Robotics and Autonomous Systems 165 (2023) 104426

Contents lists available at ScienceDirect

Robotics and Autonomous Systems

journal homepage: www.elsevier.com/locate/robot

Implementation relations and testing for cyclic systems: Adding probabilities☆

Manuel Núñez [a], Robert M. Hierons [b], Raluca Lefticaru [c,*]

[a] Design and Testing of Reliable Systems Research Group, Universidad Complutense de Madrid, Madrid, 28040, Spain
[b] Department of Computer Science, The University of Sheffield, Sheffield, SD1 4DP, UK
[c] Department of Computer Science, University of Bradford, Bradford, West Yorkshire BD7 1DP, UK

ARTICLE INFO

Article history:
Available online 6 April 2023

Keywords:
Probabilistic systems
Cyclic systems
Model-based testing
Implementation relations

ABSTRACT

This paper concerns the systematic testing of robotic control software based on state-based models. We focus on cyclic systems that typically receive inputs (values from sensors), perform computations, produce outputs (sent to actuators) and possibly change state. We provide a testing theory for such cyclic systems where time can be represented and probabilities are used to quantify non-deterministic choices, making it possible to model probabilistic algorithms. In addition, *refusals*, the inability of a system to perform a set of actions, are taken into account. We consider several possible testing scenarios. For example, a tester might only be able to passively observe a sequence of events and so cannot check probabilities, while in another scenario a tester might be able to repeatedly apply a test case and so estimate the probabilities of sequences of events. These different testing scenarios lead to a range of implementation relations (notions of correctness). As a consequence, this paper provides formal definitions of implementation relations that can form the basis of sound automated testing in a range of testing scenarios. We also validate the implementation relations by showing how observers can be used to provide an alternative but equivalent characterisation.

1. Introduction

There has been growing use of robotics in a number of important domains such as manufacturing and transport. The criticality of these domains has led to significant interest in the development of techniques for verifying robotic software, with additional challenges resulting from the nature of such software and the need for them to operate in complex, varying environments.

Testing remains a crucial component of the verification process and so it is important to have test generation techniques that are efficient and effective. Automated test generation and execution are particularly desirable since they aid scalability, reduce cost, and have the potential to limit the scope for human error. Testing can occur at several points in the development process. For example, one can test individual software and hardware components and one can test the final deployed system.

The focus of this paper is testing robotic control software and this testing can potentially be carried out within a simulation. Simulations can be used in order to test a *model* of the system (model-in-the-loop), to test the software (software-in-the-loop), or to test a hardware component (hardware-in-the-loop). Such testing can reveal design flaws before deployment, potentially leading to a reduction in development time and cost. Typically, it is often possible to run many more tests in simulation than one feasibly could in deployment. An additional benefit is that there is the potential to run tests that are not possible when testing a deployed system since, for example, there may be safety concerns. The importance of testing within simulation, for autonomous systems, was highlighted in a recent survey of the literature and interviews with practitioners [1]. It is important to note, however, that testing within simulation is not sufficient on its own: it is still important to test the deployed system since a simulation is only ever an approximation for the system's environment.

There are a number of domains, such as avionics and the automotive industry, in which testing within simulation is relatively common practice. Interestingly, an empirical study found that many real faults could be discovered by testing within simulation [2]. Despite this, a recent study found that although simulation-based testing was used within robotics development, most developers did not use it to support test automation [3].

☆ This work has been supported by EPSRC, United Kingdom grant EP/R025134/2 RoboTest: Systematic Model-Based Testing and Simulation of Mobile Autonomous Robots, the Spanish MINECO-FEDER grant PID2021-122215NB-C31 (AwESOMe) and the Region of Madrid grant S2018/TCS-4314 (FORTE-CM) co-funded by EIE Funds of the European Union.

* Corresponding author.
E-mail addresses: mn@sip.ucm.es (M. Núñez), r.hierons@sheffield.ac.uk (R.M. Hierons), r.lefticaru@bradford.ac.uk (R. Lefticaru).

https://doi.org/10.1016/j.robot.2023.104426

The aim of the work reported in this paper is to form the basis for sound and systematic automated test generation for the testing of robotic control software. We use models, in the form of state machines, as the basis for testing since these are available at a relatively early stage. There may be potential to use simulation as the basis for generating test cases for deployment testing but we see this as being a problem for future work; we briefly comment on this in Section 8.

The use of models, to drive test generation, fits within the wider area of *model-based testing (MBT)*. MBT has been an area of interest for many years, with work going back to the 1950s [4], and there are associated professional software testing tools (see, for example, [5,6]) and reports of successful use in industry (see, for example, [5,7]). Additional benefits arise where the notation used to describe the model has a formal semantics that allows one to map a model to a mathematical entity that can be analysed. A model written using such a notation can form the basis for systematic and sound automated test generation and execution (see, for example, [8]). However, until relatively recently, it appears that there has been very little use of state machines, written using a notation with a formal semantics, in robotics.

The work described in this paper was motivated by the recent development of RoboChart [9], a domain-specific state-based notation for modelling robotic software, and RoboSim [10], the corresponding notation for describing simulations. These notations are intended to be similar to the types of notations used by robotocists to define control software. The models are thus cyclic, representing software that reads values from sensors (receives inputs), then performs computations, before sending values to actuators (outputs). Importantly, these notations have been given a semantics through a mapping to the process algebra CSP [11] and CSP has a formal semantics. The existence of this formal semantics leads to the potential to develop automated and sound MBT techniques for RoboChart and RoboSim models in order to support testing within simulation and also deployed robots. Recent work has explored testing from RoboChart and RoboSim models based on their CSP semantics [12–14]. The use of the CSP semantics makes it possible to provide a common approach across both notations, while the existence of mappings from RoboChart and RoboSim to CSP ensures that developers do not need to know CSP. A CSP model can be mapped to its semantics in the form of a labelled transition system (LTS) and this is the type of model we consider. LTSs can be used to represent the operational behaviour of a system in terms of state and transitions between states. Intuitively, an LTS is a graph with a distinguished *initial* state and where edges are labelled. If the system is in a certain state q and there is an edge with label a connecting q to state q', then the LTS can move from state q to state q' through the task/event associated with the label a.

