\end{aligned}
$$

$$
\begin{aligned}
& fTock : TTTrace \to TTTrace \\
& \hline
& fTock\, \langle\rangle = \langle\rangle \land \forall X : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet fTock\, \langle ref\ X \rangle = \langle\rangle \\
& \forall e : \Sigma^{\checkmark};\ \rho : TTTrace \bullet fTock\, (\langle evt\ e \rangle \frown \rho) = fTock\, \rho \\
& \forall X : \mathbb{P}\, \Sigma^{\checkmark}_{tock};\ \rho : TTTrace \bullet fTock\, (\langle ref\ X, evt\ tock \rangle \frown \rho) = \langle ref\ X, evt\ tock \rangle \frown fTock\, \rho
\end{aligned}
$$

Timeout

$$
\begin{aligned}
& tt[\![P \triangle_d Q]\!] = \\
& \quad \{\rho_1 : tt[\![P]\!] \mid \#(\rho_1 \upharpoonright \{evt\ tock\}) < d\} \\
& \quad \cup \{\rho_1 : tt[\![P]\!];\ \rho_2 : tt[\![Q]\!];\ \phi : \text{seq } \Sigma^{\checkmark}_{tock} \mid \\
& \qquad \#(\rho_1 \upharpoonright \{evt\ tock\}) = d \land ((d = 0 \land \rho_1 = \langle\rangle) \lor (d > 0 \land \rho_1 = \phi \frown \langle evt\ tock \rangle)) \bullet \rho_1 \frown \rho_2 \\
& \quad \}
\end{aligned}
$$

Parallelism

$$tt[\![P \,[\![\, A \,]\!]\, Q]\!] = \bigcup\{\rho_1 : tt[\![P]\!];\ \rho_2 : tt[\![Q]\!] \bullet \rho_1 \,[\![\, A \,]\!]^T\, \rho_2\}$$

$$\_ \,[\![\, \_ \,]\!]^T\, \_ : (TTTrace \times \mathbb{P}\,\Sigma \times TTTrace) \to \mathbb{P}\, TTTrace$$

