Implementation relations and testing for cyclic systems with refusals and discrete time*

Raluca Lefticaru

Department of Computer Science, University of Bradford, Bradford, West Yorkshire BD7 1DP UK

Robert M. Hierons

Department of Computer Science, The University of Sheffield, Sheffield, SD1 4DP, UK

Manuel Núñez

Design and Testing of Reliable Systems research group, Universidad Complutense de Madrid, Spain

Abstract

We present a formalism to represent cyclic models and study different semantic frameworks that support testing. These models combine sequences of observable actions and the passing of (discrete) time and can be used to specify a number of classes of reactive systems, an example being robotic systems. We use implementation relations in order to formally define a notion of correctness of a system under test (SUT) with respect to a specification. As usual, the aim is to devise an extension of the classical ioco implementation relation but available timed variants of ioco are not suitable for cyclic models. This paper thus defines new implementation relations that encapsulate the discrete nature of time and take into account not only the actions that models can perform but also the ones that they can refuse. In addition to defining these relations, we study a number of their properties and provide alternative characterisations, showing that the relations are appropriate conservative extensions of trace containment. Finally, we give test derivation algorithms and prove that they are sound and also are complete in the limit.

Keywords: Model-based testing; Implementation relations; Cyclic systems.

1. Introduction

Cyclic systems [6] operate in cycles of the following form: *read* values from sensors (inputs), perform *calculations*, then *write* values to actuators (outputs). The corresponding models typically combine actions into a single step and a behaviour is then a sequence of steps, with time passing between steps. The importance of these models lies in their relevance to a range of systems (e.g. embedded control systems) including those used in robotics. This form of models is found with languages, such as Statecharts [16], that have a step semantics.

The aim of the work, we are going to discuss, is to define formal, automated, approaches to testing that fit into the standard robotics development process. Typically, a robotic system will be modelled as a state machine. Afterwards, a simulation model is produced, which is used to validate the original model. The simulation model can also be used to drive the testing of the developed system; a point of particular relevance to this paper. In contrast to the analysis of many systems where time can be (at least partially) abstracted, time plays a fundamental role in cyclic systems and must be treated as a *first-class citizen*. In the simulations, time is discrete and each time slot contains a sequence of actions. The use of robotic systems is expanding in areas such as manufacturing, healthcare, transport and home assistance. For example, the sales of these systems increased by 30% in 2017 and this represented a new record for the fifth year in a row [25]. This trend is steady and, according to a UK government report[1], the value of the global market for robotics and autonomous systems is expected to reach £13 billion by 2025.

The work described in this paper came out of the early stages of a project in the area of testing robotic systems. The motivation was to provide a foundational testing theory on which to base sound automated testing. The formalism used is the syntactically simplest one that captures the features required; syntactic simplicity is beneficial since it facilitates formal reasoning. The formalism also has the advantage that it corresponds to the form of operational semantics given to the models from which we wish to test [2].

*This work has been supported by EPSRC grant EP/R025134/2 RoboTest: Systematic Model-Based Testing and Simulation of Mobile Autonomous Robots, the Spanish MINECO-FEDER grant FAME (RTI2018-093608-B-C31) and the Region of Madrid grant FORTE-CM (S2018/TCS-4314) co-funded by EIE Funds of the European Union.

Email addresses: r.lefticaru@bradford.ac.uk (Raluca Lefticaru), r.hierons@sheffield.ac.uk (Robert M. Hierons), mn@sip.ucm.es (Manuel Núñez)

[1] https://tinyurl.com/nyf64av

[2] The models are written in a language, RoboChart [35] but, since we are testing within a simulation, we actually test from RoboSim models [9] derived from RoboChart.

The overall line of work was motivated by several factors. First, robotic systems are ubiquitous in safety-critical areas and it is of the upmost importance to increase the confidence in the correctness of these systems. It is becoming widely recognised, not only in academia but also in industry, that the use of formal methods increases the reliability of the developed systems [4, 13, 37]. As a result, since software testing [2, 36] is the main software validation technique, the combination of formal methods and testing is a promising approach to analyse the behaviour of systems; this is the second factor that motivated the work. Note that although testing was initially considered to be a mainly manual and *informal* activity, many formal approaches to testing have been defined and applied [3, 11, 20, 21].

Formal approaches to testing usually rely on state-based models, that is, models that are in the form of labelled transition systems (LTSs); these models have states and labelled transitions between the states and there are many tools to support testing [30, 44]. In testing from an LTS, it is normal to assume that the system under test (SUT) behaves like an unknown LTS (this is called the *minimal test hypothesis* [26]). So, testing involves comparing two LTSs to decide whether the first (unknown) one is correct with respect to the second one. The relation between LTSs, that describes what we mean by correctness, is usually formalised as an *implementation relation*. The literature provides a myriad of implementation relations for LTSs [14, 15], where the difference between them strongly depends on the observational power of the observer.

In this paper we introduce a framework to specify and validate robotic systems described as cyclic models. The ultimate goal is to provide techniques and tools that can be used to make the development of robotic systems more efficient and effective, through removing the need for several manual, error prone activities. Next we enumerate the main contributions. First, we introduce a state-based formalism, tockLTS, to specify cyclic models. The tockLTS formalism corresponds to the one underlying the operational semantics of tock-CSP [42, Chapter 14], a timed variant of CSP. This process algebra is used to give semantics to RoboSim [9], a language used to define simulation models in robotics. Note that we consider tockLTS, instead of RoboSim, because it has a simpler syntax but also is more general. In fact, we expect that it will be relatively easy to adapt our framework to other simulation languages and to languages having a cyclic nature and a step semantics (e.g. Statecharts [16]). Second, we define several implementation relations (notions of correctness). Having different notions of correctness is useful because the users of our framework can decide which one better suites their needs. There are several types of observations that can be made in the framework. As usual, we assume that it is possible to observe the actions that a system performs and also the passing of time. In addition, we also consider a notion of *refusal*; the observation that the system cannot perform certain actions. Our previous work shows that the use of refusals makes it possible to distinguish between distinct behaviours that otherwise would be identical. We study properties of these implementation relations, in particular, we show how they are related via containment and present alternative characterisations. Having defined implementation relations, the final contribution consists of a testing framework for each implementation relation. We provide such a framework and corresponding test derivation algorithms. These algorithms are *sound*: if the SUT does not pass one of the test cases produced by the algorithm then we know that the SUT is faulty. These algorithms are also exhaustive *in the limit*; if the SUT is faulty then it fails at least one test case that can be returned by the test generation algorithm. Observe that this means that finite test suites do not provide guaranteed fault detection effectiveness. However, this limitation is inevitable unless one introduces, for example, a finite fault model that restricts the class of faults that might occur. The approach is also common practice in ioco-based implementation relations [45], like the ones that we introduce in this paper; we discuss this further in Section 8 of the paper.

The choice of formalism was motivated by several factors. First, as previously noted, the work was developed within a robotics project and the aim was to use a formalism that would be familiar to roboticists but that could also be provided with a formal semantics (to support sound test generation and also formal verification). This led to the development of the RoboChart language, which is based on UML Statecharts; the view of colleagues working in robotics was that roboticists would not use languages, such as timed automata [1], that are less familiar. RoboChart models are mapped to RoboSim, in order to run simulations. Since we were initially interested in testing within a simulation, testing should be from RoboSim. However, it makes little sense to have a testing theory based on a language such as RoboSim, which has a rich syntax (so the theory would be complex and difficult to reason about). As a result, we based the testing theory on the syntactically simplest formalism that could capture the semantics of a RoboSim model. Note that, in addition, RoboSim models can be mapped to CSP specifications and the formalism used in this paper is inspired by CSP with discrete time (tock-CSP [42, Chapter 14]). The formalism essentially mirrors the operational semantics of tock-CSP but with one crucial difference: since we are interested in testing, it is necessary to have a semantics that reflects the different roles of inputs and outputs. As a result of the above, the testing theory developed can be used to reason about testing from RoboSim models and also as the basis for deriving suitable test cases from such models[3]. The formalism used in this paper is also quite general, the hope being that it will be possible to use the proposed formalism to express the semantics of other models in languages similar to RoboChart and RoboSim and that the testing theory developed can then be applied.

Interestingly, the cyclic nature of the models makes them a little like Finite State Machines (FSMs), also called Mealy Machines [31], in which there is a finite set of states, transitions between the states, and each transition has an input/output pair. There is a rich theory, and set of test generation techniques, associated with FSMs (see, for example, [19, 28, 38]). As a result, an alternative might have been to use FSMs. Unfortunately, FSMs do not provide a general solution. The first point is that a transition would not necessarily involve an input/output pair;

[3] In practice, test generation is likely to involve mapping RoboSim to tock-CSP

it could have an arbitrarily long sequence of inputs and outputs. The first value in a sequence would be an input but after that the sequence could have outputs before inputs; the controllability of such sequences would be very different to that of transitions in Mealy Machines. In addition, the 'transition structure' taking one from state s to state s' could contain cycles and so there may be infinitely many transitions between two states (as well as there potentially being infinitely many states). However, there may well be situations in which FSMs can be used and the corresponding theory and test generation algorithms would then have a number of benefits, such as there being algorithms that generate tests that are guaranteed to find all faults in a given fault model.

This paper represents a revised, extended and enhanced version of our recent work [29]. Next we briefly describe the main contributions with respect to the conference paper. First, we provide a more detailed and rigorous development of the two implementation relations previously given; one based on traces, the second with refusals included. In addition, for each implementation relation we consider the situation in which an input is not defined in a given state of the specification and alternatives that capture different context (ways in which incompleteness is handled). This leads to an additional two variants of each implementation relation previously given [29] and we prove how these relate. We also define two testing frameworks, one for each implementation relation. We give associated automated test generation algorithms, proving that these are sound and also complete in the limit.

The rest of the paper is organised as follows. In Section 2 we briefly discuss the context in which our framework fits and the most relevant related work. In Section 3 we introduce the main definitions and concepts used. Based on these, in Section 4 we provide three implementation relations based around trace inclusion. Section 5 defines the most general type of observation (the notion of timed refusal traces). In Section 6 we introduce implementation relations based on timed refusal traces and show that they are strictly stronger than the corresponding relations from Section 4. In Section 7 we present alternative characterisations of the implementation relations from Section 6. Section 8 then explores test generation and finally, Section 9 provides conclusions and describe some lines for future work.

2. Technical context and related work

In order to develop robotic systems, it is usual to start with a state-based model. The work described in this paper was developed in the context of the state-based RoboChart language [35], which allows models to be described in a way that is familiar to roboticists. A RoboChart model can be automatically mapped to a simulation model, in a language called RoboSim [9], that is consistent with the original model. Simulation models are also given a formal semantics, making it possible to automatically analyse or reason about them. A formal semantics for RoboSim [10] is given by mapping a RoboSim model to a variant of CSP, called tock-CSP [42, Chapter 14]. The operational semantics of this language is, as usual, given in terms of an LTS. One of the benefits of this approach is that we can analyse the semantics of a RoboSim model using formal tools and methodologies available for CSP. As previously explained, in this paper we work with a formalism that is similar to these LTSs (it is actually a slight generalisation). Therefore, our framework can be directly applied to the development of robotic systems.

In testing we distinguish between inputs and outputs since these play different roles and this has led to implementation relations that were not classically considered [14, 15]. The best known implementation relation is ioco [45]. There are three main differences between the classical definition of ioco and the implementation relations given in this paper. We present relations for timed systems where time is discrete, we are interested in using refusals while testing, and we do not assume that SUTs are *input-enabled*.

Concerning the last property, we do not assume that the SUT is *input-enabled*, that is, we do not impose the restriction that the SUT should be able to react to any input provided by the tester. This assumption (the SUT being input-enabled) makes sense for a range of systems and is based on the observation that the SUT will not block input. However, there are also systems that are not input-enabled and where this is deliberate. For example, certain options/fields might be greyed-out on a webpage or simply not available; consider, for instance, the options available to an editor and to an author in a journal's manuscript system. In the context of autonomous systems, a system might switch off sensors and, in addition, sensors might fail. It is well-known that one can convert a model that is not input-enabled into one that is. However, in the type of systems that we consider in this paper, such a completed model would less appropriately model an SUT in which certain inputs are disabled and could lead to the generation of test cases that either do not make sense from a testing perspective or introduces redundancy. We consider an approach in which the absence of an input denotes that input is not allowed/possible but we also define more relaxed implementation relations that adapt the ioco approach to undefined inputs.

There are a number of options regarding the observational power of a user interacting with a system. An observer might only be allowed to observe the actions in which the system participates. However, we can increase the capabilities of observers. For example, we might consider situations in which it is possible to also observe the *refusal* of a set of actions, that is, those actions in which the system cannot participate at a certain point. In the scope of testing and process algebras, this is a well-known approach [40], with some subtle differences with respect to the notion of *must testing* [12]. In classical ioco, there is only one type of refusal, called *quiescence*, and this can be observed if the system is in a state where it cannot evolve via an internal action and, in addition, the system cannot produce an output without first receiving an input. The observation of the refusal of a set A is typically represented by the situation in which the environment chooses to only engage in the actions in A and the composition of the environment and the SUT deadlocks. A deadlock is usually observed through a timeout, similar to the process of observing quiescence in classical ioco, and this takes time. As a result, the observation of a deadlock

(and so also a refusal) should precede a *duration* (an action representing a unit of time passing).

We are not aware of alternative approaches that consider models with the required features. Of particular interest is the combination of urgent outputs, refusals and discrete time. It is exactly this combination that we have to reason about for the models in which we are interested. There is a need for a formal testing theory for such models, in order to support the development of sound test generation algorithms and, ultimately, test generation and execution tools.

It is important to note that there is a variant of ioco including refusals and where systems need not be input-enabled [17]. We depart from this work in several lines (in addition to including time). First, our refusals are observed only in stable states[4] and this has some implications. Specifically, an *internal choice* between outputs is equivalent to the same external choice while if we consider inputs then we obtain semantically different processes. Using a process algebraic notation, we have $(\tau; !o_1; stop) + (\tau; !o_2; stop) \sim (!o_1; stop) + (!o_2; stop)$ while $(\tau; ?i_1; stop) + (\tau; ?i_2; stop) \nsim (?i_1; stop) + (?i_2; stop)$, where actions preceded by ? and ! denote, respectively, an input and an output, and τ denotes an internal action. Second, their notion of a process being input-enabled is more restrictive than ours: at a certain port, either all the inputs are enabled or none of them is. In their notation, we have only one port and we allow several inputs to be enabled and several to not be.

Finally, ioco does not take into account time. Note that time cannot be taken as an *ordinary* action because it is neither an input, since the tester does not control it, nor an output, since the SUT does not control it (the SUT cannot, for example, stop time). There are several timed variants of ioco (all are typically called tioco) [7, 27, 43]. The versions of tioco differ in a number of ways, including whether quiescence is a possible observation. It might be possible to use a version of tioco with discrete time, and in some situations this will be sufficient, but we prefer a *native* discrete time tioco (in addition, previous work does not consider a general notion of refusal).

From the above, one can see that a number of formalisms have been developed within the context of formal testing and these have corresponding implementation relations. However, it appears that they do not provide the combination of features required for the models in which we are interested. As a result, this paper develops a novel testing theory, with associated implementation relations, that builds on previous work on ioco and tioco.

3. Background and models

In this section we define the models and notation used in this paper along with some properties that we expect our systems to fulfill.

[4] We will say that a state is stable if it is not possible to take a transition whose label is an output or an internal action.

3.1. Cyclic models

As previously explained, the work in this paper is motivated by the nature of the types of models used with embedded control systems in areas such as robotics and the automotive industry. Such control systems operate in cycles of the following form.

1. Read values from sensors (inputs).
2. Perform calculations.
3. Write values to actuators (outputs).

The corresponding models typically combine actions into a single step and a behaviour is then a sequence of steps, with time passing between steps. Such models can be found in robotics, with RoboCalc, RoboSim and models used in simulation packages [41] but also in a number of variations of Statecharts [16].

Example 1. *This example is inspired from previous work [8], which presented a simple rescue application that uses a drone to deliver some relief to a given target location. The complete model, described using the RoboChart language [35], is more complex, comprising other elements like modules, robotic platforms or controllers. In Figure 1 we present only an excerpt: the state machine model. It has four self-explanatory states:* `Off`, `Looking`, `Delivering`, `Returning`. *The RoboChart notation allows us to specify* `entry` *and* `during` *states,* `exit` *actions, transition actions, using events and operations. For example, the transitions in our model can be triggered by different events (e.g.* `switchOn` *or* `found` *when the target is found) and operations such as* `wait(TOP)` *(an action that pauses the system during* `TOP` *time units) or move*(*LV*).

In the rest of this section we explain the formalisation we used. This is consistent with the above type of model but is rather more general than, for example, only considering models used within a particular domain. The aim is to develop formalisations and corresponding techniques that are relatively general and that can be used in the testing of embedded systems. Interestingly, it will transpire that a number of the classical testing assumptions do not hold and so implementation relations such as the different available versions of tioco are not suitable.