The recent research on testing from RoboChart and RoboSim models allowed software to be represented by a state-machine with discrete time. Discrete time was used in order to be consistent with both how simulations operate and also the typical cycle seen in embedded control software: these read from actuators (receive inputs), carry out computations, send values to actuators (outputs) and then repeat the cycle. As a result, it is sufficient to use discrete time when testing robotic control software. However, once the hardware is included there is a need to move to continuous time and so, potentially, hybrid models in which differential equations are used to model physical properties/laws. The use of continuous time and hybrid models is a problem for future work.

Recently, RoboChart has been extended to include probabilistic information [15], making it possible to model probabilistic algorithms used in robotics. The work described in this paper extends the testing framework developed for RoboChart [14] to include such probabilistic information. We follow the approach taken in RoboChart [15], in which probabilities are used to model algorithms but they are not used to model the environment. This essentially allows a 'most general' environment: one that can do anything and where one has no probabilistic information about the environment.

There is potential to incorporate information about the environment by either creating a simulation model of the expected environment or by expressing properties of the environment using a notation such as RoboWorld [16]. This would address a potential disadvantage of the approach we take, in which the environment can do anything: the test cases produced may not be realistic. Note, however, that constraining the environment could mean that testing does not reveal faults that can only be observed in environments that do not satisfy these constraints. The models would also be more complicated since they would contain both continuous variables and discrete states; they would be *hybrid*.

This paper makes the following main contributions.

1. We provide a formalism to model cyclic systems where discrete time can be represented and probabilities are used to quantify non-deterministic choices. In addition, *refusals*, the inability of a process to perform a set of actions, are taken into account.
2. The models provided by our formalism are compatible with the latest version of RoboChart [15]. In particular, our models alternate between input states (that are an extension of non-probabilistic states in RoboChart) and output states (that are the equivalent to probabilistic junctions in RoboChart).
3. We define 15 implementation relations and analyse the relations among them. Intuitively, an implementation relation indicates whether a System Under Test (SUT) represents a *valid* implementation of a specification. Depending on the interpretation of 'valid' we may have different relations and the choice of implementation relation may depend on what a tester can observe in testing. For example, the consideration of probabilities within an implementation relation only makes sense if the tester can repeat a test multiple times and so estimate probabilities within the SUT. These implementation relations vary from the simplest version, where probabilities and refusals are abstracted, to more involved ones. The simpler versions will mark more SUTs as valid.
4. We present alternative characterisations of our implementation relations based on a notion of *observer*.

Within the context of testing of robotic control software, our work provides the potential for sound automated test generation. If we start with a RoboChart model then we can use a tool (RoboTool [17]) to generate the corresponding CSP model. The CSP model can be mapped to its semantics in the form of an LTS (the type of state-based model that we consider). This overall approach has the advantage of allowing the developer to use a state-based notation (RoboChart) that is similar to those used in robotics but also utilise sound, systematic techniques that are based on the formal semantics (LTS) of the original model.

Test generation that uses the implementation relations we define is sound since it cannot reject a correct implementation (one that conforms to the model under the implementation relation used). There are then a number of ways in which test generation can be automated. One option is to take the approach often used when testing from formal state-based models, in which test generation involves randomly generating sound test cases (see, for example, [18]). Alternatively, one might seed faults in the model M, to create mutants, and for each mutant N use a model-checker to find a test case that reveals this fault. The resultant test cases are guaranteed to find the corresponding faults in

the software. Such an approach has already been developed for RoboChart models [12] but did not take into account either time or probabilities.

We classify results between theorems and propositions by taking into account the importance of a result in the context of the whole framework. For some theorems we do not include a proof because the proof is straightforward or, despite the result being important, the theorem is a combination of previous results. Similarly, in most cases a proof is not given for a proposition since this proof would be a simple application of some results or definitions but there are exceptions. As a result of these factors, we do not give an explicit proof for all the theorems and we do provide proofs of some propositions.

The rest of the paper is organised as follows. We start, in Section 2, by describing the context we consider and outlining related work. Section 3 describes the types of models we use and provides background concepts and definitions. In Section 4 we give the first set of implementation relations, which are based on traces but without probabilities or refusals. In Section 5 we show how probabilities can be added and extend this to include refusals in Section 6. We then consider, in Section 7, an alternative characterisation, showing how a notion of observers captures the observations we consider and so can be used as the basis for providing alternative, but equivalent, definitions of the implementation relations. Finally, we draw conclusions and discuss possible future work in Section 8.

2. Technical context and related work

As explained in Section 1, the work described in this paper is motivated by the development and use of the RoboChart [9] and RoboSim [10] notations. These notations essentially allow one to model a piece of robotic software as a state machine: there are logical states, internal variables, and transitions between states. Transitions can have guards, which define conditions under which the transitions can be triggered. Transitions can also update the values of internal variables. Next, we present the running example used in this paper.

Example 1. This example, abstracting a mail delivery robot, is inspired by previous work, where it has been modelled in PRISM [19] and later using the RoboChart notation [15]. The robot is used for internal mail delivery in a workspace with several offices, labelled $0-3$. A map of the workspace is given in Fig. 1(a): lines between offices represent corridors that can be used for navigation. The robot can recharge its battery in office 0 and can be called to any office to fetch mail and deliver it elsewhere. Following a fetch command, the robot could move to another office or remain in the same office, if the battery is almost empty, following the Markov model in Fig. 1(b), that models only the navigation between the offices. The transitions are labelled with the associated probability. In this case, $1/3$ means that the robot could choose between the three possible actions, stay in the office, move to one adjacent office, or move to the other adjacent office, with the same probability. Fig. 1(c) extends the movement state machine with additional states, for example charging states, possible only in office 0 and other transitions, such as ready when the battery is full, and keep charging while battery is notFull. Some of these states are represented in a different colour, as they will have a different meaning in our model, as will be explained later.

Both RoboChart and RoboSim have been given a formal semantics [9,10] through a mapping to the process algebra CSP. For timed models, the mapping is actually to a timed version of CSP, called tock-CSP [20, Chapter 14]; throughout this paper, when we refer to CSP we actually mean tock-CSP. In turn, CSP itself can be given a formal semantics that describes what it means for an implementation to be correct with respect to a model or specification [20]. Communication between CSP processes is defined in terms of synchronisation and so the classical CSP semantics does not distinguish between inputs and outputs. This is problematic from a testing perspective since inputs and outputs play very different roles in testing: the tester controls inputs and the SUT controls outputs. However, recent work has shown how the classical CSP semantics can be adapted by suitably treating inputs and outputs and so can form the basis for testing [21,22].