$$\begin{array}{l}
\forall X : \mathbb{P}\,\Sigma;\ Y, Z : \mathbb{P}\,\Sigma^{\checkmark}_{tock};\ e_1, e_2 : \Sigma;\ \rho_1, \rho_2 : TTTrace \bullet \\
\quad \langle\rangle \,[\![\, X \,]\!]^T\, \langle\rangle = \{\langle\rangle\} \land \\
\quad \langle\rangle \,[\![\, X \,]\!]^T\, \langle ref\ Y\rangle = \{\langle\rangle\} \land \\
\quad \langle\rangle \,[\![\, X \,]\!]^T\, \langle evt\ \checkmark\rangle = \{\langle\rangle\} \land \\
\quad e_1 \notin X \Rightarrow \langle\rangle \,[\![\, X \,]\!]^T\, (\langle evt\ e_1\rangle \frown \rho_1) = \{\rho_2 : \langle\rangle \,[\![\, X \,]\!]^T\, \rho_1 \bullet \langle evt\ e_1\rangle \frown \rho_2\} \land \\
\quad e_1 \in X \Rightarrow \langle\rangle \,[\![\, X \,]\!]^T\, (\langle evt\ e_1\rangle \frown \rho_1) = \{\} \land \\
\quad \langle\rangle \,[\![\, X \,]\!]^T\, (\langle ref\ Y, evt\ tock\rangle \frown \rho_1) = \{\} \land \\
\quad Y \setminus (X \cup \{\checkmark, tock\}) = Z \setminus (X \cup \{\checkmark, tock\}) \Rightarrow \langle ref\ Y\rangle \,[\![\, X \,]\!]^T\, \langle ref\ Z\rangle = \{\langle ref\ (Y \cup Z)\rangle\} \land \\
\quad Y \setminus (X \cup \{\checkmark, tock\}) \neq Z \setminus (X \cup \{\checkmark, tock\}) \Rightarrow \langle ref\ Y\rangle \,[\![\, X \,]\!]^T\, \langle ref\ Z\rangle = \{\} \land \\
\quad \langle ref\ Y\rangle \,[\![\, X \,]\!]^T\, \langle evt\ \checkmark\rangle = \{W : \Sigma^{\checkmark}_{tock} \mid W \subseteq X \bullet \langle ref\ (Y \cup W)\rangle\} \land \\
\quad e_1 \notin X \Rightarrow \langle ref\ Y\rangle \,[\![\, X \,]\!]^T\, (\langle evt\ e_1\rangle \frown \rho_1) = \{\rho_2 : \langle ref\ Y\rangle \,[\![\, X \,]\!]^T\, \rho_1 \bullet \langle evt\ e_1\rangle \frown \rho_2\}) \land \\
\quad e_1 \in X \Rightarrow \langle ref\ Y\rangle \,[\![\, X \,]\!]^T\, (\langle evt\ e_1\rangle \frown \rho_1) = \{\} \land \\
\quad \langle ref\ Y\rangle \,[\![\, X \,]\!]^T\, (\langle ref\ Z, evt\ tock\rangle \frown \rho_1) = \{\} \land \\
\quad \langle evt\ \checkmark\rangle \,[\![\, X \,]\!]^T\, \langle evt\ \checkmark\rangle = \{\langle evt\ \checkmark\rangle\} \land \\
\quad e_1 \notin X \Rightarrow \langle evt\ \checkmark\rangle \,[\![\, X \,]\!]^T\, (\langle evt\ e_1\rangle \frown \rho_1) = \{\rho_2 : \langle evt\ \checkmark\rangle \,[\![\, X \,]\!]^T\, \rho_1 \bullet \langle evt\ e_1\rangle \frown \rho_2\}) \land \\
\quad e_1 \in X \Rightarrow \langle evt\ \checkmark\rangle \,[\![\, X \,]\!]^T\, (\langle evt\ e_1\rangle \frown \rho_1) = \{\} \land \\
\quad \langle evt\ \checkmark\rangle \,[\![\, X \,]\!]^T\, (\langle ref\ Y, evt\ tock\rangle \frown \rho_1) = \\
\qquad \{Z : \mathbb{P}\,\Sigma^{\checkmark}_{tock};\ \rho_2 : TTTrace \mid \\
\qquad\quad \langle ref\ Z\rangle \in \langle evt\ \checkmark\rangle \,[\![\, X \,]\!]^T\, \langle ref\ Y\rangle \land \rho_2 \in \langle evt\ \checkmark\rangle \,[\![\, X \,]\!]^T\, \rho_1 \bullet \langle ref\ Z, evt\ tock\rangle \frown \rho_2 \\
\qquad\} \land \\
\quad e_1 \notin X \land e_2 \notin X \Rightarrow (\langle evt\ e_1\rangle \frown \rho_1) \,[\![\, X \,]\!]^T\, (\langle evt\ e_2\rangle \frown \rho_2) = \\
\qquad \{\rho_3 : \rho_1 \,[\![\, X \,]\!]^T\, (\langle evt\ e_2\rangle \frown \rho_2) \bullet \langle evt\ e_1\rangle \frown \rho_3\} \cup \{\rho_3 : (\langle evt\ e_1\rangle \frown \rho_1) \,[\![\, X \,]\!]^T\, \rho_2 \bullet \langle evt\ e_2\rangle \frown \rho_3\} \land \\
\quad e_1 \notin X \land e_2 \in X \Rightarrow (\langle evt\ e_1\rangle \frown \rho_1) \,[\![\, X \,]\!]^T\, (\langle evt\ e_2\rangle \frown \rho_2) = \\
\qquad \{\rho_3 : \rho_1 \,[\![\, X \,]\!]^T\, (\langle evt\ e_2\rangle \frown \rho_2) \bullet \langle evt\ e_1\rangle \frown \rho_3\} \land \\
\quad e_1 \in X \land e_2 \in X \land e_1 = e_2 \Rightarrow (\langle evt\ e_1\rangle \frown \rho_1) \,[\![\, X \,]\!]^T\, (\langle evt\ e_2\rangle \frown \rho_2) = \{\rho_3 : \rho_1 \,[\![\, X \,]\!]^T\, \rho_2 \bullet \langle evt\ e_1\rangle \frown \rho_3\} \land \\
\quad e_1 \in X \land e_2 \in X \land e_1 \neq e_2 \Rightarrow (\langle evt\ e_1\rangle \frown \rho_1) \,[\![\, X \,]\!]^T\, (\langle evt\ e_2\rangle \frown \rho_2) = \{\} \land \\
\quad e_1 \notin X \Rightarrow (\langle evt\ e_1\rangle \frown \rho_1) \,[\![\, X \,]\!]^T\, (\langle ref\ Z, evt\ tock\rangle \frown \rho_2) = \\
\qquad \{\rho_3 : \rho_1 \,[\![\, X \,]\!]^T\, (\langle ref\ Z, evt\ tock\rangle \frown \rho_2) \bullet \langle evt\ e_1\rangle \frown \rho_3\} \land \\
\quad e_1 \in X \Rightarrow (\langle evt\ e_1\rangle \frown \rho_1) \,[\![\, X \,]\!]^T\, (\langle ref\ Z, evt\ tock\rangle \frown \rho_2) = \{\} \land \\
\quad (\langle ref\ Y, evt\ tock\rangle \frown \rho_1) \,[\![\, X \,]\!]^T\, (\langle ref\ Z, evt\ tock\rangle \frown \rho_2) = \\
\qquad \{W : \mathbb{P}\,\Sigma^{\checkmark}_{tock};\ \rho_3 : TTTrace \mid \\
\qquad\quad \langle ref\ W\rangle \in \langle ref\ Y\rangle \,[\![\, X \,]\!]^T\, \langle ref\ Z\rangle \land \rho_3 \in \rho_1 \,[\![\, X \,]\!]^T\, \rho_2 \bullet \langle ref\ W, evt\ tock\rangle \frown \rho_3 \\
\qquad\} \land \\
\quad \rho_2 \,[\![\, X \,]\!]^T\, \rho_1 = \rho_1 \,[\![\, X \,]\!]^T\, \rho_2
\end{array}$$