3.2. Traces and automata

Observations made in testing will be in the form of sequences and we use ϵ for the empty sequence. Given set A, A^* denotes the set of finite sequences of elements from A and A^ω denotes the set of infinite sequences of elements from A.

A system will interact with its environment through inputs and outputs. Throughout the paper, I and O will represent the (disjoint) input and output alphabets and we let $L = I \cup O$ denote the set of actions.

The basic, untimed, type of model we consider is an automaton in which, as usual in Automata Theory and in contrast to the standard notion of LTS, we have the concept of a final state.

Definition 1 (Automaton). *An* automaton *is a tuple* $p = (Q, q_0, L, T, F)$ *where*

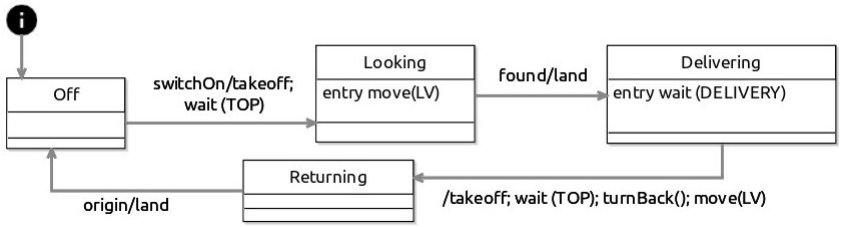


Figure 1: State machine model of a rescue drone [8]

- *Q is a countable, non-empty set of states;*
- *$q_0 \in Q$ is the initial state;*
- *L is a countable set of visible actions;*
- *$T \subseteq Q \times (L \cup \{\tau\}) \times Q$ is the transition relation, where $\tau \notin L$ represents an internal action;*
- *$F \subseteq Q$ is the set of final states.*

At any time, an automaton p is in a particular state $q \in Q$. If $(q, a, q') \in T$ for action $a \in L \cup \{\tau\}$ then p can move to state q' through a. We will sometimes use an alternative notation: a transition $(q, a, q') \in T$ can be expressed as $q \xrightarrow{a} q'$. We will also write $q \overset{a}{\not\rightarrow}$ if there does not exist q' such that $(q, a, q') \in T$. The transition relation can be extended as follows.

Definition 2. *Let $p = (Q, q_0, L, T, F)$ be an automaton with states $q, q' \in Q$, $P \subseteq Q$ be a set of states, visible actions $a, a_1, \ldots, a_n \in L$, with $n > 1$, and sequence of visible actions $\sigma \in L^*$.*

$$q \overset{\epsilon}{\Longrightarrow} q' \Leftrightarrow_{\text{def}} q = q' \text{ or } \exists q_1, \ldots, q_{n-1} \in Q : q \xrightarrow{\tau} q_1 \xrightarrow{\tau} \ldots q_{n-1} \xrightarrow{\tau} q'$$

$$q \overset{a}{\Longrightarrow} q' \Leftrightarrow_{\text{def}} \exists q_1, q_2 \in Q : q \overset{\epsilon}{\Longrightarrow} q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Longrightarrow} q'$$

$$q \overset{a_1 \ldots a_n}{\Longrightarrow} q' \Leftrightarrow_{\text{def}} \exists q_1, \ldots, q_{n-1} \in Q : q \overset{a_1}{\Longrightarrow} q_1 \overset{a_2}{\Longrightarrow} \ldots q_{n-1} \overset{a_n}{\Longrightarrow} q'$$

$$q \overset{\sigma}{\Longrightarrow} \Leftrightarrow_{\text{def}} \exists q' \in Q : q \overset{\sigma}{\Longrightarrow} q'$$

$$P \overset{\sigma}{\Longrightarrow} \Leftrightarrow_{\text{def}} \exists r \in P : r \overset{\sigma}{\Longrightarrow}$$

$$q \overset{\sigma}{\not\Longrightarrow} \Leftrightarrow_{\text{def}} \nexists q' \in Q : q \overset{\sigma}{\Longrightarrow} q'$$

$$P \overset{\sigma}{\not\Longrightarrow} \Leftrightarrow_{\text{def}} \forall r \in P : r \overset{\sigma}{\not\Longrightarrow}$$

$$p \overset{\sigma}{\Longrightarrow} \Leftrightarrow_{\text{def}} q_o \overset{\sigma}{\Longrightarrow}$$

As usual, we will not always distinguish between a model and its initial state. If $p = (Q, q_0, L, T, F)$, then we will identify p with its initial state q_0, and, for example, we will usually write $p \overset{\sigma}{\Longrightarrow}$ instead of $q_0 \overset{\sigma}{\Longrightarrow}$. The automaton $p = (Q, q_0, L, T, F)$ defines the language $L(p)$ of finite sequences that take p to a final state.

Definition 3. *Given automaton $p = (Q, q_0, L, T, F)$, the language $L(p) \subseteq L^*$ is defined as*

$$L(p) = \{\sigma \in L^* | \exists q \in F : q_0 \overset{\sigma}{\Longrightarrow} q\}$$

3.3. Timed models

We now describe our timed model, which is an LTS in which there is a special action, $\ominus$, that denotes the passing of a unit of time. We call this action 'tock' in order to be consistent with tock-CSP [42, Chapter 14].

Definition 4 (tockLTS, timed traces). *A* labelled transition system with tock (or tockLTS) *is a tuple $p = (Q, q_0, I, O, T)$ where*

- *Q is a countable, non-empty set of states;*
- *$q_0 \in Q$ is the initial state;*
- *I and O are countable disjoint sets of inputs and outputs respectively, with $L = I \cup O$ being the set of visible actions;*
- *$T \subseteq Q \times (L \cup \{\tau, \ominus\}) \times Q$ is the transition relation, where $\tau \notin L$ represents the internal action, and $\ominus$ represents a* tock *action denoting the passage of a unit of time.*

We use TockLTS(I, O) to denote the set of tockLTS with input set I and output set O.

The definition of the $\overset{\sigma}{\Longrightarrow}$ relation is similar to the one given in Definition 2, with the only difference that $\sigma \in (L \cup \{\ominus\})^*$ and, therefore, we omit it. The set of timed traces *of p is defined as*

$$\mathcal{T}\texttt{traces}(p) = \{\sigma \in (L \cup \{\ominus\})^* \mid p \overset{\sigma}{\Longrightarrow}\}$$

We will require some additional notation. Specifically, we will compute the states that can be reached from a given state after performing a sequence of actions and define a predicate to decide whether a system can refuse a set of actions at a certain state.

Definition 5. *Let $q = (Q, q_0, I, O, T)$ be a tockLTS, $p \in Q$ be a state, $P \subseteq Q$ be a set of states, $A \subseteq L$ be a set of labels, and $\sigma \in (I \cup O \cup \{\ominus\})^*$. We define the following notions:*

1. *p* **after** *$\sigma =_{\text{def}} \{p' \in Q | \, p \overset{\sigma}{\Longrightarrow} p'\}$*
2. *P* **after** *$\sigma =_{\text{def}} \bigcup\{ p$* **after** *$\sigma | p \in P\}$.*
3. *P* **refuses** *$A =_{\text{def}} \exists p \in P, \forall \mu \in A \cup \{\tau\} : p \overset{\mu}{\not\rightarrow}$.*

As usual, we expect processes to have certain properties. First, we require that time can progress: there cannot be a state from which it is impossible for time to advance. Second, we should have the *urgency* of internal actions and outputs (to be consistent with how our cyclic models operate). Third,

processes should not show *Zeno behaviour*, that is, a process should not be able to follow an infinite sequence of actions in finite time. Finally, processes should have *time determinism*: processes do not branch as a result of time passing (performing a $\ominus$), though a process can branch through internal actions that occur after a $\ominus$.

Definition 6 (Urgency, Zeno behaviour, time determinism). *Let $p = (Q, q_0, I, O, T)$ be a tockLTS. Then*

- *p is time progressing if for all $q \in Q$ there exists $\sigma \in (I \cup O)^*$ such that $q \xrightarrow{\sigma\ominus}$.*
- *p has urgent internal actions and output if for all $q \in Q$ and $a \in O \cup \{\tau\}$, if $q \xrightarrow{a}$ then $q \not\xrightarrow{\ominus}$.*
- *p has Zeno behaviour if there exists a state $q \in Q$ and an infinite path from q with finitely many* tock *actions.*
- *p has time determinism if for all states $q_1, q_2, q_3 \in Q$ we have that $q_1 \xrightarrow{\ominus} q_2 \wedge q_1 \xrightarrow{\ominus} q_3$ implies $q_2 = q_3$.*

Note that previously [29] we required that processes do not have *forced inputs*, that is, for each state of a process there exists at least one outgoing transition that is not an input. Conceptually, this requirement makes sense - it essentially says that processes cannot stop time. We weakened this assumption, to requiring that time progresses, because some cyclic models can have forced inputs; in development, there is then the obligation to demonstrate that the context in which the SUT is used ensures that the composition of the specification and its environment cannot stop time.

Example 2. *In Figure 2 we give a formal representation, using a tockLTS, of the RoboChart state machine presented in Example 1. Let $p_{drone} = (Q, q_0, I, O, T)$ denote this tockLTS, with this having state set $Q = \{O_1, O_2, O_3, L_0, L_1, L_2, D_0, D_1, D_2, D_3, D_4, R_1, R_2\}$, initial state $q_0 = O_1$ and action sets $I = \{?f, ?o, ?s\}$ and $O = \{!b, !l, !m, !t\}$.*

Essentially, the events that trigger transitions in RoboChart are transformed into input symbols ($?s$ stands for `switchOn`, $?f$ stands for `found` and $?o$ stands for `origin`) while actions or sequences of action calls are transformed into output symbols ($!t$ stands for `takeoff`, $!l$ stands for `land`, $!b$ stands for `turnBack` and $!m$ stands for `move`). We have transformed the actions involving time, such as `wait(TOP)` and `wait(DELIVERY)`, where `TOP` and `DELIVERY` are greater than 0, into sequences of actions $(\ominus \cdot \tau)^* \cdot \ominus \cdot !\alpha$ that simulate the passing of an arbitrary number of time units followed by an output action $!\alpha \in O$. We have included additional states in order to capture the more complex transition behaviours. Specifically, we have split the RoboChart transition $START \xrightarrow{ev/a_1;\ ...;\ a_n} END$ into the sequence of tockLTS transitions $S_1 \xrightarrow{?e} S_2 \xrightarrow{!a_1} S_3 \cdots S_{n+1} \xrightarrow{!a_n} E_1$. Similarly, other transitions were added to simulate `entry` actions associated with states. The state `Off` is transformed into the states O_1, O_2 and O_3, the state `Looking` is transformed into the states L_0, L_1 and L_2, and the states `Delivering` and `Returning` are transformed, respectively, into the states D_1, D_2, D_3 and D_4 and R_1 and R_2.

*We now illustrate some of the concepts from Definition 5 on p_{drone}. For example, considering $\sigma = !t \cdot \ominus$ the following sets can be computed: O_2 **after** $\sigma = \{L_0\}$, D_1 **after** $\sigma = \{D_3\}$ and, consequently, $\{D_1, O_2\}$ **after** $\sigma = \{D_3, L_0\}$. Concerning refusals, for example, we have that D_0 **refuses** $(I \cup O)$.*

Regarding the (timed) traces of a model, it is worth mentioning that they can be regarded as the visible (and tock) actions of a path. For example, p_{drone} has a path determined by the execution of the action sequence $?s \cdot !t \cdot \ominus \cdot \tau \cdot \ominus \cdot \tau \cdot \ominus \cdot !m$. However, the corresponding timed trace will consist only of visible and tock actions: $?s \cdot !t \cdot \ominus \cdot \ominus \cdot \ominus \cdot !m \in \mathcal{T}\mathtt{traces}(p_{drone})$.

It is easy to check that p_{drone} satisfies the properties from Definition 6: the model has time determinism and it has urgent internal and output actions. If we were to add, for example, a $\ominus$ transition departing from L_0 then the urgency property would no longer be true. Similarly, an additional $\ominus$ transition in O_3 would violate time determinism. Note that time is always progressing in p_{drone} as from any state it is possible to reach a state from which there is a $\ominus$ transition. It is easy to check that p_{drone} does not have Zeno behaviour. However, if the transition $O_3 \xrightarrow{\ominus} L_0$ was replaced by the transition $O_3 \xrightarrow{\mu} L_0$, with $\mu \in I \cup U \cup \{\tau\}$, then the new tockLTS would have an infinite path described by $(\mu\tau)^\omega$ with no $\ominus$ actions. It would therefore exhibit Zeno behaviour.

The tockLTS we consider have the following property; the proof easily follows from the absence of Zeno behaviour and the time progressing assumption.

Proposition 1. *Let $p = (Q, q_0, I, O, T)$ be a tockLTS. We have that for all $q \in Q$ there exists an infinite path $\sigma = \mu_1 \ominus \mu_2 \ominus \mu_3 \ldots \in ((I \cup O)^*\{\ominus\})^\omega$ such that $q \overset{\sigma}{\Longrightarrow}$ and $\forall i, \mu_i \in (I \cup O)^*$.*

4. Implementation relations based on traces

If the environment can only observe traces of visible actions and time (i.e. it cannot observe refusals) then we have a number of associated implementation relations. We start with the simplest of these, which simply requires every behaviour (trace) of the SUT to also be a trace of the specification.

Definition 7. *Let p and q be two tockLTSs. We say that p conforms to q under* timed trace inclusion *if and only if $\mathcal{T}\mathtt{traces}(p) \subseteq \mathcal{T}\mathtt{traces}(q)$. We denote this $p \preceq_{tr} q$.*

The following property is immediate from the definition.

Proposition 2. *The timed trace inclusion relation is reflexive and transitive but need not be symmetric or antisymmetric.*

Consider now the situation in which σ is a trace of the specification and there is an input $?a$ such that $\sigma?a$ is not a trace of the specification. If σ is also a trace of the SUT then trace inclusion requires that the SUT cannot receive input $?a$ after σ; $?a$ is not defined after σ. This makes sense in contexts in

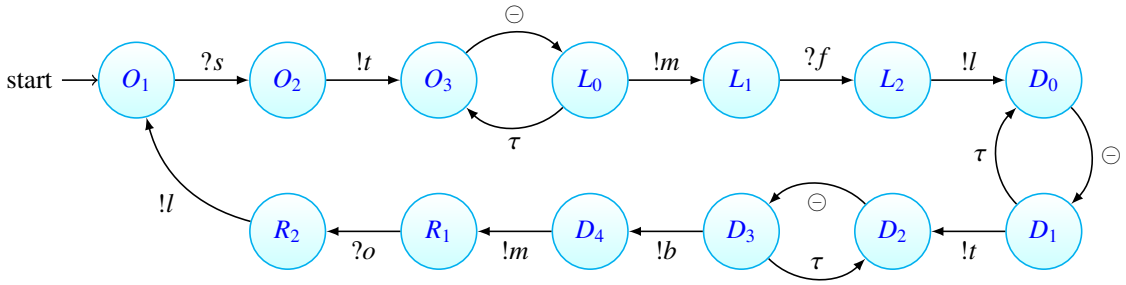


Figure 2: TockLTS drone model

which an input being undefined in the specification corresponds to that input not being allowed. For example, an input being undefined might correspond to the requirement that the corresponding field or option either does not appear in an interface or is greyed-out or that a system does not have access to a particular sensor (it is turned off). However, there are also cases where an input not being specified denotes the situation in which any behaviour is allowed ('do not care') and for such situations it is clear that timed trace inclusion is not the right implementation relation. We now define two additional implementation relations that interpret an input being undefined in slightly different ways but in ways that are consistent with the (untimed) implementation relations *ioco* and *uioco* [45].

One approach is to say that if a trace σ_1 can take the specification to a state q in which $?a$ is not defined then any behaviour is allowed if the SUT receives $?a$ after σ_1. The idea here is simply that the specification might have been in a state q in which $?a$ is not defined and so we should allow any behaviour after this.

Definition 8. *Let p and q be two tockLTSs. We say that $p \preceq^u_{tr} q$ if and only if for all $\sigma \in \mathcal{T}\mathtt{traces}(p)$ either $\sigma \in \mathcal{T}\mathtt{traces}(q)$ or there exists a prefix $\sigma_1?a$ of σ, with $?a \in I$, such that $\sigma_1 \in \mathcal{T}\mathtt{traces}(q)$ and there is a state q_1 such that $q \stackrel{\sigma_1}{\Longrightarrow} q_1$ and $q_1 \stackrel{?a}{\not\Rightarrow}$.*

The following property is immediate from the definition.

Proposition 3. *The relation $\preceq^u_{tr}$ is reflexive but need not be symmetric or antisymmetric.*

It is interesting to note that, as the following example shows, this relation need not be transitive. This lack of transitivity is essentially a result of our not requiring that systems are input-enabled.