Since RoboChart and RoboSim have a formal semantics, and this formal semantics provides a precise mathematical model, there is potential for systematic and sound automated test generation. The focus on testing from RoboChart and RoboSim models has thus been based on this CSP semantics (see, for example, [12–14]). The corresponding test generation techniques allow models to contain discrete time; as previously mentioned, the use of discrete, as opposed to continuous, time was motivated by systems and simulations often being cyclic.

The previous work on testing from RoboChart and RoboSim did not allow models to be probabilistic and so could not be used to test probabilistic properties of robotic algorithms. The work described in this paper extends the testing framework developed for RoboChart [14] to include probabilistic information as well as time. In order to be consistent with the probabilistic version of RoboChart [15], probabilities are used to model robotic software but they are not used to model the environment. As a result, probabilistic information can be provided regarding the output produced, since outputs are produced by the SUT. In contrast, probabilities are not used to define the likelihood of inputs in a given state.

The main goal of this paper is to define a wide variety of implementation relations that can be used to decide, taking into account different characteristics and features of the analysed systems, whether an SUT is a good implementation of a specification. We started by considering ioco [18]; this is a widely used implementation relation to relate specifications and SUTs but its original formulation does not take into account either time or probabilities. There are several timed variants of ioco [23–25] sharing the same name, tioco, aiming at extending ioco to correctly deal with the inclusion of time in the analysis of systems. They differ from each other in several aspects. The most relevant for our work is whether quiescence[1] can be observed because quiescence is indeed a particular case of refusal. Although we could have started with one of these notions, we preferred to build on top of our previous work [14] where we introduced an implementation relation specifically targeting cyclic systems and consistent with (tock-)CSP [20, Chapter 14].

Concerning work on testing probabilistic systems, the first approaches considered testing frameworks where either tests were passed with a certain probability or the semantic model relied on the probability associated with performing a certain trace [26–29]. However, early work did not distinguish between inputs and outputs and so cannot be used within an ioco-inspired framework. Subsequent work did distinguish between inputs and outputs but only in the limited scope of probabilistic FSMs [30–32]. Finally, proper probabilistic extensions of ioco in the context of input–output probabilistic transition systems were introduced [33–35]. We consider our previous work [34,35] as our initial step but we have to take into account certain restrictions to be consistent with recent work on adding probabilities to RoboChart [15].

[1] A system is in a quiescent state if it cannot change state or produce output without first receiving an input.

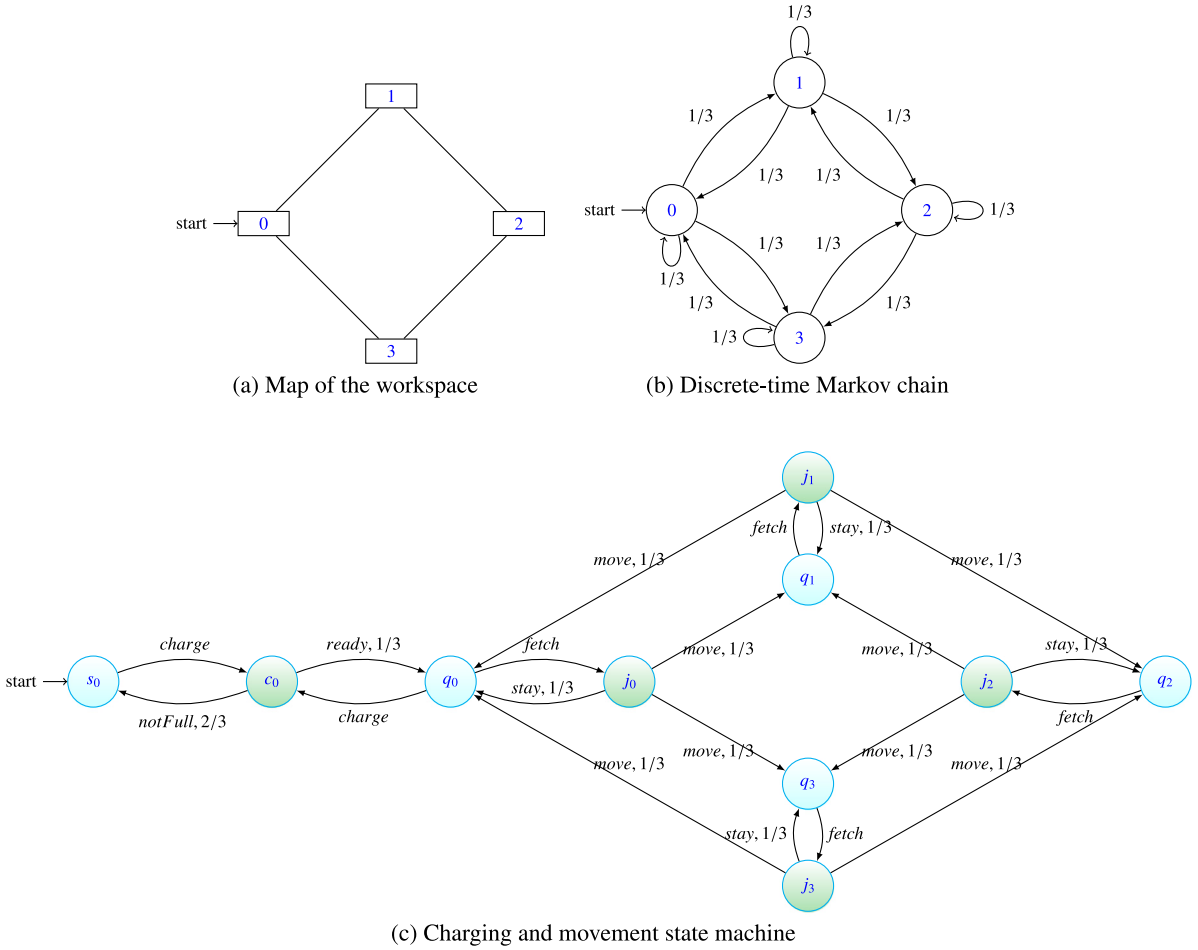

Fig. 1. State machine model of a component of a mail delivery robot. (For interpretation of the colours in this figure, the reader is referred to Example 2). *Source:* Inspired from [15,19].