Hiding

$$tt[\![P \setminus X]\!] = \bigcup\{\rho : tt[\![P]\!] \bullet hideTrace\, X\, \rho\}$$

$$hideTrace : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \to (TTTrace \to \mathbb{P}\, TTTrace)$$

$$\begin{aligned}
&\forall X, Y : \mathbb{P}\, \Sigma^{\checkmark}_{tock};\ e : \Sigma^{\checkmark}_{tock};\ \rho_1 : TTTrace \bullet \\
&\quad hideTrace\, X\, \langle\rangle = \{\langle\rangle\} \land \\
&\quad hideTrace\, X\, (\langle evt\ e\rangle \frown \rho_1) = \{\rho_2 : hideTrace\, X\, \rho_1 \mid e \in X\} \cup \{\rho_2 : hideTrace\, X\, \rho_1 \mid e \notin X \bullet \langle evt\ e\rangle \frown \rho_2\} \land \\
&\quad hideTrace\, X\, \langle ref\ Y\rangle = \{Z : \mathbb{P}\, Y \mid X \subseteq Y \bullet \langle ref\ Z\rangle\} \land \\
&\quad hideTrace\, X\, (\langle ref\ Y, evt\ tock\rangle \frown \rho_1) = \\
&\qquad \{\rho_2 : hideTrace\, X\, \rho_1 \mid tock \in X\} \\
&\qquad \cup \{Z : \mathbb{P}\, Y;\ \rho_2 : hideTrace\, X\, \rho_1 \mid tock \notin X \land X \subseteq Y \bullet \langle ref\ Z, evt\ tock\rangle \frown \rho_2\})
\end{aligned}$$

Renaming

$$tt[\![P[\![f]\!]]\!] = \bigcup\{\rho : tt[\![P]\!] \bullet renameTrace\, f\, \rho\}$$

$$renameTrace : (\Sigma^{\checkmark}_{tock} \to \Sigma^{\checkmark}_{tock}) \to (\mathrm{seq}\ Obs \to \mathbb{P}\, \mathrm{seq}\ Obs)$$

$$\begin{aligned}
&\forall f : \Sigma^{\checkmark}_{tock} \to \Sigma^{\checkmark}_{tock};\ e : \Sigma^{\checkmark}_{tock};\ X : \mathbb{P}\, \Sigma^{\checkmark}_{tock};\ \phi : \mathrm{seq}\ Obs \bullet \\
&\quad renameTrace\, f\, \langle\rangle = \{\langle\rangle\} \land \\
&\quad renameTrace\, f\, (\langle evt\ e\rangle \frown \phi) = \{t : renameTrace\, f\, \phi \bullet \langle evt\ (f\ e)\rangle \frown t\} \land \\
&\quad renameTrace\, f\, (\langle ref\ X\rangle \frown \phi) = \{t : renameTrace\, f\, \phi;\ Y : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \mid X = (f^{\sim})(\!|\ Y\ |\!) \bullet \langle ref\ Y\rangle \frown t\}
\end{aligned}$$

B SUMMARY OF ADDITIONAL SEMANTIC FUNCTIONS

$$iott^{O}[\![P]\!] \mathrel{\widehat{=}} \{\rho : TTTrace \mid addOuts^{O}(\rho) \in tt[\![P]\!]\}$$

$$iott^{O}_{M}[\![TT]\!] \mathrel{\widehat{=}} \{\rho : \mathrm{ran}\ addTick \mid addOuts(\rho) \in TT \bullet addOuts(\rho)\}$$

$$tstraces[\![P]\!] \mathrel{\widehat{=}} st(\!|\ iott^{O}[\![P]\!]\ |\!)$$

$$eiott^{O}[\![P]\!] \mathrel{\widehat{=}} \{\rho \in ETrace \mid \mathrm{et2iott}^{O}(\rho) \in iott[\![P]\!]\}$$