Example 3. *Consider the tockLTSs p (left), q (centre) and r (right) depicted in Figure 3. We have that $p \preceq^u_{tr} q$ since the specification (that is, q) does not say anything about what the SUT (that is, p) should do concerning $?i$ and the subsequent evolutions. It is trivial that $q \preceq^u_{tr} r$ because, in particular, we have $q \preceq_{tr} r$. However, it is obvious that $p \preceq^u_{tr} r$ does not hold.*

The $\preceq^u_{tr}$ implementation relation can be seen as allowing any response to an input $?a$ in a state of the specification in which $?a$ is not defined. As a result, we can see this in terms of completing the specification in the following way.

Definition 9. *Let $p = (Q, q_0, I, O, T)$ be a tockLTS. The completion of p, denoted $C(p)$, is the tockLTS $(Q \cup \{q_c\}, q_0, I, O, T')$ in which $q_c \notin Q$ is a fresh state and $T' = T \cup T_1 \cup T_2$ in which:*

1. $T_1 = \{(q, ?a, q_c) | q \in Q \wedge ?a \in I \wedge q \stackrel{?a}{\not\Rightarrow}\}$
2. $T_2 = \{(q_c, a, q_c) | a \in I \cup O\}$.

Note that $C(p)$ presents a Zeno behaviour: once we reach the fresh state q_c, we cannot perform any $\ominus$. This is not a problem because we only use this process as a theoretical tool to provide an alternative characterisation of the $\preceq^u_{tr}$ relation.

Example 4. *Consider the tockLTS drone model given in Figure 2 and described in Example 2. We build its completion $C(p_{drone})$ by adding a fresh state q_c and the transition sets T_1, T_2 as explained in Definition 9. Because p_{drone} has many states that do not accept any input, T_1 will include transitions from each of these states, $\{O_2, O_3, L_0, L_1, D_0 - D_4, R_2\}$, to the new state, q_c, labelled with all the input actions. For the remaining states, $\{O_1, L_1, R_1\}$, the transition set T_1 will include only transitions with inputs which are not already accepted by each of these states. For example, for O_1 it includes only $O_1 \stackrel{?f}{\longrightarrow} q_c, O_1 \stackrel{?o}{\longrightarrow} q_c$. T_2 consists of self-loops labelled with all the visible actions.*

Then we have the following result. The proof follows easily from the fact that $\preceq^u_{tr}$ extends $\preceq_{tr}$ by *accepting* undefined behaviours. This is exactly the role of $C(q)$ with respect to q: extend the latter with all potential behaviours after unspecified inputs.

Theorem 1. *Given tockLTS p and q with the same alphabets, $p \preceq^u_{tr} q$ if and only if $p \preceq_{tr} C(q)$.*

The above deals with undefined inputs in a manner similar to the (untimed) implementation relation *uioco* for (untimed) input-output transition systems and this is also similar to how undefined inputs are considered in the literature on testing from finite state machines (see, for example, [18, 19, 38, 39]). However, the implementation relation *ioco* takes a different perspective, which in effect says that the response to input $?a$ after trace σ_1 is defined in a specification q if there is some state q_1 such that q_1 can be reached from the initial state of q by σ_1 ($q \stackrel{\sigma_1}{\Longrightarrow} q_1$) and $?a$ is defined in q_1 ($q_1 \stackrel{?a}{\Longrightarrow}$). The following adapts timed trace inclusion by taking the *ioco* approach to inputs not being specified.

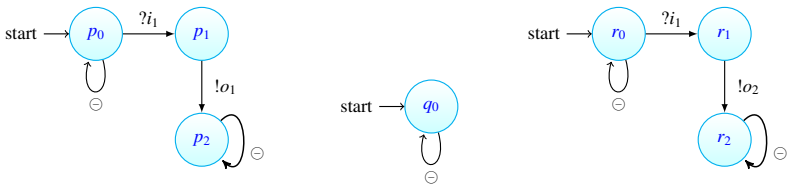


Figure 3: Models (un)related by variants of trace inclusion

Definition 10. *Let p and q be two tockLTSs. We say that $p \preceq^{i}_{tr} q$ if and only if for all $\sigma \in \mathcal{T}\mathtt{traces}(p)$ either $\sigma \in \mathcal{T}\mathtt{traces}(q)$ or there exists a prefix $\sigma_1 ?a$ of σ, with $?a \in I$, such that $\sigma_1 \in \mathcal{T}\mathtt{traces}(q)$ and for all q_1 such that $q \stackrel{\sigma_1}{\Longrightarrow} q_1$ we have that $q_1 \stackrel{?a}{\not\Rightarrow}$.*

Naturally, we cannot use the completion $C(q)$ above to reason about testing from a specification q under implementation relation $\preceq^{i}_{tr}$; $C(q)$ might have behaviours (timed traces) not allowed. However, this is not the case if we first convert q into an equivalent deterministic tockLTS, using a simple adaption of the classical transformation from non-deterministic to deterministic finite automata [24]. Given tockLTS q we will let $det(q)$ denote the determinised version; states of $det(q)$ will be sets of states of q reached by a common trace.

Definition 11. *Let $p = (Q, q_0, I, O, T)$ be a tockLTS. We write $det(p)$ to denote the automaton $(\mathcal{P}(Q), \{q_0\}, I, O, T', \mathcal{P}(Q))$ in which $(Q_1, a, Q_2) \in T'$ for $Q_1, Q_2 \in \mathcal{P}(Q)$ and $a \in I \cup O \cup \{\tau, \ominus\}$ if and only if $Q_2 = \bigcup\{q$ **after** $a | q \in Q_1\}$.*

Observe that we say that $det(p)$ is an automaton, rather than a tockLTS (and so include a set of final states), because $det(p)$ need not satisfy some of the requirements that we place on tockLTS. For example, $det(p)$ need not have urgent output; this might be the case if $p \stackrel{\sigma}{\Rightarrow} p_1$, $p \stackrel{\sigma}{\Rightarrow} p_2$, $p_1 \stackrel{\ominus}{\longrightarrow}$ and there is an output $!o$ such that $p_2 \stackrel{!o}{\longrightarrow}$.

Example 5. *Considering the tockLTS p_{drone} from Example 2 we will provide a few states and transitions for its determinised automaton. Here, the only sources of non-determinism in this model are due to τ transitions. Thus, it is relatively straightforward to construct $det(p_{drone})$. For example, from $\{O_3\}$ the action $\ominus$ moves $det(p_{drone})$ to $\{O_3, L_O\}$. Further, $!m$ takes $det(p_{drone})$ from $\{O_3, L_O\}$ to $\{L_1\}$.*

We obtain the following result. This follows by simply observing that in $det(q)$, an input $?a$ is specified after a trace σ if and only if we have that $q \stackrel{\sigma}{\Rightarrow} q_1$ for some state such that $q \stackrel{?a}{\longrightarrow}$.

Theorem 2. *Given tockLTS p and q with the same alphabets, $p \preceq^{i}_{tr} q$ if and only if $L(p) \subseteq L(C(det(q)))$.*

Note that in the above we did not use $\preceq_{tr}$ to compare p and $C(det(q))$ since $det(q)$ is an automaton and not necessarily a tockLTS.

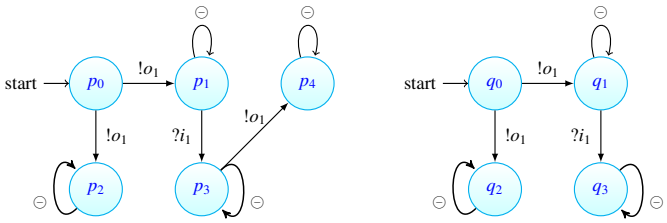


Figure 4: Models illustrating the difference between $\preceq^{u}_{tr}$ and $\preceq^{i}_{tr}$

Observe that the process of constructing $det(q)$ could, in principle, lead to states of $det(q)$ corresponding to infinite sets of states of q, which means that even bounded approaches to (partial) construction might not work. However, it is relatively common to assume that processes have the following property.

Definition 12. *A tockLTS $q = (Q, q_0, I, O, T)$ is* finitely-branching *if for every $q_1 \in Q$ and $a \in I \cup O \cup \{\ominus\}$ we have that q_1* **after** *a is finite.*

Under this condition, the states of $det(q)$ that are reached by finite traces correspond to finite sets of states of Q. The proof of the result easily follows by induction on the length of σ.

Proposition 4. *Given tockLTS q, if q is finitely-branching then for all $\sigma \in \mathcal{T}\mathtt{traces}(q)$ we have that q_1* **after** *σ is finite.*

We can now compare the above implementation relations.

Theorem 3. *Let p and q be tockLTS with the same alphabets. We have the following results:*

1. *If $p \preceq_{tr} q$ then $p \preceq^{u}_{tr} q$ and $p \preceq^{i}_{tr} q$.*
2. *It is possible that one or more of $p \preceq^{u}_{tr} q$ and $p \preceq^{i}_{tr} q$ holds but not $p \preceq_{tr} q$.*
3. *If $p \preceq^{i}_{tr} q$ then $p \preceq^{u}_{tr} q$.*
4. *It is possible that $p \preceq^{u}_{tr} q$ but not $p \preceq^{i}_{tr} q$.*

Proof. The first result is immediate from the definitions of $p \preceq_{tr} q$, $p \preceq^{u}_{tr} q$, and $p \preceq^{i}_{tr} q$.

For the second result, consider the specification tockLTS q given in Figure 3 (center) and p be the tockLTS given in Figure 3 (left). It is clear that $p \preceq^{u}_{tr} q$ and $p \preceq^{i}_{tr} q$ since under these relations, an implementation of q can do anything after receiving $?i_1$. However, we do not have that $p \preceq_{tr} q$ since, for example, $?i_1 !o_1$ is a timed trace of p but not q.

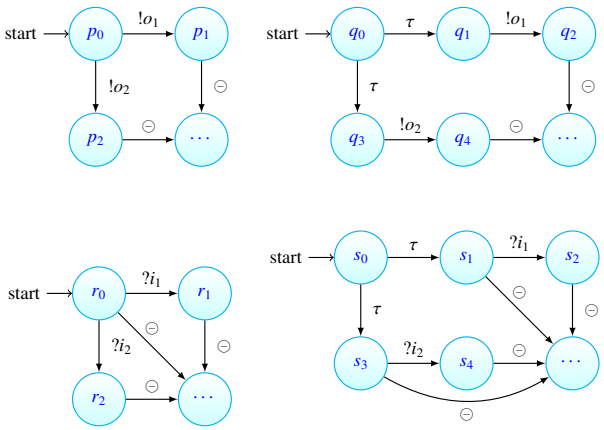


Figure 5: Models related by (refusal) timed trace inclusion

Now consider how $\preceq^u_{tr}$ and $\preceq^i_{tr}$ relate. First if $p \preceq^i_{tr} q$ then, by definition, for all $\sigma \in \mathcal{T}\texttt{traces}(p)$ either $\sigma \in \mathcal{T}\texttt{traces}(q)$ or there exists a prefix $\sigma_1?a$ of σ, with $?a \in I$, such that $\sigma_1 \in \mathcal{T}\texttt{traces}(q)$ and for all q_1 such that $q \stackrel{\sigma_1}{\Longrightarrow} q_1$ we have that $q_1 \stackrel{?a}{\not\rightarrow}$. If we consider the second part of this condition, this implies that there exists a prefix $\sigma_1?a$ of σ, with $?a \in I$, such that $\sigma_1 \in \mathcal{T}\texttt{traces}(q)$ and there is a state q_1 such that $q \stackrel{\sigma_1}{\Longrightarrow} q_1$ and $q_1 \stackrel{?a}{\not\rightarrow}$. But this means that σ is allowed under both $p \preceq^u_{tr} q$ and $p \preceq^i_{tr} q$.

For the last part, we require p and q such that $p \preceq^u_{tr} q$ but not $p \preceq^i_{tr} q$. Consider the tockLTSs p and q depicted in Figure 4. Essentially, they differ through it being possible for p to produce $!o_1$ after $!o_1?i_1$ while q cannot. We have that $p \preceq^u_{tr} q$ since $!o_1$ can take q to a state (q_2) in which $?i_1$ is not specified and so (under $\preceq^u_{tr}$) all behaviours are allowed after $!o_1?i_1$. In contrast, we do not have that $p \preceq^i_{tr} q$ since $!o_1$ can take q to a state (q_1) in which $?i_1$ is specified and so (under $\preceq^i_{tr}$) the only behaviours allowed after $!o_1?i_1$ are those that are traces of q. The result therefore follows.

The previous result shows that the three implementation relations differ in terms of how they deal with unspecified inputs. As a result, we have the following.

Theorem 4. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Let p and q be two elements of TockLTS(I, O). If q is input-enabled then the following three statements are equivalent: $p \preceq_{tr} q$; $p \preceq^u_{tr} q$; and $p \preceq^i_{tr} q$*

The trace based implementation relations have some benefits. For example, they have a number of nice properties and are relatively simple to define. In addition, they correspond to inclusion between two formal languages and so we can use concepts and results from formal language theory in order to reason about them. However, there is potential to strengthen them by allowing refusals to be observed. As a result, there are some systems that are related under these implementation relations but that, intuitively, should not be. The following discusses timed trace inclusion but similar arguments can be applied to the other two implementation relations defined in this section.

Example 6. *Let us consider the fragments of models given in Figure 5 (top and bottom). These two pairs of models conform to each other under timed trace inclusion because we have $\mathcal{T}\texttt{traces}(p) \subseteq \mathcal{T}\texttt{traces}(q)$ and $\mathcal{T}\texttt{traces}(q) \subseteq \mathcal{T}\texttt{traces}(p)$, $\mathcal{T}\texttt{traces}(r) \subseteq \mathcal{T}\texttt{traces}(s)$ and $\mathcal{T}\texttt{traces}(s) \subseteq \mathcal{T}\texttt{traces}(r)$. However, often we will want to be able to distinguish between such processes. On the one hand, we expect both conformances between p and q because outputs cannot be controlled by the environment. In other words, a choice between outputs should work exactly as the corresponding internal choice. For example, even though $?o_1$ and $?o_2$ are available at p_0, a user/tester cannot choose which of them will be performed. On the other hand, r and s should not be equivalent. The issue is that the tester or user can choose between two inputs in the same way that one can choose among the available options in a vending machine. If we have the corresponding internal choice and we reach, for example, state s_1 then input $?i_2$ is not available. The implementation relation that we present in the next section satisfies all of these properties.*

5. Timed refusal traces

In this section we extend the notion of a timed trace to allow refusals to be observed, leading to timed refusal traces. We then define the timed refusal traces of a tockLTS. Finally, we develop an approach in which a tockLTS p is transformed into an automaton $\mathcal{M}(p)$ whose language describes the set of timed refusal traces of p. In the next section we define implementation relations based on timed refusal traces and prove that we can express these in terms of language inclusion based on $\mathcal{M}(p)$.

Recall that we are interested in models that are cyclic/have a step semantics: a sequence of actions occurs without time (in the model) passing and then there is a tock action. A refusal of a set $X \subseteq L$ is typically observed through the tester only being willing to engage in the actions in X and the composition of the tester and the SUT deadlocking. Since deadlocks are observed (in testing) through timeouts, the observation of a refusal takes time and so we only allow a refusal to be observed immediately before a tock action. Since outputs and internal actions are urgent, this means that a refusal can only be observed in a stable state.

Definition 13 (Stable state). *Let $p = (Q, q_0, I, O, T)$ be a tockLTS, with $L = I \cup O$. We say that the state $q \in Q$ is* stable *if for all $a \in O \cup \{\tau\}$ we have that $q \stackrel{a}{\not\rightarrow}$.*

Given a set $X \subseteq L$ of actions, we use $R(X)$ to denote the refusal of set X. Further, we let $\mathcal{R}(L) = \{R(X) | X \subseteq L\}$ denote the set of all possible refusals.

We can extend the transition relation of a tockLTS with refusals as follows.

Definition 14 (Refusal). *Let $p = (Q, q_0, I, O, T)$ be a tockLTS and $X \subseteq I \cup O$. For all $q \in Q$ we write $q \xrightarrow{R(X)}$ if the following hold:*

1. $q \stackrel{\ominus}{\longrightarrow}$ *and*

2. *for all* $x \in X$ *we have that* $q \overset{x}{\not\longrightarrow}$.

This constitutes the observation of the refusal $R(X)$*, that is, at a given stable state the model cannot perform the actions belonging to* X*.*

Note that, as previously indicated, due to the urgency of outputs and τ we have that if a $\ominus$ can occur in a state q then q must be stable. As a result of this definition, the observation of a refusal $R(X)$ implies that no element $a \in X \cup O \cup \{\tau\}$ can occur in state q: for all $a \in X \cup O \cup \{\tau\}$ we have that $q \overset{a}{\not\longrightarrow}$. We therefore obtain the following result.