Concerning refusals, the original formulation [36] did not include characteristics such as time, probabilities and a distinction between inputs and outputs. There exists a testing framework for probabilistic processes where refusals could be observed [37]. Unfortunately, this approach does not distinguish between inputs and outputs and, therefore, it is not suitable for the situation considered in this paper. There is a variant of ioco where refusals can be observed and inputs might be unspecified, that is, systems do not need to be input-enabled [38]. Our approach departs from this one in several lines (in addition to taking into account time and probabilities). First, in order to be consistent with previous work in formal modelling of cyclic systems, we only allow refusals to be observed in states that do not have *urgent* actions, that is, states from which no outgoing transition is labelled by an output. Without this restriction, the tester is able to distinguish systems that should be equivalent. Specifically, using a process algebraic notation where actions preceded by ? and ! denote, respectively, an input and an output and $P +_\pi Q$ denotes that P is chosen with probability π and Q is chosen with probability $1 - \pi$, we have that $(?i_1; !o_1; stop) +_\pi (?i_1; !o_2; stop)$ and $?i_1; ((!o_1; stop) +_\pi (!o_2; stop))$ are semantically equivalent in all the implementation relations studied in this paper. The reason for this is that we cannot observe refusals in the states where outputs are available. Second, this previous work [38] considers a distributed setting in which there are multiple ports, at which input can be received and to which output can be sent, and a restricted notion of input-enableness. The restriction placed is that if an input at a port p is blocked then all inputs at p are blocked (for the corresponding state). We only have one port (testing is not distributed) and so do not have this constraint. Therefore, concerning observation of refusals, we will build on top of our previous work [14].

Table 1
Classification of related work.

	Distinguish inputs & outputs	Time	prob.	Refusals
[21,22]	Yes	Yes	No	No
[14]	Yes	Yes	No	Yes
[18]	Yes	No	No	Limited (δ)
[23,25]	Yes	Yes	No	Limited (δ)
[24]	Yes	Yes	No	No
[26–29]	No	No	Yes	No
[30–32]	Limited (FSM)	No	Yes	No
[33,35]	Yes	No	Yes	Limited (δ)
[36]	No	No	No	Yes
[37]	No	No	Yes	Yes
[38]	Yes	No	No	Yes

In Table 1 we summarise the main characteristics of the discussed semantic frameworks. This table shows that there are several approaches considering the features that we would like to have, but none of them has all of them. Therefore, we consider that this paper represents a novel testing theory, building on top of ioco and our previous work [14] appropriately extended with probabilistic information [35], to analyse robotic software

where time must be taken into account, probabilities govern non-deterministic decisions and the refusal of a set of actions can be observed.

3. Background and models

In this section we present the different formalisms used in this paper to define models of robotic software. In addition, we review the most relevant properties that we expect these models to have.

3.1. Cyclic models

We are interested in cyclic models, such as those of embedded control systems, which can be found in robotics. Such systems operate in cycles in which they read values from sensors (and these are the *inputs* of the system), perform some calculations and write values to actuators (and these are *outputs* of the system). In addition, the passage of time is usually recorded between the occurrence of these cycles. It is important to note that the calculations might be probabilistic since probabilistic algorithms are used in robotics (see, for example, [39]) as well as network protocols such as the IEEE 802 standard.

3.2. Traces and LTSs

One of the main goals of this paper is to provide a testing framework for cyclic systems where time is taken into account, refusals can be observed and probabilities control some of the decisions. In this line, we need to define what an observation is. The most basic notion, which will be extended with other information, is that in testing we observe sequences of actions from a certain set A. As usual, we let A^* (resp. A^ω) be the set of finite (resp. infinite) sequences of elements belonging to A. We let $\epsilon \in A^*$ denote the empty sequence.

The most basic formalism that we will use in this paper is able to appropriately represent sequences of actions but still does not include information concerning time, probabilities and refusals. We will use a classical Labelled Transition System (LTS). As explained in Section 1, LTSs provide the semantics for several different notations used to represent robotic software and are suitable for describing the types of cyclic software considered in this paper.

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

- 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;
- $T \subseteq Q \times L \times Q$ is the transition relation.

The LTS is initially in state q_0, which can be seen as the situation in which the software has just been switched on and has yet to perform any computations. When the robotic software performs actions, the internal state may change, and this is represented by changing the state of the LTS. Formally, if the LTS is in a state $q \in Q$ and receives an action a such that $(q, a, q') \in T$, for a certain state $q' \in Q$, then it can move to state q' through a. For example, consider the model given in Fig. 1(c) and let us omit, for now, the probabilities labelling the edges. When the modelled mail delivery robot is switched on, it is in its initial state s_0. After some actions, including charging and moving between rooms, suppose that it is in state j_2. In this situation, the modelled robot is in room 2 and can move to room 1, performing the transition $(j_2, move, q_1)$, stay in room 2, performing the transition $(j_2, move, q_2)$, or move to room 3, performing the transition $(j_2, move, q_1)$.

Next we introduce some notation concerning transitions and their composition to form sequences.

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

$$
\begin{aligned}
q \xrightarrow{a} q' &\Leftrightarrow_{\text{def}} (q, a, q') \in T \\
q \overset{a}{\not\longrightarrow} &\Leftrightarrow_{\text{def}} \nexists q' \in Q : (q, a, q') \in T \\
q \overset{\epsilon}{\Longrightarrow} q' &\Leftrightarrow_{\text{def}} q = q' \\
q \overset{a_1 \ldots a_n}{\Longrightarrow} q' &\Leftrightarrow_{\text{def}} \exists q_1, \ldots, q_{n-1} \in Q : q \xrightarrow{a_1} q_1 \ldots q_{n-1} \xrightarrow{a_n} q' \\
q \overset{\sigma}{\Longrightarrow} &\Leftrightarrow_{\text{def}} \exists q' \in Q : q \overset{\sigma}{\Longrightarrow} q' \\
P \overset{\sigma}{\Longrightarrow} &\Leftrightarrow_{\text{def}} \exists r \in P : r \overset{\sigma}{\Longrightarrow} \\
q \overset{\sigma}{\not\Longrightarrow} &\Leftrightarrow_{\text{def}} \nexists q' \in Q : q \overset{\sigma}{\Longrightarrow} q' \\
P \overset{\sigma}{\not\Longrightarrow} &\Leftrightarrow_{\text{def}} \forall r \in P : r \overset{\sigma}{\not\Longrightarrow} \\
p \overset{\sigma}{\Longrightarrow} &\Leftrightarrow_{\text{def}} q_o \overset{\sigma}{\Longrightarrow}
\end{aligned}
$$

We define the *language* of p, denoted by $L(p)$, as the set of (finite) sequences of actions $\{\sigma \in L^* | q_0 \overset{\sigma}{\Longrightarrow}\}$.