Proposition 5. *Given tockLTS* $p = (Q, q_0, I, O, T)$*,* $q \in Q$ *and* $X \subseteq I \cup O$*, we have that* $q \xrightarrow{R(X)} q$ *if and only if* $q \xrightarrow{R(X \cup O)} q$*.*

Example 7. *In order to illustrate refusals, we consider the two models from Figure 5 (top). It can be easily checked that states* p_0, q_0, q_1, q_3 *are not stable. In addition, refusals could be observed in any stable state of the two tockLTS models,* $q' \in \{p_1, p_2, q_2, q_4\}$*. We denote this by* $q' \xrightarrow{R(X)} q'$*, where* $X \subseteq I \cup O$*.*

For our running example p_{drone}*, refusals can be observed only in the stable states in which* $\ominus$ *is possible. This is exactly the set* $\{D_0, D_2, O_3\}$ *of states. The transition relation can be extended with self-loops for refusals in each state* $q' \in \{D_0, D_2, O_3\}$*. For example, we will have the transition* $D_0 \xrightarrow{R(\{?f,?o,?s\})} D_0$*.*

Note also that the second condition of Definition 14 implies that we include $R(X)$ as a refusal if all the actions in X can be refused, even if there are other actions from $L \setminus X$ that can be refused. Therefore, we do not only include *maximal* refusals. In fact, doing this would lead to some undesirable effects (this will be clearer after we give our implementation relations using refusals).

We can then give the set of refusal traces of a tockLTS in which, as we already said, $p \overset{\sigma}{\Longrightarrow}$ is defined in terms of $p \overset{x}{\rightarrow}$ and $p \xrightarrow{R(X)}$, in the usual way. Recall, however, that a refusal can only be observed immediately before a tock action. We therefore obtain a set of potential refusal traces (those that satisfy this condition) and we call these *timed refusal traces*. Also note that a timed refusal trace cannot end in a refusal since the observation of a refusal takes time (and so must be followed by a $\ominus$). As a result, this set is not prefix closed.

Definition 15 (Timed refusal traces). *Let* L *be a set of actions. We define the set of* timed refusal traces *over* L *as* $RT(L) = (L^* \cup (\mathcal{R}(L)\{\ominus\}))^*$*.*

Let $p = (Q, q_0, I, O, T)$ *be a tockLTS, with* $L = I \cup O$*. The set of* timed refusal traces *of* p *is defined as*

$$\mathcal{TR}\texttt{traces}(p) = \{\sigma \in (L \cup \{\ominus\} \cup \mathcal{R}(L))^* | p \overset{\sigma}{\Longrightarrow}\} \cap RT(L)$$

Example 8. *Considering again* p_{drone} *from Figure 2 with refusals added in the stable states* $\{D_0, D_2, O_3\}$*, as explained in the previous example, some timed refusal traces are:*

$\sigma_1 = ?s \cdot !t \cdot R(\{?f, ?o, ?s\}) \cdot \ominus \cdot !m \cdot ?f \cdots$

$\sigma_2 = ?s \cdot !t \cdot \ominus \cdot !m \cdot ?f \cdot !l \cdot R(\{?f, ?o\}) \cdot \ominus \cdots$

Trace inclusion corresponds to a relation between the languages defined by the automata corresponding to two LTS. The benefit is that it is possible to use standard results and algorithms from formal language theory. This is particularly useful if the processes are deterministic finite state automata since there are efficient algorithms for many standard problems, including deciding language inclusion (that is, trace inclusion in our setting). We now show how we can generate an automaton whose traces are exactly the timed refusal traces of a tockLTS q.

In order to explore one approach that might be used to achieve this, consider the fragment of a model in Figure 6 (a). This can refuse all actions other than $?i_2$ when in state q_1. It might seem that we can simply add a self-loop transition, with such a refusal, in state q_1. However, we would then have the problem that such a self-loop need not be followed by a $\ominus$ action. For example, the inclusion of such a self-loop in state q_1 would allow refusal traces such as $?i_1 R(\{?i_1\})?i_2$. Such a refusal trace should not be allowed since it has a refusal followed by an action other than $\ominus$.

One possible solution is outlined in Figure 6 (b). Rather than adding a self-loop, we include a transition, to a new state $\widetilde{q_1}$, that is labelled with the refusal. From $\widetilde{q_1}$ there is only one possible action, which is $\ominus$. We also require that $\widetilde{q_1}$ is not a final state of the automaton. As a result, any path that reaches a final state and includes the transition from q_1 to $\widetilde{q_1}$ must follow this transition by a transition with label $\ominus$. Note that we require the notion of a final state and so the model is an *automaton* and not a tockLTS.

We now formally define the automaton $\mathcal{M}(p)$ that includes these refusals.

Definition 16. *Let* $p = (Q, q_0, I, O, T)$ *be a tockLTS, with* $L = I \cup O$*. Let us consider the subset of states that can evolve by performing* $\ominus$*, that is, the set* $Q_\ominus = \{q \in Q | q \xrightarrow{\ominus}\}$*. We define a set of fresh states* $\tilde{Q} = \{\tilde{q} | q \in Q_\ominus\}$ *(i.e.* $Q \cap \tilde{Q} = \emptyset$*). The new set of states* $\tilde{Q}$ *has a state for each state of* $Q_\ominus$*.*

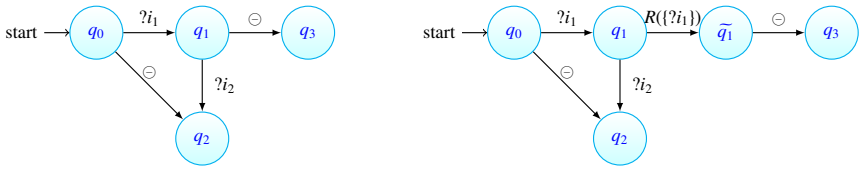
We let $\mathcal{M}(p)$ *denote the automaton* $(Q \cup \tilde{Q}, q_0, I \cup O \cup \mathcal{R}(L), T', F)$ *where*

- $T' = T \cup \{(q, R(X), \tilde{q}) | q \in Q_\ominus \wedge q \xrightarrow{R(X)}\} \cup \{(\tilde{q}, \ominus, q') | q \in Q_\ominus \wedge q \xrightarrow{\ominus} q'\}$.
- $F = Q$.

The following result shows that the previous construction is correct.

Theorem 5. *Let* $p = (Q, q_0, I, O, T)$ *be a tockLTS. We have that* $\mathcal{TR}\texttt{traces}(p) = L(\mathcal{M}(p))$*.*

Proof. First, observe that both sets are subsets of $RT(L)$, where $L = I \cup O$. We will prove a slightly stronger result than the one stated before. Specifically, we will prove that for all $\sigma \in RT(L)$, we have that σ takes p to state q if and only if q is a final state of $\mathcal{M}(p)$ and σ takes $\mathcal{M}(p)$ to state q.



(a) Original model

(b) Refusal automaton

Figure 6: A refusal can only happen immediately before a duration or tock $\ominus$

We use proof by induction on the length of σ. The base case, with σ being the empty sequence, is immediate.

Inductive hypothesis: the result holds if the sequence has length less than k ($k > 0$). Let us suppose that σ has length k and σ takes one of p and $\mathcal{M}(p)$ to state q. By the definition of $RT(L)$, $\sigma = \sigma_1 a$ for some $a \in L \cup \{\ominus\}$ (i.e. sequences in $RT(L)$ cannot end in refusals). There are two cases to consider. First, if σ_1 does not end with a refusal then, by the inductive hypothesis, we have that σ_1 reaches the same states in p and $\mathcal{M}(p)$. In addition, by construction we have that a takes p and $\mathcal{M}(p)$ to the same state and so the result follows. The second case is where σ_1 ends in a refusal and so $\sigma = \sigma_2 R(X)\ominus$ for some $X \subseteq L$ and $\sigma_2 \in RT(L)$. By the inductive hypothesis, σ_2 takes p and $\mathcal{M}(p)$ to the same state q_1. By construction, $R(X)\ominus$ takes p and $\mathcal{M}(p)$ to the same state q and so the result follows.

6. Implementation relations with refusals

In the previous section we defined the notion of a timed refusal trace. In this section we extend the previous implementation relations in the natural way: we base implementation relations on the timed refusal traces of a process and not just its timed traces.

Definition 17. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Let p and q be two elements of TockLTS(I, O). We say that p conforms to q under* timed refusal trace inclusion *if and only if $\mathcal{TR}\mathtt{traces}(p) \subseteq \mathcal{TR}\mathtt{traces}(q)$. We denote this $p \preceq_{\mathcal{T}} q$.*

We now examine this implementation relation before considering alternatives that treat unspecified inputs in different ways. First, we present an example showing some relations between models and why maximal refusals do not provide the expected implementation relation.

Example 9. *Consider again the fragments of models p and q given in Figure 5 (top). We cannot add refusals to traces in states p_0, q_0, q_1 and q_3 because they are not stable. Therefore, we have $\mathcal{TR}\mathtt{traces}(p) \subseteq \mathcal{TR}\mathtt{traces}(q)$ and $\mathcal{TR}\mathtt{traces}(q) \subseteq \mathcal{TR}\mathtt{traces}(p)$.*

Consider now r and s given in Figure 5 (bottom). Assuming that $I = \{?i_1, ?i_2\}$ and $O = \emptyset$, we have the following sets[5] of timed refusal traces:

$$\mathcal{TR}\mathtt{traces}(r) = \{?i_1 \cdots, ?i_2 \cdots, R(\emptyset)\ominus \cdots, \ldots\}$$

$$\mathcal{TR}\mathtt{traces}(s) = \left\{ \begin{array}{l} ?i_1 \cdots, ?i_2 \cdots, R(\emptyset)\ominus \cdots, \\ R(\{?i_1\})\ominus \cdots, R(\{?i_2\})\ominus \cdots, \ldots \end{array} \right\}$$

We have $\mathcal{TR}\mathtt{traces}(r) \subseteq \mathcal{TR}\mathtt{traces}(s)$, so that r conforms to s under timed refusal trace inclusion, but the converse is not the case. This shows that an external choice between inputs is a good implementation of the internal choice between the same inputs.

These last two fragments help show why we cannot restrict ourselves to only computing the maximal refusal sets. If we were to do this, the timed refusal traces of r would be the same but the ones corresponding to s would be $R(\{?i_1\})\ominus, R(\{?i_2\})\ominus, \ldots$ and we would no longer have timed trace inclusion.

The following is immediate from Theorem 5.

Theorem 6. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Let p and q be two elements of TockLTS(I, O). Then $p \preceq_{\mathcal{T}} q$ if and only if $L(\mathcal{M}(p)) \subseteq L(\mathcal{M}(q))$.*

We now define two alternative implementation relations. The first approach corresponds to that used with $\preceq^{u}_{tr}$. Similar to before, the idea is that any behaviour is allowed if an unspecified input is received. However, since behaviours are now timed refusal traces, rather than timed traces, the definitions are written in terms of the states reached by a prefix of a timed refusal trace.

Definition 18. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Let p and q be two elements of TockLTS(I, O). We say that $p \preceq^{u}_{\mathcal{T}} q$ if and only if for all $\sigma \in \mathcal{TR}\mathtt{traces}(p)$ either $\sigma \in \mathcal{TR}\mathtt{traces}(q)$ or there exists a prefix $\sigma_1 ?a$ of σ, with $?a \in I$, such that $\sigma_1 \in \mathcal{TR}\mathtt{traces}(q)$ and there exists q_1 such that $q \stackrel{\sigma_1}{\Longrightarrow} q_1$ and $q_1 \stackrel{?a}{\not\Rightarrow}$.*

The second approach corresponds to that used with the implementation relation $\preceq^{i}_{tr}$.

Definition 19. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Let p and q be two elements of TockLTS(I, O). We say that $p \preceq^{i}_{\mathcal{T}} q$ if for all $\sigma \in \mathcal{TR}\mathtt{traces}(p)$ either $\sigma \in \mathcal{TR}\mathtt{traces}(q)$ or there exists a prefix $\sigma_1 ?a$ of σ, with $?a \in I$, such that $\sigma_1 \in \mathcal{TR}\mathtt{traces}(q)$ and for all q_1 such that $q \stackrel{\sigma_1}{\Longrightarrow} q_1$ we have that $q_1 \stackrel{?a}{\not\Rightarrow}$.*

[5] We only enumerate the relevant elements to show the differences between the models.

We can compare the three implementation relations that are based on timed refusal traces, with the proof of the following being equivalent to that of the corresponding result for traces (Theorem 3).

Theorem 7. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Let p and q be two elements of TockLTS(I, O). Then the following hold.*

1. *If $p \preceq_{\mathcal{T}} q$ then $p \preceq^{u}_{\mathcal{T}} q$ and $p \preceq^{i}_{\mathcal{T}} q$.*
2. *It is possible that one or both of $p \preceq^{u}_{\mathcal{T}} q$ and $p \preceq^{i}_{\mathcal{T}} q$ hold but not $p \preceq_{\mathcal{T}} q$.*
3. *If $p \preceq^{i}_{\mathcal{T}} q$ then $p \preceq^{u}_{\mathcal{T}} q$.*
4. *It is possible that $p \preceq^{u}_{\mathcal{T}} q$ but not $p \preceq^{i}_{\mathcal{T}} q$.*

Similar to before, the implementation relations coincide if the specification is input-enabled.

Theorem 8. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Let p and q be two elements of TockLTS(I, O). If q is input-enabled then the following three statements are equivalent: $p \preceq_{\mathcal{T}} q$; $p \preceq^{u}_{\mathcal{T}} q$; and $p \preceq^{i}_{\mathcal{T}} q$.*

We can now compare the implementation relation defined in this section with those based on trace inclusion introduced in Section 4. The proof of the following result follows from the fact that, for a process r, we have $\mathcal{T}\texttt{traces}(r) = \mathcal{TR}\texttt{traces}(r) \cap L^{*}$.

Proposition 6. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Let p and q be two elements of TockLTS(I, O). Then the following hold.*

1. *If $p \preceq_{\mathcal{T}} q$ then $p \preceq_{tr} q$.*
2. *If $p \preceq^{u}_{\mathcal{T}} q$ then $p \preceq^{u}_{tr} q$.*
3. *If $p \preceq^{i}_{\mathcal{T}} q$ then $p \preceq^{i}_{tr} q$.*

However, the converse is not the case as the following result shows.

Proposition 7. *Let I and O be countable disjoint sets of inputs and outputs, respectively. There exist p and q in TockLTS(I, O) such that $p \preceq_{tr} q$ but $p \preceq_{\mathcal{T}} q$ does not hold.*

Proof. In order to prove this it is sufficient to give an example of such tockLTSs. Consider r and s depicted in Figure 5 (bottom). In Example 6 we showed that $\mathcal{T}\texttt{traces}(r) = \mathcal{T}\texttt{traces}(s)$. Therefore, s conforms to r under timed trace inclusion. On the contrary, in Example 9 we showed that $\mathcal{TR}\texttt{traces}(s) \not\subseteq \mathcal{TR}\texttt{traces}(r)$. Therefore, s does not conform to r under timed refusal trace inclusion.

To summarise, $s \preceq_{tr} r$ but $s \preceq_{\mathcal{T}} r$ does not hold. The result therefore holds.

Note that the above proof considered parts of tockLTS models in which all inputs are specified and thus is not affected by whether we compare $\preceq_{tr}$ with $\preceq_{\mathcal{T}}$, $\preceq^{u}_{tr}$ with $\preceq^{u}_{\mathcal{T}}$, or $\preceq^{i}_{tr}$ with $\preceq^{i}_{tr}$. We therefore have the following.

Proposition 8. *Let I and O be countable disjoint sets of inputs and outputs, respectively. There exist p and q in TockLTS(I, O) such that the following hold.*

1. *$p \preceq^{u}_{tr} q$ but not $p \preceq^{u}_{\mathcal{T}} q$*
2. *$p \preceq^{i}_{tr} q$ but not $p \preceq^{i}_{\mathcal{T}} q$*

We therefore obtain the following result, that says that the implementation relations that use timed refusal traces are strictly stronger than the corresponding implementation relations that only look at traces.

Theorem 9. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Let p and q be two elements of TockLTS(I, O). Then the following hold.*

1. *If $p \preceq_{\mathcal{T}} q$ then $p \preceq_{tr} q$ but the converse does not hold.*
2. *If $p \preceq^{u}_{\mathcal{T}} q$ then $p \preceq^{u}_{tr} q$ but the converse does not hold.*
3. *If $p \preceq^{i}_{\mathcal{T}} q$ then $p \preceq^{i}_{tr} q$ but the converse does not hold.*

The previous result tells us that if we can observe timed refusal traces in testing then we have more powerful implementation relations than the ones we obtain when we only consider timed traces. It is also the case that if the environment (e.g. the user) can observe timed refusal traces (through, for example, the refusal of actions being observed as a result of options not being available on a screen) then it is insufficient to test for trace inclusion and its variants: the user might consider an SUT p to be faulty with respect to a specification q even though, for example, they have the same sets of timed traces.

Finally, we show how timed refusal trace inclusion can be expressed in terms of language containment between automata. The following is an immediate consequence of Theorem 5 and tells us that $\mathcal{M}(q)$ captures the behaviours we require when reasoning about timed refusal trace inclusion.

Theorem 10. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Let p and q be two elements of TockLTS(I, O). We have that $p \preceq_{\mathcal{T}} q$ if and only if $\mathcal{TR}\texttt{traces}(p) \subseteq L(\mathcal{M}(q))$.*

7. Alternative characterisation based on observers

In this section we provide an alternative characterisation of the implementation relations that are based around timed refusal trace inclusion.

Implementation relations should correspond to the ability of the environment, or a tester, to distinguish between processes; typically we require that all observations that can be made of the SUT are also observations that can be made when interacting with the specification (see, for example, [45]). In this section we define the notion of an observer, in our context, and how such an observer interacts with a tockLTS. This will provide an alternative, but equivalent, characterisation of timed refusal trace inclusion.

We follow the classical approach of ioco [45], in which a special action θ is included in an observer to denote the observation of a refusal. In the case of ioco, as previously explained, the only refusal is associated with the inability of a model to perform output at a certain state (this is denoted by δ in the models and by θ in the observer/tester).

Next we define the notion of observer. An observer is an automaton in which all states are final. We also require that certain additional constraints are satisfied.

Definition 20 (Observer). *Let I and O be countable disjoint sets of inputs and outputs, respectively. An* observer $u = (Q, q_0, I \cup O \cup \{\ominus, \theta\}, T, Q)$ *is an automaton that satisfies the following properties for each state $q \in Q$:*

1. *If $q \overset{\ominus}{\not\rightarrow}$ then for all $a \in O$ we have that $q \overset{a}{\rightarrow}$.*
2. *There exists at most one $q' \in Q$ such that $q \overset{\ominus}{\rightarrow} q'$. If there is such a q' then for all $a \in O$ we have that $q \overset{a}{\not\rightarrow}$.*
3. *If (q, θ, q') is a transition of u then $\ominus$ is the only action available in state q'.*

We let $\mathcal{U}(I, O)$ denote the set of observers with input set I and output set O.

Note that observers cannot perform internal transitions (the rationale is that they essentially *record* what they can observe from a process). The last rule ensures that a refusal must be followed by a $\ominus$. The first rule is the standard condition that a tester is able to observe outputs while the second rule indicates that time can pass only if no output is available. Finally, the observation of a refusal takes time and so a θ must be followed by a $\ominus$.

Example 10. *In Figure 7 we provide an observer for the tockLTS drone model given in Figure 2. Note that there are many possible observers and, in addition, the graphical representation captures only part of an observer. It, does, however, illustrate the observation of a refusal (the θ-labelled transition), time passing ($\ominus$-transition) and acceptance of all outputs.*

We can now define a parallel composition operator $\between$ between a process $p \in \mathit{TockLTS}(I, O)$ and an observer $u \in \mathcal{U}(I, O)$. This is similar to the operators for LTS [45] but we choose to enrich the observations made with refusal sets.

Definition 21 (Synchronised parallel communication). *Let I and O be countable disjoint sets of inputs and outputs, respectively. Let $p = (Q, q_0, I, O, T) \in \mathit{TockLTS}(I, O)$ and $u = (Q', q'_0, I \cup O \cup \{\ominus, \theta\}, T', Q') \in \mathcal{U}(I, O)$. The composition of the observer u and the model p, denoted by $u \between p$, is an automaton $(Q \times Q', (q_0, q'_0), I \cup O \cup \mathcal{R}(I \cup O) \cup \{\ominus\}, T'', Q \times Q')$ in which T'' is defined as follows:*

- *If $(q_1, \tau, q_2) \in T$ then for all $q' \in Q'$ we have $((q_1, q'), \tau, (q_2, q')) \in T''$.*
- *If $(q_1, a, q_2) \in T$ and $(q'_1, a, q'_2) \in T'$, with $a \in I \cup O \cup \{\ominus\}$, then we have $((q_1, q'_1), a, (q_2, q'_2)) \in T''$.*
- *Let $X \subseteq I \cup O$. If $(q_1, \ominus, q_2) \in T$ and $(q'_1, \theta, q'_2) \in T'$ then $((q_1, q'_1), R(X), (q_1, q'_2)) \in T''$ if and only if the following conditions hold:*
  - *for all $a \in I \cup O$ we have that either there does not exist q_3 such that $(q_1, a, q_3) \in T$ or there does not exist q'_3 such that $(q'_1, a, q'_3) \in T'$.*

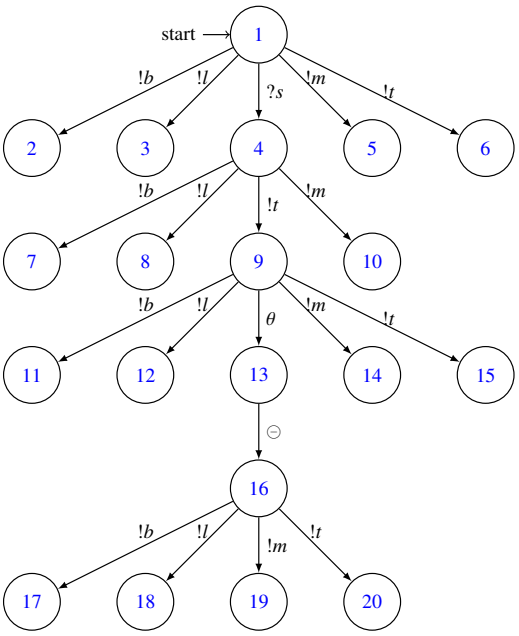
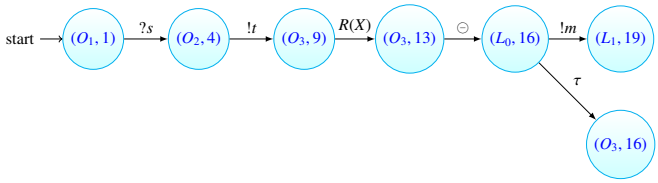


Figure 7: Observer example



where $X \subseteq \{!b, !l, !m, !t\}$.

Figure 8: Composition of a tockLTS with an observer

  - *for all $a \in X$ we have that there exists q'_3 such that $(q'_1, a, q'_3) \in T'$.*

The set of observations that the observer u can make of p, denoted by $obs^{\theta}(u, p)$, is given by the following:

$$obs^{\theta}(u, p) =_{\text{def}} \{\sigma \in (I \cup O \cup \mathcal{R}(L) \cup \{\ominus\})^* \mid u \between p \overset{\sigma}{\Longrightarrow}\}$$

Note that in the last rule of the composition, since q_1 may evolve via $\ominus$ then we have that it must be a stable state (for all $a \in O \cup \{\tau\}$ we have that $q_1 \overset{a}{\not\rightarrow}$); this follows from the fact that p is a tockLTS and tockLTSs have urgent outputs and internal actions (Definition 6). In this rule, also note that we *discard* the state reached after the performance of $\ominus$ from q_1: the composition makes p remain in the same state. Naturally, the second rule then ensures that this can be followed by a $\ominus$.

Example 11. *Figure 8 shows* part *of the composition of the observer given in the previous example (Figure 7) with the tockLTS model from Figure 2.*

The following shows how observations relate to timed refusal traces and is a result of the definition of $u \between p$ and the definition of timed refusal traces (Definition 15).

Proposition 9. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Given $\sigma \in (I \cup O \cup \{\ominus\} \cup \mathcal{R}(L))^*$ and p in TockLTS(I, O), there is an observer $u \in \mathcal{U}(I, O)$ such that $\sigma \in obs^{\theta}(u, p)$ if and only if σ is a prefix of a timed refusal trace of p.*

Note that in the above result σ need not be a timed refusal trace of p since σ could end in a refusal; to make this a timed refusal trace it would be necessary to add the $\ominus$ that follows this refusal. The following result is immediate from Proposition 9.

Theorem 11. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Let p and q be two elements of TockLTS(I, O). We have that $p \preceq_{\mathcal{T}} q$ if and only if for all $u \in \mathcal{U}(I, O)$ we have that $obs^{\theta}(u, p) \subseteq obs^{\theta}(u, q)$.*

Since observers capture the observations that can be made, this tells us that timed refusal trace inclusion is a suitable implementation relation for our scenario.

8. Test generation

In this section we present a testing framework for cyclic systems specified by using a tockLTS. The goal is to use the specification of the SUT to derive *sound* and *exhaustive* test suites with respect to an implementation relation. These test suites are said to be *complete*. The soundness of the process ensures that if an SUT fails one of the derived tests then the SUT is faulty (in other words, it does not conform to the specification with respect to the chosen implementation relation). In addition, exhaustiveness ensures that if an SUT is faulty then there exists a derived test case that is failed by the SUT. While soundness is relatively easy to achieve (in particular, the empty set of tests is sound), exhaustiveness is usually not reached if the SUT is a black-box. The problem is that we do not have a bound on the number of tests that we have to apply. We will provide an algorithm that achieves exhaustiveness *in the limit* [34]. The idea is that given a natural number n, we can construct test suites that find all faults that can be observed in observations of length up to n. If n tends to infinity then we achieve exhaustiveness in the limit. A recent approach [5] shows how one can derive test suites that are exhaustive as long as the number of states of the SUT does not exceed a given bound. Both approaches have a similar underlying idea: if the length of the tested sequences or the number of states tends to infinity then the derived test suites are complete in the limit.

First, we define a testing framework where testers do not have the capabilities to detect refusals. As a result, we will be able to characterise our implementation relation based on the timed traces of a process, that is, the $\preceq_{tr}$ relation. Next, we will extend the tests with refusals and will be able to provide a characterisation of the implementation relation based on timed refusal traces, that is, the $\preceq_{\mathcal{T}}$ relation.

8.1. Complete test suites with respect to $\preceq_{tr}$

Our first task is to define the notion of *test*. Usually, a test case can either provide an input to the SUT or observe output. In our framework, a test case can also observe the passing of time. A difference with respect to the usual notion of test case is that we will also test for the (in)ability of the SUT to perform an input at a certain point of time. In this case, if testing detects that the SUT is able to perform an input that is not allowed by the specification then we will stop the testing process and indicate that a faulty behaviour was detected. Similarly, in the next section we will test for the capabilities of the SUT to perform refusals (and test that they are followed by a tock action indicating the passing of time). There are two main differences with respect to usual automata: tests do not have internal actions (actually, they are deterministic) and they must represent a finite process, that is, the induced graph cannot have cycles. Finally, note that the notions of test case and observer, introduced in the previous section, are related but tests are more restrictive: they are deterministic, have finite behaviour and at most one input can be used to continue the testing process from each state of the test.

Definition 22 (test). *Let I and O be countable disjoint sets of inputs and outputs respectively. An I/O-*test *(or simply test when I and O can be inferred from the context) is an acyclic and deterministic automaton $p = (Q, q_0, I \cup O \cup \{\ominus\}, T, F)$ such that Q is finite and for all $q \in F$ we have that either there are no outgoing transitions from q or the following two restrictions hold:*

- *Observe any possible output or passing of time. For each $a \in O \cup \{\ominus\}$ there exists $q_a \in Q$ such that $(q, a, q_a) \in T$.*
- *Continue with at most one input. There exists at most one input $?i$ such that $q \xrightarrow{?i}$.*

In addition, we require that non-final states, denoting that the testing process has failed, are deadlocked, that is, if $q \notin F$ then there does not exist $a \in I \cup O \cup \{\ominus\}$ and $q' \in Q$ such that $(q, a, q') \in T$.

We use $\mathcal{T}(I, O)$ to denote the set of I/O-tests.

In a test case, we distinguish three types of states. Two of them are states in which an input is not applied. This can mean either that the test case has been successfully passed or failed (graphically represented by ✓ and ×, respectively). In the first case (✓), there are two cases: the test case has terminated or the test case is in a state where it can observe further output, or the passing of time, and then continue. Failing states (×) always lead to the test case terminating and the set of failing states is given by $Q \backslash F$. The third type of state of a test case represents the potential application of an input to the SUT, paired with the potential to observe any output or the passing of time. This third type encompasses two different situations. If the state reached after the input belongs to F, then we have an *expected* input; otherwise it is an input that was not allowed and indicates that a fault has been detected. Note that we require tests to have a finite number of states, capturing the idea that the application of a test case should produce a verdict in finite time. In practice, if the set of outputs is infinite then we might *compress* transitions so that we retain finiteness. Given a state q we could, for

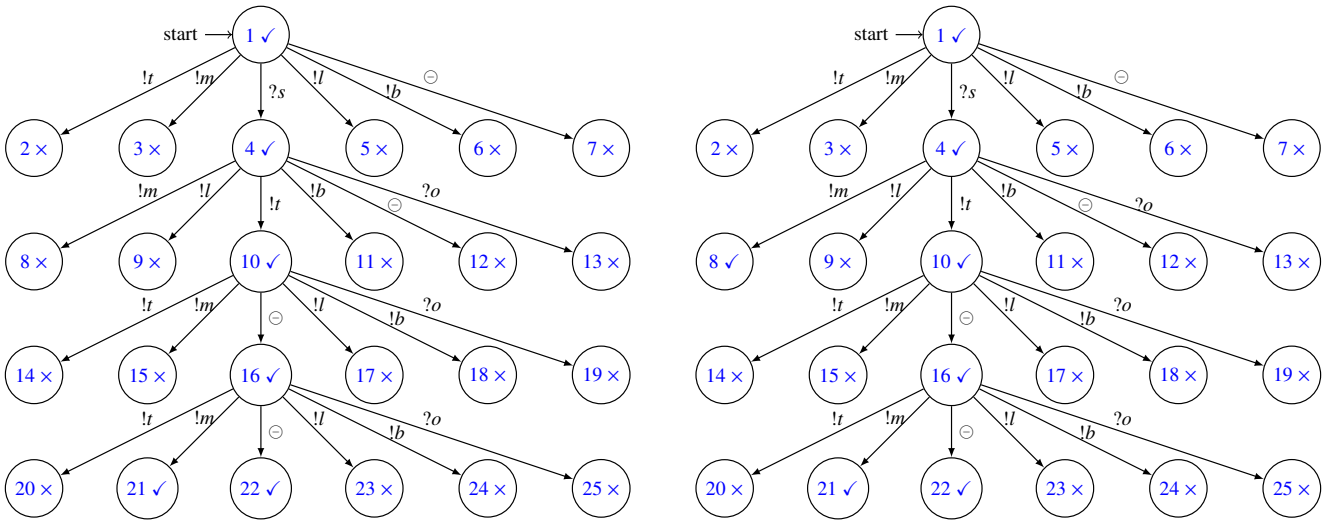
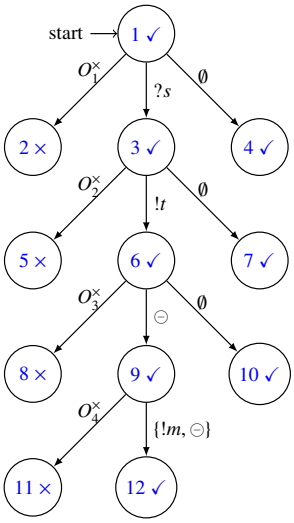


Figure 9: I/O-tests



where $O_1^\times = \{!t, !m, !l, !b, \ominus\}$, $O_2^\times = \{!m, !l, !b, \ominus, ?o\}$, $O_3^\times = \{!t, !m, !l, !b, ?o\}$ and $O_4^\times = \{!t, !l, !b, ?o\}$.

Figure 10: I/O-test in compressed form

example, partition the set of transitions from q to deadlocked states into two groups: the ones reaching pass and fail states, respectively. For example, if we consider the test case given in Figure 9 (left), then we would obtain the test case given in Figure 10. For the sake of simplicity, we will assume that the tests are in an *uncompressed* form but will rely to its equivalent representation if we are working with an infinite set of outputs.

Next, we define the application of a test to an SUT. Essentially, we will compose in parallel the test case and the SUT and we will say that the application of the test case has failed if a non-final state of the test case can be reached.

Definition 23 (test application). *Let I and O be countable disjoint sets of inputs and outputs respectively. Let $p = (Q, q_0, I, O, T)$ be a tockLTS and $t = (Q', q_0', I \cup O \cup \{\ominus\}, T', F)$ be an I/O-test. We define the application of the test case t to the system p, denoted by $p \intercal t$, as the automaton $(Q \times Q', (q_0, q_0'), I \cup O \cup \{\ominus, \tau\}, T'', Q \times F)$ where T'' is the smallest set of transitions fulfilling the following rules*

- *If $(q_1, \tau, q_2) \in T$ then for all $q' \in Q'$ we have $((q_1, q'), \tau, (q_2, q')) \in T''$.*
- *For each $a \in I \cup O \cup \{\ominus\}$, if $(q_1, a, q_2) \in T$ and $(q_1', a, q_2') \in T'$ then $((q_1, q_1'), a, (q_2, q_2')) \in T''$.*

We say that the application of t to p has failed *if there is a sequence of transitions belonging to T'' departing from (q_0, q_0') and reaching a state belonging to $Q \times (Q' \backslash F)$; otherwise, we say that the application was* successful.