As usual, the language of an LTS p is the set of (finite) sequences of actions that take p from its initial state to another state of the LTS. In other words, we collect all the sequences of actions that the modelled robotic software is allowed to perform. For example, if we consider again the model given in Fig. 1(c) and omit probabilistic information, one of the allowed behaviours corresponds to the *charge notFull charge ready fetch move* sequence, meaning that the robot is charging, while it is not full, it continues charging; then it is full and ready to work, it receives a fetch mail request and moves to another room.

3.3. Probabilistic-timed models

Once we have the basic formalism, we go one step forward to include time and probabilistic information. First, we would like to split the set of actions appearing in a model into two sets: *inputs* and *outputs*. Throughout the paper, I and O will represent the (disjoint) input and output alphabets of the considered systems and, therefore, we will have $L = I \cup O$.

Next, we discuss how time is added. In this paper we use discrete time and there are four main reasons for this. First, we focus on testing the software part of a robotic system. If we were testing the whole integrated robotic system (software and hardware), then we would need to model and test a cyber–physical system. In this case, it would be more appropriate to use continuous time and apply a variant, including probabilities and refusals, of one of the available hybrid conformance relations [40–43]. Second, the types of models used in robotics, and more generally embedded systems, are typically cyclic and have a step-semantics: a sequence of computations occur within a step, then time progresses, and a new cycle (step) occurs. Statecharts [44,45] provide a particularly popular language for such models. Third, simulators used in robotics also typically operate in such a step-based manner. Finally, we needed an approach that is consistent with the probabilistic extensions of the RoboChart language that has been devised to support the development of robotic systems [15]. Therefore, we use time that discretely evolves and we will use a special symbol, $\ominus$, to denote the passing of a unit of time. In order to be consistent with (tock-)CSP [20, Chapter 14] we also call this action 'tock'.

There are several approaches to define formalisms including probabilistic information [46,47]. The main distinguishing point is related to how probabilities are associated with actions. A *reactive* approach [48,49] considers that, for each state, the probabilities

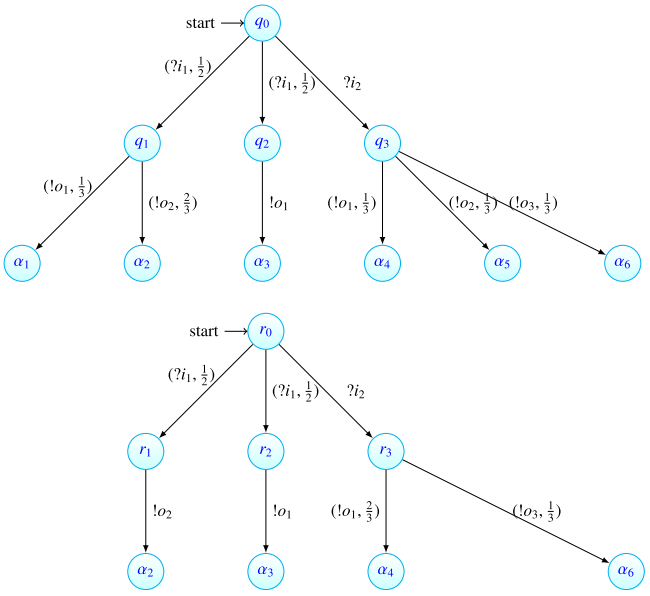

Fig. 2. Alternating reactive/generative models.

associated with a specific action add up to 1. The idea is that a system receives a certain action and probabilistically chooses, among the transitions labelled by this action, which one is selected. In a *generative* [48] approach we have that all the probabilities associated with the actions that can be performed at a certain state add up to 1. The idea is that a system probabilistically chooses which action, among all the available possibilities, to perform.

We have defined our formalism with two objectives in mind. First, we would like to consider the most widely used approach where there is a distinction between inputs and outputs: a combined reactive/generative view.

- Probabilities associated with inputs are *reactive*. Specifically, given a state s and an input $?i$, if one or more transitions from s have input $?i$ then there is a probability distribution over all the transitions from s with input $?i$. A graphical example of this is state q_0 in Fig. 2, where probabilities equal to 1 are omitted.
- Probabilities associated with outputs are *generative*. Specifically, given state s from which there are transitions with outputs, there is a probability distribution over all the transitions from s that are labelled with an output. A graphical example of this is state q_1 in Fig. 2.

The second objective is to provide a formalism that is consistent with recent work that has defined a probabilistic extension of RoboChart [15]. In this work, there are states and junctions, with states and junctions alternating: every transition from a junction goes to a state and every transition from a state goes to a junction. Transitions leaving junctions are assigned probabilities and those leaving states have no probabilities. Transitions from junctions represent system actions/computations and so the use of probabilities on these transitions makes it possible to represent probabilistic algorithms. Since we have both inputs and outputs, probabilistic junctions can be simulated by states where all the outgoing transitions are labelled by an output. In addition, RoboChart 'states' can be simulated by states from which all the outgoing transitions are labelled by inputs. We also have to ensure that transitions departing a probabilistic junction cannot reach another probabilistic junction. Note that a *reactive* choice can be easily transformed into a choice between inputs by translating the corresponding probabilities to the subsequent outputs. For example, the two models depicted in Figs. 2 (top) and 3 are semantically equivalent under all the notions used in this paper.

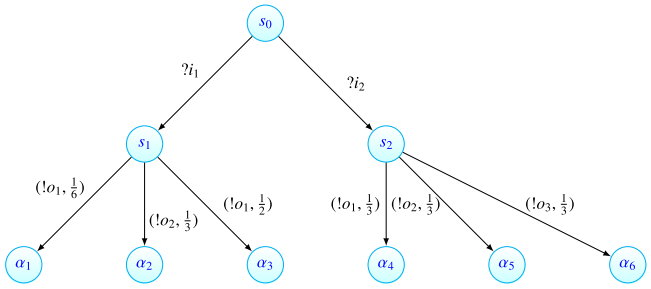

Fig. 3. Model without probabilistic choices between inputs.