Example 12. *Consider the I/O-tests t_1 (Figure 9 (left)) and t_2 (Figure 9 (right)), where I and O are the sets of inputs and outputs, respectively, corresponding to our running example. Consider the (faulty) implementation of our running example p_1 given in Figure 11. We have that the application of t_1 to p_1 fails because the sequence $?s!m$, which can be performed by $p_1 \intercal t_1$, reaches a fail state of the test. On the contrary, the application of t_2 to p_1 is successful.*

The following result is immediate from the definition of $\intercal$.

Lemma 1. *Let I and O be countable disjoint sets of inputs and outputs respectively. Let $p \in TockLTS(I, O)$, t be an I/O-test and $\sigma \in (I \cup O \cup \{\ominus\})^*$. We have that $p \intercal t \overset{\sigma}{\Longrightarrow}$ if and only if $\sigma \in \mathcal{T}\texttt{traces}(p)$ and $t \overset{\sigma}{\Longrightarrow}$.*

Next we present an algorithm to derive tests from a specification such that an SUT conforms to the specification with respect to $\preceq_{tr}$ if and only if the SUT successfully passes all the tests of the suite. The algorithm is given in Figure 12. The application of the algorithm to a specification produces a single test. If

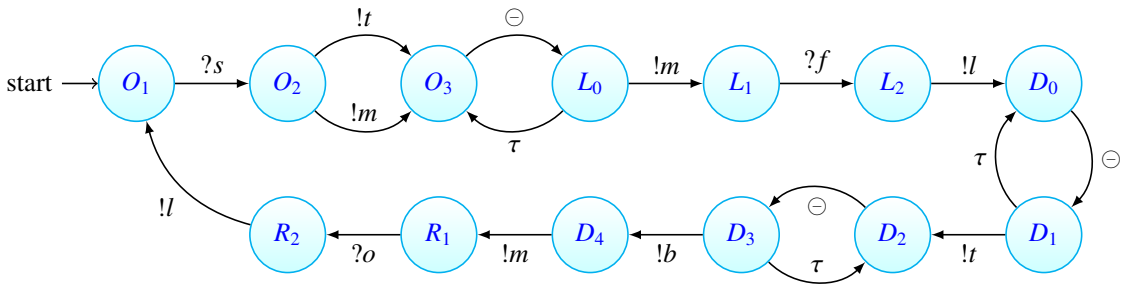


Figure 11: Faulty implementation of the TockLTS drone model

`Input`: A specification $q = (Q, q_0, I, O, T)$.
`Output`: A test case $t = (Q', q'_0, I \cup O \cup \{\ominus\}, T', F)$.

`Initialization`

Set a fresh state q'_0 as initial state; $Q', F := \{q'_0\}$; $T' := \emptyset$; $Q_{aux} := \{(\{q_0\}, q'_0)\}$.

`Loop`

While $Q_{aux} \neq \emptyset$ do

1. Choose $(P, q') \in Q_{aux}$; $Q_{aux} := Q_{aux} \backslash \{(P, q')\}$.
2. Select one of the following alternatives.
   (a) Successful termination (by construction, $q' \in F$).
   (b) Continue testing: apply an input (expected or not) and observe outputs and tock.
      i. Apply an input. Choose $?i \in I$ and
         - Create a fresh state q_i; $Q' := Q' \cup \{q_i\}$; $T' := T' \cup \{(q', ?i, q_i)\}$.
         - If $P \stackrel{?i}{\Longrightarrow}$ then $Q_{aux} := Q_{aux} \cup \{(P \textbf{ after } ?i, q_i)\}$; $F := F \cup \{q_i\}$.
      ii. Observe outputs and tock. For each $a \in O \cup \{\ominus\}$ do
         - Create a fresh state q_a; $Q' := Q' \cup \{q_a\}$; $T' := T' \cup \{(q', a, q_a)\}$.
         - If $P \stackrel{a}{\Longrightarrow}$ then $Q_{aux} := Q_{aux} \cup \{(P \textbf{ after } a, q_a)\}$; $F := F \cup \{q_a\}$.

Figure 12: Test derivation I: the $\preceq_{tr}$ relation

we consider all the non-deterministic choices in the algorithm (the choice in Step 2 between stop and continue testing and the choice of the input in Step 2(b)i) then we obtain a test suite. Note that in Steps 2(b)i and ii we continue testing (third clause) only if we have chosen an expected input or we are dealing with an output or tock that the specification can perform from the reached state after partially traversing it.

Definition 24. *Let q be a tockLTS. We denote by $\mathcal{T}\texttt{est}_{\preceq_{tr}}(q)$ the smallest set including all the tests that can be derived by applying the algorithm given in Figure 12 to q.*

Example 13. *Let p be our running example (see Example 2) and consider the tests t_1 and t_2 depicted, respectively, in Figure 9 (left) and (right). We have that $t_1 \in \mathcal{T}\texttt{est}_{\preceq_{tr}}(p)$ while $t_2 \notin \mathcal{T}\texttt{est}_{\preceq_{tr}}(p)$. In the latter case, note that state 8 should be a fail state. Note that the transition departing from state 4 and labelled by ?o could be produced by our derivation algorithm because it reaches a fail state (this transition is not available in p after performing ?s).*

Next we study the computational complexity of our algorithm. Therefore, we will restrict to *finite* tockLTSs, that is, we will assume that we have finite sets of inputs, outputs, states and transitions. Note that, as previously discussed, the derivation algorithm returns a test case but if the algorithm is repeated then there is no bound on the number of test cases that can be produced. Therefore, we consider the computational complexity of producing a single test case. In addition, observe that the time taken to produce a test case will inevitably depend on the size of the test case and, in particular, it is only possible to place an upper bound on the time taken if we either fix or bound the size of the test case. As a result, we explore the complexity of producing a test case that has n inputs. Note that this is equivalent to applying Step 2(b) of the test case derivation algorithm a total of n times.

Theorem 12. *Let $p = (Q, q_0, I, O, T)$ be a tockLTS. The space needed to store a test case with n inputs and generated by the algorithm given in Figure 12 applied to p is in $O(n \cdot |O| \cdot |Q|)$.*

Proof. Each application of Step 2(b) generates $|O|+2$ new states

in the test case and as many transitions. In addition, the auxiliary storage space (that is, the set Q_{aux}) increases by at most $|O|+2$ pairs. Note that each pair added to Q_{aux} has a single state of the test case and also a subset of Q. As a result, the auxiliary space added in each iteration is in $O(|O| \cdot |Q|)$. Therefore, we have that, the space needed to compute a test case with n inputs, including both the space needed for the test case (states and transitions) and auxiliary space needed to construct it, is in $O(n \cdot |O| + n \cdot |O| + n \cdot |O| \cdot |Q|)$, that is, in $O(n \cdot |O| \cdot |Q|)$.

Theorem 13. *Let $p = (Q, q_0, I, O, T)$ be a tockLTS. The time needed to produce a test case with n inputs and generated by the algorithm given in Figure 12 applied to p is in $O(n \cdot |O| \cdot |Q| \cdot (|Q| + |T|))$.*

Proof. Each application of Step 1 can be done in constant time: we simply need to choose any element of a set. The first item of Step 2(b)i also takes constant time. Concerning the second item of this step, in order to decide whether $P \stackrel{?i}{\Longrightarrow}$, it is enough to traverse the multi-graph induced by p from each element of P. Since traversing a graph, either using a depth-first or a breadth-first strategy, can be done in $O(|E| + |V|)$, where E is the set of edges and V is the set of vertex, and $P \subseteq Q$, then this operation is in $O(|Q| \cdot (|Q| + |T|))$. Note that this operation also computes the states belonging to P **after** $?i$ that are needed in the second part of the item. Using a similar reasoning, we have that Step 2(b)ii can be done in time $O(|O| \cdot |Q| \cdot (|Q| + |T|))$ because we have to repeat the previous process, in the worst case, for all the elements in $O \cup \{\ominus\}$. Therefore, we have that, in the worst case, the time needed to compute a test case with n inputs is in $O(n \cdot (|Q| \cdot (|Q| + |T|) + (|O| \cdot |Q| \cdot (|Q| + |T|))))$, that is, in $O(n \cdot |O| \cdot |Q| \cdot (|Q| + |T|))$.

The final part of this section consists of proving that the derived test suites are indeed complete with respect to the $\preceq_{tr}$ implementation relation. First, we provide two auxiliary results.

Proposition 10. *Let $q = (Q, q_0, I, O, T)$ be a tockLTS and $t = (Q', q'_0, I \cup O \cup \{\ominus\}, T', F) \in \mathcal{T}\texttt{est}_{\preceq_{tr}}(q)$ be a test. Let $\sigma \in (I \cup O \cup \{\ominus\})^*$ such that $t \stackrel{\sigma}{\Longrightarrow}$ and let $q' \in Q'$ be the unique state such that $t \stackrel{\sigma}{\Longrightarrow} q'$. We have that $q' \in F$ if and only if $\sigma \in \mathcal{T}\texttt{traces}(q)$.*

Proof. We will prove the result by induction on the length of σ. The base case, with σ being the empty sequence, is immediate because t can only reach the initial state (by construction, it belongs to F) and the empty sequence is always a trace of q.

Inductive hypothesis: the result holds if the sequence has length less than k ($k > 0$). Let us suppose that σ has length k. Therefore, $\sigma = \sigma_1 a$ and the result holds for σ_1. First, note that σ_1 must reach a state belonging to F because there are no outgoing transitions departing from states in $Q' \backslash F$. Therefore, by induction, $\sigma_1 \in \mathcal{T}\texttt{traces}(q)$. Depending on whether a is an allowed/unexpected input, an output or a tock, the transition of the test case labelled by a can be produced, respectively, by Step 2(b)i (if $a \in I$) and Step 2(b)ii (otherwise). In all cases it is straightforward to check that the reached state after a belongs to F if and only if a can be performed from the states belonging to q **after** σ_1. In other words, this holds if and only if $\sigma = \sigma_1 a \in \mathcal{T}\texttt{traces}(q)$.

Proposition 11. *Let I and O be countable disjoint sets of inputs and outputs respectively. Let $q \in$ TockLTS(I, O) and $\sigma \in \mathcal{T}\texttt{traces}(q)$. There exists a test case $t = (Q', q'_0, I \cup O \cup \{\ominus\}, T', F) \in \mathcal{T}\texttt{est}_{\preceq_{tr}}(q)$ such that $t \stackrel{\sigma}{\Longrightarrow} q'$ and $q' \in F$.*

Proof. We will prove the result by induction on the length of σ. The base case, with σ being the empty sequence, is immediate because it is enough to consider the test case built after applying the initialisation of the algorithm given in Figure 12, entering the loop, choosing the only existing pair of Q_{aux}, and choosing the option (a) in Step 2. This process produces an *empty* test, with one final state, and we have $q'_0 \stackrel{\epsilon}{\Longrightarrow} q'_0$

Inductive hypothesis: the result holds if the sequence has length less than k ($k > 0$). Let us suppose that σ has length k. Therefore, $\sigma = \sigma_1 a$ and the result holds for σ_1. Again, we have that the state of the test reached after performing σ_1 belongs to F. Let q_{σ_1} be this state. Note that at this stage we will have that $(q$ **after** $\sigma_1, q_{\sigma_1}) \in Q_{aux}$. In addition, since σ is a trace of q, we have $(q$ **after** $\sigma_1) \stackrel{a}{\Longrightarrow}$. If a is an input then we will apply Step 2(b)i; if a is an output or a tock, then we will apply Step 2(b)ii. In both cases, the reached state belongs to F. Independently of the way we complete this test, we have that the requested property holds.

Theorem 14 (Soundness of $\mathcal{T}\texttt{est}_{\preceq_{tr}}(q)$). *Let I and O be countable disjoint sets of inputs and outputs respectively. Let $p, q \in$ TockLTS(I, O) and $t \in \mathcal{T}\texttt{est}_{\preceq_{tr}}(q)$. If $p \pitchfork t$ fails then $p \preceq_{tr} q$ does not hold.*

Proof. If $p \pitchfork t$ fails then there exists a sequence $\sigma \in (I \cup O \cup \{\ominus\})^*$ reaching a non-final state of the automaton $p \pitchfork t$ from its initial state. First, by Lemma 1 we have that $\sigma \in \mathcal{T}\texttt{traces}(p)$. Second, since we reach a non-final state of the test case and by Proposition 10, we have that $\sigma \notin \mathcal{T}\texttt{traces}(q)$. Therefore, $\mathcal{T}\texttt{traces}(p) \not\subseteq \mathcal{T}\texttt{traces}(q)$ and we conclude that $p \preceq_{tr} q$ does not hold.

Example 14. *Consider again our running example p, the test case t_1 given in Figure 9 (left) and the system p_1 given in Figure 11 (top). In Example 13 we saw that $t_1 \in \mathcal{T}\texttt{est}_{\preceq_{tr}}(p)$ and in Example 12 we obtained that $p_1 \pitchfork t_1$ fails. The soundness of our framework allows us to state that $p_1 \preceq_{tr} p$ does not hold even without computing the traces of both processes.*

Theorem 15 (Exhaustiveness of $\mathcal{T}\texttt{est}_{\preceq_{tr}}(q)$). *Let I and O be countable disjoint sets of inputs and outputs respectively. Let $p, q \in$ TockLTS(I, O). If $p \preceq_{tr} q$ does not hold then there exists $t \in \mathcal{T}\texttt{est}_{\preceq_{tr}}(q)$ such that $p \pitchfork t$ fails.*

Proof. If $p \preceq_{tr} q$ does not hold then there exists $\sigma \in (I \cup O \cup \{\ominus\})^*$ such that $\sigma \in \mathcal{T}\texttt{traces}(p)$ but $\sigma \notin \mathcal{T}\texttt{traces}(q)$. Let $\sigma' \in \mathcal{T}\texttt{traces}(q)$ be the longest sequence such that there exists σ'' such that $\sigma = \sigma'\sigma''$. Note that σ' might be empty but σ'' cannot be empty. So, $\sigma'' = a\sigma'''$ and $\sigma' a \notin \mathcal{T}\texttt{traces}(q)$.

By Proposition 11 we can build a test case whose set of traces includes σ'. If we extend this test case to consider a, we will have that a reaches a non-final state of the test, denoting the fail of the application of the test case if this state is reached. Obviously, this state is reached because $\sigma' a$ is a trace of t and a trace of p and applying Lemma 1, it is also a trace of $p \pitchfork t$. Therefore, p fails t, as requested.

Corollary 1 (Completeness of $\mathcal{T}\mathtt{est}_{\preceq_{tr}}(q)$). *Let I and O be countable disjoint sets of inputs and outputs respectively. Let $p, q \in TockLTS(I, O)$. We have that $p \preceq_{tr} q$ if and only if for all $t \in \mathcal{T}\mathtt{est}_{\preceq_{tr}}(q)$ we have that $p \pitchfork t$ does not fail.*

8.2. Complete test suites with respect to $\preceq_{\mathcal{T}}$

In order to capture refusals, we have to increase the *testing power* of tests. We slightly modify the notion of test case given in Definition 22 by including two new types of *action*. First, we will add the capability to test the existence of a refusal. Second, we will be able to test refusals such that its performance, followed by a tock action, will detect a faulty behaviour.

Definition 25 (test). *Let I and O be countable disjoint sets of inputs and outputs respectively. An I/O-*test case with refusals *(or simply test case when I and O can be inferred from the context and it is clear that we are taking into account refusals) is an acyclic and deterministic automaton $p = (Q, q_0, I \cup O \cup \mathcal{R}(I \cup O) \cup \mathcal{P}(\mathcal{R}(I \cup O)) \cup \{\ominus\}, T, F)$ such that for all $q \in F$ we have that either there are no outgoing transitions from q or the following three restrictions hold:*

- *Observe any possible output and passing of time. For each $a \in O \cup \{\ominus\}$ there exists $q_a \in Q$ such that $(q, a, q_a) \in T$.*
- *Continue with at most one input or refusal. There exists at most one $a \in I \cup \mathcal{R}(I \cup O)$ such that there exists $q_a \in Q$ such that one of the following hold.*
  - *$a \in I$ and $(q, a, q_a) \in T$, or*
  - *$a \in \mathcal{R}(I \cup O)$, $(q, a, q_a) \in T$, $q_a \in F$ and there exists $q'_a \in Q$ such that $(q_a, \ominus, q'_a) \in T$. This is the only transition departing from q_a.*
- *Check unexpected refusals. There exist $\mathcal{A} \in \mathcal{P}(\mathcal{R}(I \cup O))$, $q_{\mathcal{A}} \in F$ and $q'_{\mathcal{A}} \in Q \backslash F$ such that $(q, \mathcal{A}, q_{\mathcal{A}}), (q_{\mathcal{A}}, \ominus, q'_{\mathcal{A}}) \in T$. The last one is the only transition departing from $q_{\mathcal{A}}$.*

In addition, we require that non-final states, denoting that the testing process has failed, are deadlocked, that is, if $q \notin F$ then there does not exist $a \in I \cup O \cup \mathcal{R}(I \cup O) \cup \mathcal{P}(\mathcal{R}(I \cup O)) \cup \{\ominus\}$ and $q' \in Q$ such that $(q, a, q') \in T$.

We use $\mathcal{T}(I, O)_{\mathcal{R}}$ to denote the set of I/O-tests with refusals.

Similarly to our previous notion of test, tests with refusals will have ✓ and × states and states representing the potential application of an input $?i$ to the SUT. In this case, this application will be again paired with the potential to observe any output and the passing of time. In addition, tests will be able to observe that the SUT cannot perform a certain set of actions. We will use this option with two goals: check that the SUT does not have unexpected refusals and continue testing after observing an allowed refusal. In the former case, we use a unique transition, labelled by a set belonging to $\mathcal{P}(\mathcal{R}(I \cup O))$, to include all the refusals that the SUT should not observe. In both cases, in order to observe a refusal in testing, we have to make sure that it is followed by the performance of a tock action. An obvious corollary of the previous definition is that any sequence of transitions performed by a test case can have, at most, one transition labelled by a set in $\mathcal{P}(\mathcal{R}(I \cup O))$. This fact will be explicitly used in the proof of some of the results in this section.

Next, we define the application of a test case with refusals to an SUT. Again, we will say that the application of the test case has failed if a non-final state of the test case can be reached.

Definition 26 (test application). *Let I and O be countable disjoint sets of inputs and outputs respectively. Let $p = (Q, q_0, I, O, T)$ be a tockLTS and $t = (Q', q'_0, I \cup O \cup \mathcal{R}(I \cup O) \cup \mathcal{P}(\mathcal{R}(I \cup O)) \cup \{\ominus\}, T', F)$ be an I/O-test with refusals. We define the application of the test case t to the system p, denoted by $p \pitchfork t$, as the automaton $(Q \times Q', (q_0, q'_0), I \cup O \cup \mathcal{R}(I \cup O) \cup \mathcal{P}(\mathcal{R}(I \cup O)) \cup \{\ominus, \tau\}, T'', Q \times F)$ where T'' is the smallest set of transitions fulfilling the following rules*

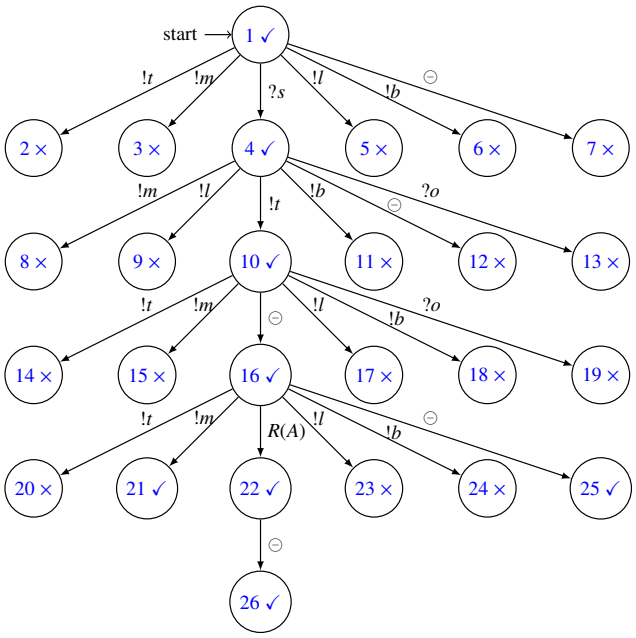
- *If $(q_1, \tau, q_2) \in T$ then for all $q' \in Q'$ we have $((q_1, q'), \tau, (q_2, q')) \in T''$.*
- *For each $a \in I \cup O \cup \{\ominus\}$, if $(q_1, a, q_2) \in T$ and $(q'_1, a, q'_2) \in T'$ then $((q_1, q'_1), a, (q_2, q'_2)) \in T''$.*
- *If $q_1 \xrightarrow{R(A)} q_1$, with $q_1 \in Q$, and $(q'_1, R(A), q'_2) \in T'$ then $((q_1, q'_1), R(A), (q_1, q'_2)) \in T''$.*
- *If $q_1 \xrightarrow{R(A)} q_1$, with $q_1 \in Q$, $R(A) \in \mathcal{A}$ and $(q'_1, \mathcal{A}, q'_2) \in T'$, then $((q_1, q'_1), \mathcal{A}, (q_1, q'_2)) \in T''$.*

We say that the application of t to p has failed *if there is a sequence of transitions belonging to T'' departing from (q_0, q'_0) and reaching a state belonging to $Q \times (Q' \backslash F)$; otherwise, we say that the application was* successful.

The differences with respect to Definition 23 appear in the new third and fourth clauses. If the test case offers a refusal and the process is able to refuse that set, then they will synchronise. Taking into account the structure of tests, where the offering of a refusal is always followed by a tock, the next performed action will be $\ominus$. If the test case offers a set of refusals such that the process is able to refuse at least one of these refusals, then the process and the test case simultaneously evolve.

The next result is an adaption of Lemma 1 and its proof is immediate.

Lemma 2. *Let I and O be countable disjoint sets of inputs and outputs respectively. Let $p \in TockLTS(I, O)$, t be an I/O-test with refusals and $\sigma \in (I \cup O \cup \mathcal{R}(I \cup O) \cup \mathcal{P}(\mathcal{R}(I \cup O)) \cup \{\ominus\})^*$. We have that $p \pitchfork t \overset{\sigma}{\Longrightarrow}$ if and only if $t \overset{\sigma}{\Longrightarrow}$ and there exists $\sigma' \in (I \cup O \cup \mathcal{R}(I \cup O) \cup \{\ominus\})^*$, that can be formed from σ by replacing each occurrence of each set of refusals $\mathcal{A} \in \mathcal{P}(\mathcal{R}(I \cup O))$ by one of its elements $R(A) \in \mathcal{A}$, such that $p \overset{\sigma'}{\Longrightarrow}$.*



where $A = \{!t, !m, !l, !b, ?s, ?o, ?f\}$ and unexpected refusals have been omitted.

Figure 13: I/O-test with refusals

Moreover, we have the following results concerning the membership of σ to $\mathcal{TR}\texttt{traces}(p)$.

1. *If either the last or the penultimate action of σ belongs to $\mathcal{P}(\mathcal{R}(I \cup O))$, that is, $\sigma = \sigma''\mathcal{A}$ or $\sigma = \sigma''\mathcal{A}\ominus$ then $\sigma'' \in \mathcal{TR}\texttt{traces}(p)$. In the latter case, there exists $R(B) \in \mathcal{A}$ such that $\sigma''R(B)\ominus \in \mathcal{TR}\texttt{traces}(p)$.*
2. *If the last action of σ belongs to $\mathcal{R}(I \cup O)$, that is, $\sigma = \sigma''R(A)$ then $\sigma'' \in \mathcal{TR}\texttt{traces}(p)$.*
3. *Otherwise, $\sigma = \sigma'$ and $\sigma \in \mathcal{TR}\texttt{traces}(p)$.*

Next we present an algorithm to derive tests from a specification such that an SUT conforms to the specification with respect to $\preceq_{\mathcal{T}}$ if and only if the SUT successfully passes all the tests of the suite. The algorithm is a variation of the one given in Figure 12 to take into account refusals. Again, the application of the algorithm to a specification produces a single test. If we consider all the non-deterministic choices in the algorithm (the choice in Step 2 between the options (a) and (b) and the choice of the input or refusal in Step 2(b)i) then we obtain a test suite. The new algorithm is given in Figure 14.

Definition 27. *Let q be a tockLTS. We denote by $\mathcal{T}\texttt{est}_{\preceq_{\mathcal{T}}}(q)$ the smallest set including all the tests that can be derived by applying the algorithm given in Figure 14 to q.*

Example 15. *Let p be our running example (see Example 2) and t be the test case given in Figure 13. We have that $t \in \mathcal{T}\texttt{est}_{\preceq_{\mathcal{T}}}(p)$.*

Next we study the computational complexity of the new algorithm. Again, we will assume that we have finite sets of inputs, outputs, states and transitions. This time, we will use as *unit of measure* the number of inputs and expected refusals appearing in the test case (this number is equal to the number of non-trivial iterations of the main loop).

Theorem 16. *Let $p = (Q, q_0, I, O, T)$ be a tockLTS. The space needed to store a test case with a total of n inputs and expected refusals and generated by the algorithm given in Figure 14 applied to p is in $O(n \cdot (2^{|I|} + |O| \cdot |Q|))$.*

Proof. Each application of Step 2(b) generates at most, $|O| + 5$ new states in the test case and as many transitions. The difference with respect to the proof of Theorem 12 is that we now have two additional states and transitions for the set of unexpected refusals and if we apply a refusal instead of an input then we produce two states and transitions instead of just one. Similarly, the set Q_{aux} increases by at most $|O|+2$ pairs, storing again each pair belonging to Q_{aux} a single state and a subset of Q.

However, we have to take into account that a transition of the test case does not have a *small* amount of information. In the previous case, the information stored in a transition was an action. However, in the new framework each iteration adds a transition labelled by a set of unexpected refusals. Therefore, in the worst case, we need space in $O(2^{|I|})$ to store these transitions. Note that although refusals are subsets of $I \cup O$, we can omit outputs because they can be always added to any refusal (see Proposition 5). Therefore, we have that the space needed to compute a test case with a total of n inputs and/or expected refusals, including both the space needed for the test case and auxiliary space needed to construct it, is in $O(n \cdot |O| + n \cdot (|O| + 2^{|I|}) + n \cdot (|O| \cdot |Q|))$, that is, in $O(n \cdot (2^{|I|} + |O| \cdot |Q|))$.

Theorem 17. *Let $p = (Q, q_0, I, O, T)$ be a tockLTS. The time needed to produce a test case with n inputs and expected refusals and generated by the algorithm given in Figure 14 applied to p is in $O(n \cdot |Q| \cdot (|Q| + |T|) \cdot (|O| + 2^{|I|}))$.*

Proof. The main difference with respect to the proof of Theorem 13 is that each iteration of the main loop needs to compute whether each refusal set is expected or not for the set of states that we are processing: unexpected refusals are always used (Step 2(b)iii) while expected refusals can be used in Step 2(b)i.B. Since we can omit outputs, as explained in the proof of Theorem 16, each iteration needs additional processing in $O(2^{|I|})$ with respect to an iteration of the previous algorithm. Therefore, we have that, in the worst case, the time needed to compute a test case with n inputs and/or expected refusals is in $O(n \cdot |Q| \cdot (|Q| + |T|) \cdot (|O| + 2^{|I|}))$.

The final part of this section involves proving that the derived test suites are indeed complete with respect to the $\preceq_{\mathcal{T}}$ implementation relation. We will follow a similar methodology to the one previously used for $\preceq_{tr}$ and start with some auxiliary results.

Proposition 12. *Let $q = (Q, q_0, I, O, T)$ be a tockLTS and $t = (Q', q'_0, I \cup O \cup \mathcal{R}(I \cup O) \cup \mathcal{P}(\mathcal{R}(I \cup O)) \cup \{\ominus\}, T', F)$ be a test. Let $\sigma \in (I \cup O \cup \mathcal{R}(I \cup O) \cup \mathcal{P}(\mathcal{R}(I \cup O)) \cup \{\ominus\})^*$ be such that $t \stackrel{\sigma}{\Longrightarrow}$ and let $q' \in Q'$ be the unique state such that $t \stackrel{\sigma}{\Longrightarrow} q'$. We have that $q' \in F$ if and only if one of the following hold*

`Input`: A specification $q = (Q, q_0, I, O, T)$.
`Output`: A test case with refusals $t = (Q', q'_0, I \cup O \cup \mathcal{R}(I \cup O)) \cup \mathcal{P}(\mathcal{R}(I \cup O)) \cup \{\ominus\}, T', F)$.

`Initialization`

Set a fresh state q'_0 as initial state; $Q', F := \{q'_0\}$; $T' := \emptyset$; $Q_{aux} := \{(\{q_0\}, q'_0)\}$.

`Loop`

While $Q_{aux} \neq \emptyset$ do

1. Choose $(P, q') \in Q_{aux}$; $Q_{aux} := Q_{aux} \backslash \{(P, q')\}$.
2. Select one of the following alternatives.
   (a) Successful termination (by construction, $q' \in F$).
   (b) Continue testing: apply an input (expected or not) or refusal, observe outputs and tock and look for unexpected refusals.
      i. Apply an input or a refusal. Choose one of the following options.
         A. Apply an input. Choose $?i \in I$ and
            - Create a fresh state q_i; $Q' := Q' \cup \{q_i\}$; $T' := T' \cup \{(q', i, q_i)\}$.
            - If $P \stackrel{?i}{\Longrightarrow}$ then $Q_{aux} := Q_{aux} \cup \{(P \textbf{ after } ?i, q_i)\}$; $F := F \cup \{q_i\}$.
         B. Apply an expected refusal. Let $\mathcal{R}_P = \{R(A) \in \mathcal{R}(I \cup O) | P \stackrel{R(A)}{\Longrightarrow}\}$. If $\mathcal{R}_P \neq \emptyset$ then choose $R(A) \in \mathcal{R}_P$ and
            - Create two fresh states q_A and q'_A; $Q' := Q' \cup \{q_A, q'_A\}$; $T' := T' \cup \{(q', R(A), q_A), (q_A, \ominus, q'_A)\}$.
            - $Q_{aux} := Q_{aux} \cup \{(P \textbf{ after } \ominus, q_{A'})\}$; $F := F \cup \{q_A, q'_A\}$.
      ii. Observe outputs and tock. For each $a \in O \cup \{\ominus\}$ do
         - Create a fresh state q_a; $Q' := Q' \cup \{q_a\}$; $T' := T' \cup \{(q', a, q_a)\}$.
         - If $P \stackrel{a}{\Longrightarrow}$ then $Q_{aux} := Q_{aux} \cup \{(P \textbf{ after } a, q_a)\}$; $F := F \cup \{q_a\}$.
      iii. Look for unexpected refusals. Let $\mathcal{A} = \{R(A) \in \mathcal{R}(I \cup O) | P \stackrel{R(A)}{\not\Longrightarrow}\}$ and
         - Create two fresh states $q_{\mathcal{A}}$ and $q'_{\mathcal{A}}$. $Q' := Q' \cup \{q_{\mathcal{A}}, q'_{\mathcal{A}}\}$; $T' := T' \cup \{(q', \mathcal{A}, q_{\mathcal{A}}), (q_{\mathcal{A}}, \ominus, q'_{\mathcal{A}})\}$.
         - $F := F \cup \{q_{\mathcal{A}}\}$.

Figure 14: Test derivation II: the $\preceq_{\mathcal{T}}$ relation

- $\sigma \in \mathcal{TR}\texttt{traces}(q)$, *or*
- *there exists* $R(A) \in \mathcal{R}(I \cup O)$ *such that* $\sigma = \sigma_1 R(A)$ *and* $\sigma_1 \in \mathcal{TR}\texttt{traces}(q)$, *or*
- *there exists* $\mathcal{A} \in \mathcal{P}(\mathcal{R}(I \cup O))$ *such that* $\sigma = \sigma_1 \mathcal{A}$ *and* $\sigma_1 \in \mathcal{TR}\texttt{traces}(q)$. *In this case, for all* $R(A) \in \mathcal{A}$ *we have that* $\sigma_1 R(A) \ominus \notin \mathcal{TR}\texttt{traces}(q)$.

Proof. First, note that the last two actions of σ cannot be $\mathcal{A}\ominus$ because we would reach a non-final state of the test. We will prove the result by induction on the length of σ. The base case, with σ being the empty sequence, is immediate because t can only reach the initial state (by construction, it belongs to F) and the empty sequence is always a trace of q.

Inductive hypothesis: the result holds if the sequence has length less than k ($k > 0$). Let us suppose that σ has length k. Therefore, $\sigma = \sigma_1 a$ and the result holds for σ_1. First, note that σ_1 must reach a state belonging to F because there are no outgoing transitions departing from states in $Q' \backslash F$. We consider three cases.

If $a \in I \cup O \cup \{\ominus\}$, taking into account the observation that we made in the beginning of the proof, then we reason as in the proof of Proposition 10. Depending on whether a is an allowed/unexpected input, an output or a tock, the transition of the test labelled by a can be produced, respectively, by Step 2(b)iA (if $a \in I$) and Step 2(b)ii (otherwise). In all cases it is straightforward to check that the reached state after a belongs to F if and only if a can be performed from the states belonging to q **after** σ_1. In other words, this holds if and only if $\sigma = \sigma_1 a \in \mathcal{T}\texttt{traces}(q)$.

If $a \in \mathcal{R}(I \cup O)$ then σ' must finish with an action belonging to $a \in I \cup O \cup \{\ominus\}$ because there cannot be two consecutive occurrences of single refusals (because each single refusal must be followed by a tock). Therefore, $\sigma_1 \in \mathcal{T}\texttt{traces}(q)$.