In conclusion, we consider a version of Probabilistic Input–Output Transition Systems [34,35] (PIOTSs) where we impose a separation between inputs and outputs and have an alternation between inputs and outputs in the sense that transitions labelled by inputs (resp. outputs) cannot reach a state where inputs (resp. outputs) are available. As we mentioned before, in order to extend our previous work and provide the formalism with the capability to represent time, we have added $\ominus$ as an action denoting the passage of one time unit.

Definition 3 (*ptockLTS*)**.** A *probabilistic labelled transition system with tock (or ptockLTS)* 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 actions;
- $T \subseteq Q \times (\{\ominus\} \cup (L \times (0, 1])) \times Q$ is the *transition relation*, where $\ominus$ denotes that one unit of time passes.

Transitions belonging to T must also satisfy the following:

- p has *urgent outputs*, that is, for all $q \in Q$ and $!o \in O$, if $(q, (!o, \pi), q') \in T$ then there does not exist $q'' \in Q$ such that $(q, \ominus, q'') \in T$.
- p is *time deterministic*, that is, for all $q_1, q_2, q_3 \in Q$ we have that $(q_1, \ominus, q_2) \in T$ and $(q_1, \ominus, q_3) \in T$ implies $q_2 = q_3$.
- p is *reactive for inputs*, that is, for all $q \in Q$ and $?i \in I$ if $(q, (?i, \pi), r) \in T$ then

$$\sum \{\!|\ \pi' | \exists q' \in Q : (q, (?i, \pi'), q') \in T\ |\!\} = 1$$

  where $\{\!|$ and $|\!\}$ are multiset delimiters.
- p is *generative for outputs*, that is, for all $q \in Q$ and $!o \in O$ if $(q, (!o, \pi), r) \in T$ then

$$\sum \{\!|\ \pi' | \exists q' \in Q, !o' \in O : (q, (!o', \pi'), q') \in T\ |\!\} = 1$$

- p has a *separation between inputs and outputs*, that is, for all $q \in Q$ if there exist $\pi \in (0, 1]$, $r \in Q$ and $?i \in I$ (resp. $!o \in O$) such that $(q, (?i, \pi), r) \in T$ (resp. $(q, (!o, \pi), r) \in T$), then there do not exist $\pi' \in (0, 1]$, $q' \in Q$ and $!o \in O$ (resp. $?i \in I$) such that $(q, (!o, \pi'), q') \in T$ (resp. $(q, (?i, \pi'), q') \in T$). If all the probabilistic transitions departing from a state $q \in Q$ are labelled by inputs, then we will say that q is an *input state*; otherwise, we will say that q is an *output state*. Note that, due to urgency, a state having only a unique transition labelled by $\ominus$ is an input state.

- p is *alternating*, that is, for all $q \in Q$ if there exist $\pi \in (0, 1]$, $r \in Q$ and $?i \in I$ (resp. $!o \in O$) such that $(q, (?i, \pi), r) \in T$ (resp. $(q, (!o, \pi), r) \in T$), then there do not exist $\pi' \in (0, 1]$, $q' \in Q$ and $?i' \in I$ (resp. $!o' \in O$) such that $(r, (?i', \pi'), q') \in T$ (resp. $(r, (!o', \pi'), q') \in T$).
- p does not have *redundant transitions*, that is, for all $q, q' \in Q$ there do not exist $\pi_1, \pi_2 \in (0, 1]$, $\pi_1 \neq \pi_2$, and $a \in L$ such that $(q, (a, \pi_1), q') \in T$ and $(q, (a, \pi_2), q') \in T$.
- The *initial state* of p is an *input state*, that is, for all $(q_0, (a, \pi), q) \in T$ we have that $a \in I$. Note that q_0 may have a transition labelled by $\ominus$.

We denote by $pTockLTS(I, O)$ the set of ptockLTSs having I as input set and O as output set.

Let us comment on the restrictions associated with transitions of a ptockLTS. First, *urgency* of outputs is usually required to guarantee consistency with the normal behaviour of cyclic systems. Second, *time determinism* implies that a ptockLTS cannot branch as the result of time passing (that is, by performing a transition labelled by $\ominus$). As explained before, we combine a reactive model for inputs with a generative model for outputs and we do not allow *mixed* states: given a state, either all the available probabilistic actions are inputs or all the available probabilistic actions are outputs. We also want that our models are alternating: transitions labelled by an input (resp. output) cannot reach a state where an input (resp. output) is available. The lack of redundant transitions helps us to simplify subsequent computations involving probabilities: if we had two transitions with the same action and initial and final states, we could simply add the probabilities and consider a single transition. Finally, for compatibility with previous work, we require that the initial state is an input state. Note that we do not have any restriction on the states reached after the performance of a tock: it must depart from an input state but it can reach either an input or an output state.

Example 2. In Fig. 4 we present a ptockLTS corresponding to the state machine described in Example 1. Let $p_{robot} = (Q, s_0, I, O, T)$ denote this ptockLTS, with this having state set $Q = \{s_0, q_0 - q_3, j_0 - j_3, c_0, c_{-,-}, j_{-,-}\}$, initial state s_0, action sets $I = \{?c, ?f\}$ and $O = \{!m, !n, !r, !s\}$. Essentially, the events that trigger transitions in Fig. 1 are transformed into input symbols ($?f$ stands for `fetch`, $?c$ stands for `charge`) while responses are transformed into output symbols ($!m$ stands for `move`, $!s$ means `stay` in the same office, $!r$ stands for `ready`, meaning battery is full and $!n$ stands for battery `notFull`). The states $q_0 - q_3$ show the current office ($0 - 3$) where the robot is located; in case the robot receives an $?f$ input (for fetch), the next office will be probabilistically chosen. The states $j_0 - j_3$ are junctions and a state labelled $j_{k,l}$ shows it will move from office k to office l. Moreover, the model has $\ominus$ actions to simulate the passing of time (e.g. moving from one room to another or charging the battery takes time). In addition, to show the passing of time while no input is provided, self loops with $\ominus$ actions are added in states $s_0, q_0 - q_3$. Since ptockLTS models have urgent outputs, $\ominus$ actions cannot occur in the output states. Also, p_{robot} satisfies the other ptockLTS properties: it has time determinism, it is reactive for inputs and generative for outputs. The model has a separation between input and output states. In the graphical representation given in Fig. 4 we use two colours[2] to denote these states: light blue for input states and green for output states. For example, some input states are s_0, q_0 and $j_{0,1}$ while some output states are c_0, j_0 and j_1. It can be easily checked that p_{robot} is alternating and it does not have redundant transitions.

[2] Since this model is more involved, we use colours to distinguish between types of states. In the rest of the models depicted in the paper we use a single colour for all the states.

The original RoboChart model [15] that served as inspiration for Examples 1 and 2 is more complex: it has three state machines to model movement control, battery and task management, the first one inspiring our running example. We simplified the movement control model, in order to make it more readable, in several ways. We used a smaller office map, with 4 offices instead of the 9 in the original specification. In addition, a RoboChart model can include variables, constants, triggers, transitions with guards and other actions associated that are not captured in our ptockLTS formalism. For example, there are variables for the battery level, with transitions in the original model expressing how the battery is charging or how its level is decreased when the robot is moving. These are simplified in our ptockLTS model: we essentially say that a battery is empty or full. This battery level/status is represented through the state structure rather than the use of a variable; we have essentially expanded out the values for a variable that represents the battery status. Since all the variables used in the original RoboChart model have discrete values, from finite sets, one could just expand out the state space and eliminate all of the variables and data types, leading to a larger model but with no variables.

Although we use a simpler model to help with the exposition, the work we describe would apply to more complex models: one could again expand out the values variables or abstract out their values where appropriate. The resultant state machine would have no variables or guards and one could then generate (abstract) test cases based on the ptockLTS model. Note that if the values of some variables have been abstracted, then the resultant test cases need to be mapped to concrete test cases for either the testing of the SUT in simulation or the testing of the deployed system. Typically, this is achieved by building an adapter or it may be possible to use, for example, a model-checker [50,51] or a search-based testing technique [52].

We can adapt Definition 2 in the expected way (i.e. probabilities are included) to the new model. Therefore, we do not repeat the definition of 'double arrows' such as $q \stackrel{\sigma}{\Longrightarrow} q'$ or $q \stackrel{\sigma}{\not\Longrightarrow}$. We do need, however, to define a notion of trace in the new model where probabilities are taken into account. In particular, a sequence of actions can be performed several times (via different states) and, therefore, we have to add these probabilities.

Definition 4 (*Traces*)**.** Let $p = (Q, q_0, I, O, T)$ be a ptockLTS, $q \in Q$ be a state and $\sigma \in (I \cup O \cup \{\ominus\})^*$ be a sequence of input, output and tock actions. We let $prob(q, \sigma)$ denote the probability of performing the sequence σ from state q. Formally,

$$prob(q, \sigma) = \begin{cases} 1 & \text{if } \sigma = \epsilon \\ prob(q', \sigma') & \text{if } \sigma = \ominus\sigma' \wedge \\ & \quad (q, \ominus, q') \in T \\ \sum \{\!|\ \pi \cdot prob(q', \sigma') | (q, (a, \pi), q') \in T\ |\!\} & \text{if } \sigma = a\sigma' \wedge a \in I \cup O \end{cases}$$

The *set of timed traces* of p is defined as

$$\texttt{traces}(p) = \{\sigma \in (I \cup O \cup \{\ominus\})^* \mid prob(q_0, \sigma) > 0\}$$

The *set of probabilistic-timed traces* of p is defined as

$$\mathcal{P}\texttt{traces}(p) = \{(\sigma, prob(q_0, \sigma)) \mid \sigma \in (I \cup O \cup \{\ominus\})^*\}$$

We define two *languages* for a ptockLTS. First, timed traces simply forget the probability with which the trace can be performed as long as it is positive. This set is a conservative extension of our previous work [14] where probabilities were not involved. In the second notion, probabilistic-timed traces, a trace is a pair (sequence of actions, probability of performing the sequence). Note that if $\sigma \notin \texttt{traces}(p)$ then we have $(\sigma, 0) \in \mathcal{P}\texttt{traces}(p)$.