If $a \in \mathcal{P}(\mathcal{R}(I \cup O))$, again, we must have that the last action of σ_1 belongs to $a \in I \cup O \cup \{\ominus\}$ (it cannot be a single refusal because they are always followed by tock and there can be at most one occurrence of an action belonging to $\mathcal{P}(\mathcal{R}(I \cup O))$ in a sequence performed by a test). Again, we conclude $\sigma_1 \in \mathcal{T}\texttt{traces}(q)$. Moreover, by construction we trivially have the second part of the result, that is, for all $R(A) \in \mathcal{A}$ we have that $\sigma_1 R(A) \ominus \notin \mathcal{TR}\texttt{traces}(q)$.

The proof of the next result is an obvious adaption of the proof of Proposition 11 and, therefore, we omit it. In particular, note that the sequences considered in the result do not contain elements of $\mathcal{P}(\mathcal{R}(I \cup O))$ because they are (timed refusal) traces of a process.

Proposition 13. *Let I and O be countable disjoint sets of inputs and outputs respectively. Let $q \in TockLTS(I, O)$ and*

$\sigma \in \mathcal{TR}\mathtt{traces}(q)$. *There exists a test case* $t = (Q', q'_0, I \cup O \cup \mathcal{R}(I \cup O)) \cup \mathcal{P}(\mathcal{R}(I \cup O)) \cup \{\ominus\}, T', F) \in \mathcal{T}\mathtt{est}_{\leq_{tr}}(q)$ *such that* $t \overset{\sigma}{\Longrightarrow} q'$ *and* $q' \in F$.

Theorem 18 (Soundness of $\mathcal{T}\mathtt{est}_{\leq_{\mathcal{T}}}(q)$). *Let I and O be countable disjoint sets of inputs and outputs respectively. Let $p, q \in$ TockLTS(I, O) and $t \in \mathcal{T}\mathtt{est}_{\leq_{\mathcal{T}}}(q)$. If $p \pitchfork t$ fails then $p \leq_{\mathcal{T}} q$ does not hold.*

Proof. If $p \pitchfork t$ fails then there exists a sequence $\sigma \in (I \cup O \cup \mathcal{R}(I \cup O) \cup \mathcal{P}(\mathcal{R}(I \cup O)) \cup \{\ominus\})^*$ reaching a non-final state of the automaton $p \pitchfork t$ from its initial state. We have two possibilities: $\sigma \in (I \cup O \cup \mathcal{R}(I \cup O)) \cup \{\ominus\})^*$ or $\sigma = \sigma'\mathcal{A}\ominus$, with $\sigma' \in (I \cup O \cup \mathcal{R}(I \cup O)) \cup \{\ominus\})^*$ and $\mathcal{A} \in \mathcal{P}(\mathcal{R}(I \cup O))$. Note that σ cannot end with an action belonging to $\mathcal{R}(I \cup O)) \cup \mathcal{P}(\mathcal{R}(I \cup O))$ because transitions labelled by these actions reach final states.

Consider the first case, $\sigma \in (I \cup O \cup \mathcal{R}(I \cup O) \cup \{\ominus\})^*$. By Lemma 2 we have that $p \overset{\sigma}{\Longrightarrow}$ and since σ does not end either with a refusal or a set of refusals we obtain $\sigma \in \mathcal{TR}\mathtt{traces}(p)$. Second, since we reach a non-final state, we apply Proposition 12 and given the fact that only one of the three possibilities is viable (again, σ does not end with either an element belonging to $\mathcal{P}(\mathcal{R}(I \cup O)) \cup \mathcal{R}(I \cup O))$ we have that $\sigma \notin \mathcal{TR}\mathtt{traces}(q)$. Therefore, $\mathcal{TR}\mathtt{traces}(p) \not\subseteq \mathcal{TR}\mathtt{traces}(q)$ and we conclude that $p \leq_{\mathcal{T}} q$ does not hold.

In the second case we have $\sigma = \sigma'\mathcal{A}\ominus$, with $\sigma' \in (I \cup O \cup \mathcal{R}(I \cup O) \cup \{\ominus\})^*$ and $\mathcal{A} \in \mathcal{P}(\mathcal{R}(I \cup O))$. Applying Lemma 2 we have that there exists $R(B) \in \mathcal{A}$ such that $\sigma'' = \sigma' R(B)\ominus \in \mathcal{TR}\mathtt{traces}(p)$. Applying Proposition 12 we have that $\sigma' \in \mathcal{TR}\mathtt{traces}(q)$ (note that σ' reaches a final state because this sequence can be extended). Taking into account the definition of $\mathcal{A}$ in Step 2(b)iii of the derivation algorithm, this set contains the refusals $R(A)$ such that q cannot perform $R(A)\ominus$ after performing σ'. In particular, it cannot perform the sequence $R(B)\ominus$ and, therefore, $\sigma'' \notin \mathcal{TR}\mathtt{traces}(q)$. Thus, $\mathcal{TR}\mathtt{traces}(p) \not\subseteq \mathcal{TR}\mathtt{traces}(q)$ and we conclude that $p \leq_{\mathcal{T}} q$ does not hold.

Theorem 19 (Exhaustiveness of $\mathcal{T}\mathtt{est}_{\leq_{\mathcal{T}}}(q)$). *Let I and O be countable disjoint sets of inputs and outputs respectively. Let $p, q \in$ TockLTS(I, O). If $p \leq_{\mathcal{T}} q$ does not hold then there exists $t \in \mathcal{T}\mathtt{est}_{\leq_{\mathcal{T}}}(q)$ such that $p \pitchfork t$ fails.*

Proof. If $p \leq_{\mathcal{T}} q$ does not hold then there exists $\sigma \in (I \cup O \cup \mathcal{R}(I \cup O) \cup \{\ominus\})^*$ such that $\sigma \in \mathcal{TR}\mathtt{traces}(p)$ but $\sigma \notin \mathcal{TR}\mathtt{traces}(q)$. Let $\sigma' \in \mathcal{TR}\mathtt{traces}(q)$ be the longest sequence such that there exists σ'' such that $\sigma = \sigma'\sigma''$. Note that σ' might be empty but σ'' cannot be empty. So, $\sigma'' = a\sigma'''$ and $\sigma' a \notin \mathcal{TR}\mathtt{traces}(q)$. Now, we distinguish two cases.

If $a \notin \mathcal{R}(I \cup O)$ then we proceed as in the proof of Theorem 15. By Proposition 13 we can build a test case with trace σ'. If we extend this test case to consider a, then we will have that a reaches a non-final state of the test, denoting the fail of the application of the test case if this state is reached. Obviously, this state is reached in $p \pitchfork t$ because $\sigma' a$ is a trace of t and a trace of p and applying Lemma 2, it is also a trace of $p \pitchfork t$. Therefore, p fails t, as requested.

If $a \in \mathcal{R}(I \cup O)$, that is, $a = R(B)$ for a certain $B \subseteq I \cup O$, then we know that a is followed by a tock action. Therefore, we have $\sigma = \sigma' R(B) \ominus \sigma'''$, with $\sigma' \in \mathcal{TR}\mathtt{traces}(q)$, $\sigma' R(B)\ominus \notin \mathcal{TR}\mathtt{traces}(q)$ and $\sigma' R(B)\ominus \in \mathcal{TR}\mathtt{traces}(p)$. In other words, we have $q \overset{\sigma'}{\Longrightarrow}$ but $q \overset{\sigma' R(B)\ominus}{\Longrightarrow\!\!\!\!\!\!/\;\;}$. Again, by Proposition 13 we can build a test case with trace σ'. We will have that $R(B)$ will be a member of the set of refusals $\mathcal{A}$ built after applying step 2(b)iii to the state that we reach in the test case t after performing σ'. If we apply this test case to p, taking into account that $p \overset{\sigma' R(B)\ominus}{\Longrightarrow}$, we have that p fails t, as requested.

Corollary 2 (Completeness of $\mathcal{T}\mathtt{est}_{\leq_{\mathcal{T}}}(q)$). *Let I and O be countable disjoint sets of inputs and outputs respectively. Let $p, q \in$ TockLTS(I, O). We have that $p \leq_{\mathcal{T}} q$ if and only if for all $t \in \mathcal{T}\mathtt{est}_{\leq_{\mathcal{T}}}(q)$ we have that $p \pitchfork t$ does not fail.*

9. Conclusions and future work

There has been significant interest in testing from formal models since this brings the potential for automated systematic testing. In order to test from a formal model one requires an implementation relation that says what it means for the system under test (SUT) to be a correct implementation of the specification. This paper considered cyclic models, in which behaviours are of the form of sequences of observable actions separated by discrete time intervals. The work was motivated by the use of cyclic simulators in a number of areas, including robotic systems.

Although many implementation relations are variants of the well known ioco implementation relation, ioco and its timed versions were not suitable for cyclic models. As a result, there was a need to define novel implementation relations that take into account the discrete nature of time in cyclic models. We defined two types of implementation relation that differ in whether or not it is possible to observe the situation in which a model can refuse a set of actions. There were three variants of each type of implementation relation, with these varying in how unspecified inputs are treated.

We introduced two alternative characterisations of timed refusal trace inclusion. First, we showed how one can define an automaton whose language is exactly the set of timed refusal traces of a model; this allows one to express correctness in terms of formal language containment for automata. We also showed how one can define timed refusal trace inclusion in terms of the observations that can be made by an observer interacting with processes (the specification and SUT); this demonstrates that timed refusal trace inclusion corresponds to the notion of observation for our models.

Finally, we introduced two testing frameworks that appropriately capture our main implementation relations. Specifically, we gave test derivation algorithms such that a process conforms to a specification if and only if the process successfully passes all the tests that can be derived from the specification.

There are several possible lines of future work. Regarding the testing of deployed robots, as opposed to testing in a simulation, we would like to further explore the role of the *envir-*

onment in our framework. Although one can test within a simulation (using a *model* of the environment), in practice the real environment will not behave like the model. As a result, it is necessary to address this 'reality gap' when testing the actual deployed robot. We believe that it will be possible to map the tests produced using the proposed algorithms to tests that can be used in testing a deployed robot. However, it is likely that there will also be a need to have additional tests that explore the reality gap, possibly based on a search-based approach that aims to maximise observed differences between the simulation of the environment and the actual environment.

There are also several possible lines of future work associated with the formalism used, test theory, and test generation algorithms. It would be interesting to explore conditions under which models can be expressed as Finite State Machines, allowing the use of the associated test generation algorithms. More generally, there should also be scope to introduce additional test generation algorithms, possibly including algorithms that take into account fault models that describe the faults that might occur. Sometimes it is difficult to interact with the systems in which we are interested. In such situations, it is necessary to use a more *passive* testing approach. An approach might build on our previous work that considers the role of asynchronous communications [22, 32, 33]. There is the potential to enrich models to include, for example, probabilities or continuous variables (i.e. hybrid systems) and, again, our previous work [23] will be a starting point. Finally, we plan to carry out case studies with robotic systems.

References

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2017.

[3] R. V. Binder, B. Legeard, and A. Kramer. Model-based testing: where does it stand? *Communications of the ACM*, 58(2):52–56, 2015.

[4] S. Bonfanti, A. Gargantini, and A. Mashkoor. A systematic literature review of the use of formal methods in medical software systems. *Journal of Software: Evolution and Process*, 30(5):e1943, 2018.

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

[6] W. Bożejko and G. Bocewicz, editors. *Modelling and Performance Analysis of Cyclic Systems*. Springer, 2019.

[7] L. Brandán Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In *4th Int. Workshop on Formal Approaches to Testing of Software, FATES'04, LNCS 3395*, pages 64–78. Springer, 2004.

[8] A. Cavalcanti, J. Baxter, R. M. Hierons, and R. Lefticaru. Testing robots using CSP. In *13th Int. Conf. on Tests and Proofs, TAP'19, LNCS 11823*, pages 21–38. Springer, 2019.

[9] A. Cavalcanti, P. Ribeiro, A. Miyazawa, A. Sampaio, M. Conserva Filho, and A. Didier. RoboSim Reference Manual. Technical report, University of York, 2019.

[10] A. Cavalcanti, A. Sampaio, A. Miyazawa, P. Ribeiro, M. S. Conserva Filho, A. Didier, W. Li, and J. Timmis. Verified simulation for robotics. *Science of Computer Programming*, 174:1–37, 2019.

[11] A. R. Cavalli, T. Higashino, and M. Núñez. A survey on formal active and passive testing with applications to the cloud. *Annales of Telecommunications*, 70(3-4):85–93, 2015.

[12] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[13] J. Fitzgerald, J. Bicarregui, P. G. Larsen, and J. Woodcock. Industrial deployment of formal methods: Trends and challenges. In A. Romanovsky and M. Thomas, editors, *Industrial Deployment of System Engineering Methods*, pages 123–143. Springer, 2013.

[14] R. van Glabbeek. The linear time-branching time spectrum II. The semantics of sequential processes with silent moves. In *4th Int. Conf. on Concurrency Theory, CONCUR'93, LNCS 715*, pages 66–81. Springer, 1993.

[15] R. van Glabbeek. The linear time-branching time spectrum I. The semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of process algebra*, chapter 1. North Holland, 2001.

[16] D. Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[17] L. Heerink and J. Tretmans. Refusal testing for classes of transition systems with inputs and outputs. In *19th Joint Int. Conf. on Protocol Specification, Testing, and Verification and Formal Description Techniques, FORTE/PSTV'97*, pages 23–38. Chapman & Hall, 1997.

[18] R. M. Hierons. Testing from partial finite state machines without harmonised traces. *IEEE Transactions on Software Engineering*, 43(11):1033–1043, 2017.

[19] R. M. Hierons. FSM quasi-equivalence testing via reduction and observing absences. *Science of Computer Programming*, 177:1–18, 2019.

[20] R. M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A.J.H Simons, S. Vilkomir, M.R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):9:1–9:76, 2009.

[21] R. M. Hierons, J.P. Bowen, and M. Harman, editors. *Formal Methods and Testing, LNCS 4949*. Springer, 2008.

[22] R. M. Hierons, M. G. Merayo, and M. Núñez. An extended framework for passive asynchronous testing. *Journal of Logical and Algebraic Methods in Programming*, 86(1):408–424, 2017.

[23] R. M. Hierons and M. Núñez. Implementation relations and probabilistic schedulers in the distributed test architecture. *Journal of Systems and Software*, 132:319–335, 2017.

[24] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[25] World robotics 2018. International Federation of Robotics. Statistical department, 2018.

[26] ISO/IEC JTCI/SC21/WG7, ITU-T SG 10/Q.8. Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. ISO – ITU-T, 1996.

[27] M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[28] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines: A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[29] R. Lefticaru, R. M. Hierons, and M. Núñez. An implementation relation for cyclic systems that uses refusals and discrete time. In *17th Int. Conf. on Software Engineering and Formal Methods, SEFM'19, LNCS 11724*, pages 393–409. Springer, 2019.

[30] R. Marinescu, C. Seceleanu, H. Le Guen, and P. Pettersson. *A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs*, volume 98 of *Advances in Computers*, chapter 3, pages 89–140. Elsevier, 2015.

[31] G.H. Mealy. A method for synthesizing sequential circuits. *Bell System Techical Journal*, 34:1045–1079, 1955.

[32] M. G. Merayo, R. M. Hierons, and M. Núñez. Passive testing with asynchronous communications and timestamps. *Distributed Computing*, 31(5):327–342, 2018.

[33] M. G. Merayo, R. M. Hierons, and M. Núñez. A tool supported methodology to passively test asynchronous systems with multiple users. *Information & Software Technology*, 104:162–178, 2018.

[34] M. G. Merayo, M. Núñez, and I. Rodríguez. Formal testing from timed finite state machines. *Computer Networks*, 52(2):432–460, 2008.

[35] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling (to appear)*, 2019.

[36] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.

[37] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and

M. Deardeuff. How Amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[38] A. Petrenko and N. Yevtushenko. Testing from partial deterministic FSM specifications. *IEEE Transactions on Computers*, 54(9):1154–1165, 2005.

[39] A. Petrenko, N. Yevtushenko, and G. von Bochmann. Testing deterministic implementations from their nondeterministic FSM specifications. In *9th IFIP Workshop on Testing of Communicating Systems, IWTCS'96*, pages 125–140. Chapman & Hall, 1996.

[40] I. Phillips. Refusal testing. *Theoretical Computer Science*, 50(3):241–284, 1987.

[41] E. Rohmer, S. P. N. Singh, and M. Freese. V-REP: A versatile and scalable robot simulation framework. In *26th IEEE/RSJ Int. Conference on Intelligent Robots and Systems, IROS'13*, volume 1, pages 1321–1326. IEEE Computer Society, 2013.

[42] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2010.

[43] J. Schmaltz and J. Tretmans. On conformance testing for timed systems. In *6th Int. Conf. on Formal Modeling and Analysis of Timed Systems, FORMATS'08, LNCS 5215*, pages 250–264. Springer, 2008.

[44] M. Shafique and Y. Labiche. A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer*, 17(1):59–76, 2015.

[45] J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing, LNCS 4949*, pages 1–38. Springer, 2008.