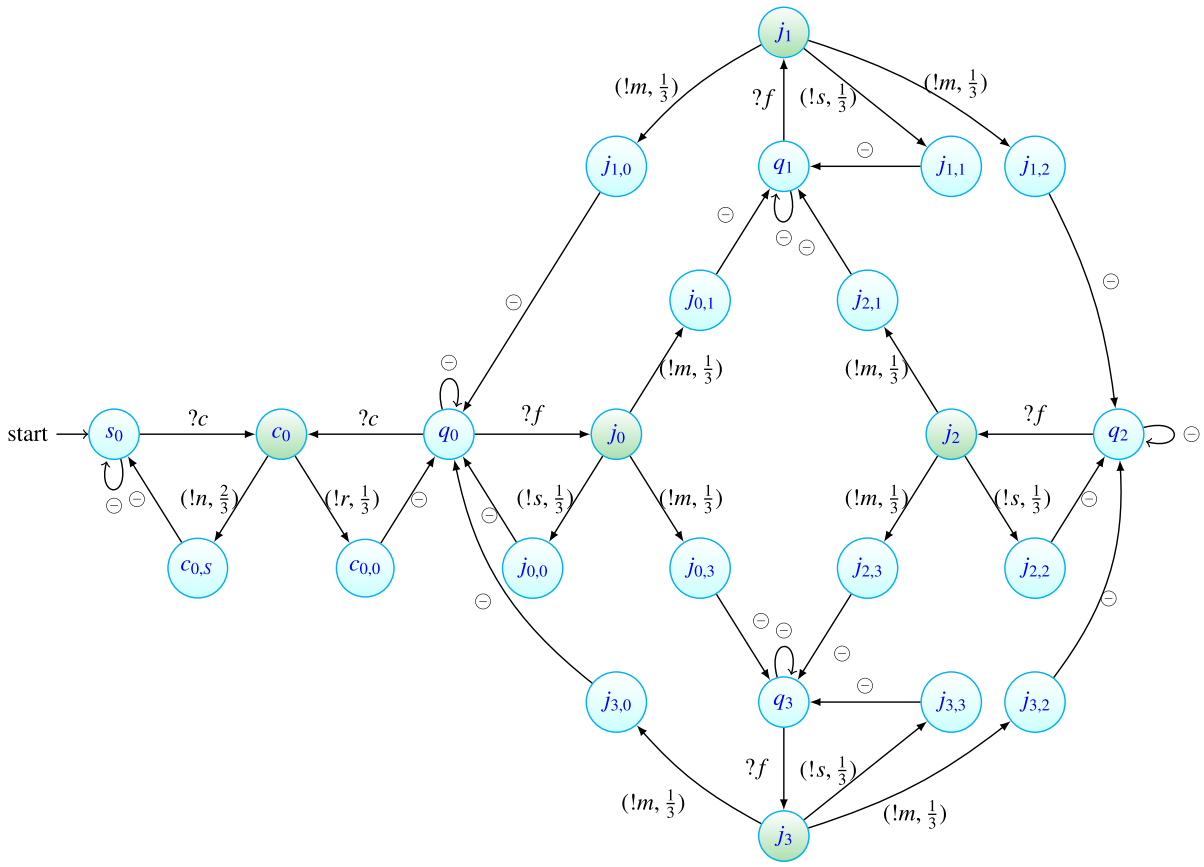

Fig. 4. ptockLTS mailing robot model. (For interpretation of the colours in this figure, the reader is referred to Example 2).

Next, we introduce additional notation to compute the set of states that can be reached after performing a given sequence of actions. Note that probabilities are *abstracted* when defining this concept.

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

1. q `after` $\sigma =_{\text{def}} \{q' \in Q \mid q \stackrel{\sigma}{\Longrightarrow} q'\}$
2. P `after` $\sigma =_{\text{def}} \bigcup \{q$ `after` $\sigma \mid q \in P\}$.

Finally, we would like to impose two additional restrictions on the ptockLTSs that we are going to use. First, they must allow time to progress. Technically, for each state of p, it must be possible to perform a sequence of inputs and outputs that leads to a state where $\ominus$ is available. Second, we do not allow ptockLTSs to show *Zeno behaviour*, that is, it is not possible to perform an infinite sequence of actions in finite time. Note that, again, probabilities are not taken into account when defining these properties.

Definition 6 (*Time Progressing and Zeno Behaviour*)**.** Let $p = (Q, q_0, I, O, T)$ be a ptockLTS. We say that p is *time progressing* if for all $q \in Q$ there exists $\sigma \in (I \cup O)^*$ such that $q \stackrel{\sigma\ominus}{\Longrightarrow}$. We say that p has *Zeno behaviour* if there exists a state $q \in Q$ and an infinite path from q with finitely many *tock* actions.

Example 3. Consider again the model given in Fig. 4. We illustrate now some of the concepts previously defined on p_{robot}. For example, if $\sigma = ?c!r\ominus?f!m\ominus$, then the following set can be computed s_0 **after** $\sigma = \{q_1, q_3\}$ and we can say that σ is a trace, that is, $\sigma \in \texttt{traces}(p_{robot})$. Moreover, after computing the associated probability of the trace, $prob(s_0, \sigma) = 1 \cdot \frac{1}{3} \cdot 1 \cdot 1 \cdot \left(\frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 1\right) = \frac{2}{9}$, it results that $(\sigma, \frac{2}{9}) \in \mathcal{P}\texttt{traces}(p_{robot})$.

Considering $\ominus$ actions, note that they are allowed in all input states of this model, thus ensuring time progression. If p_{robot} would not have, for example, state $c_{0,S}$ and instead of the two transitions $(c_0, (!n, \frac{2}{3}), c_{0,S})$ and $(c_{0,S}, \ominus, s_0)$ it would only have had one transition $(c_0, (!n, \frac{2}{3}), s_0)$, then the model would exhibit Zeno behaviour, given an infinite path $(?c{\cdot}!n)^\omega$ that has a finite number of $\ominus$ actions (more precisely, 0).

It is easy to prove, taking into account the two previous restrictions imposed on ptockLTSs, that they fulfil the following property. This property explicitly states that a time progressing ptockLTS cannot *stop time*.

Proposition 1. *Let $p = (Q, q_0, I, O, T)$ be a ptockLTS. 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 $q \stackrel{\sigma}{\Longrightarrow}$ and for all i we have that $\mu_i \in (I \cup O)^*$.*

This paper concerns the definition of implementation relations that define what it means for an SUT to be a correct implementation of a specification. In order to be able to formalise implementation relations, one needs to assume that the SUT can be represented by a model and it is normal to assume that this model (of the SUT) is one that can be expressed using the formalisation being used (the minimum hypothesis [53]). As previously noted, the formalism allows one to model the reading of values of sensors (represented as inputs) and the sending of values to actuators (represented as outputs). The formalism also has the cyclic step-based structure used in robotic software and also simulations. We assume that the behaviour of the SUT can be represented using the same formalism as the specification, and so the paper makes the following assumptions.

1. The specification is a ptockLTS q.
2. The behaviour of the SUT can be represented by an unknown ptockLTS p that has the same sets of inputs and outputs as q.
3. Both p and q are time progressing and do not have Zeno behaviour.

3.4. Including refusals

Finally, we will extend our formalism to consider *refusals*. Intuitively, a model can refuse a set of actions at a certain state if none of these actions is immediately available. However, we need to impose certain restrictions on when a model can refuse actions. The underlying step semantics leads to models that perform a (possibly empty) sequence of actions and then a time unit elapses, that is, a tock action is performed. In order to effectively observe the refusal of a set X, we need the tester to offer the SUT only the actions in X and then observe a deadlock. In testing, deadlocks are usually observed through a timeout. As a result, the observation of a refusal takes time and so we only allow a refusal to be observed immediately before time has passed, that is, before the performance of a tock action.

An immediate consequence of this situation is that refusals can only be observed in input states because, due to the urgency of output actions, tock actions are not available at output states. As a result of this, whenever a refusal is observed the SUT must be in a state where all outputs can be refused. The observation of the refusal of a set of outputs thus provides no more information than the refusal of the empty set; it simply implies stability. Thus, we do not include outputs in refusal sets.

Example 4. Consider again the model given in Fig. 4. Refusals are not observed in output states. For example, if the robot is in state j_0 then it will probabilistically choose whether to stay in room 0 or move to rooms 1 or 3, but it will not be possible to observe the refusal of a set of actions. However, if the robot is in an input state, then it is possible to observe the robot being in a state where it can refuse certain inputs. For example, if the robot is in state q_0 then the robot cannot refuse either a fetch or a charge command. In contrast, if it is in state s_0, then the robot can refuse a fetch command. Note that refusing this (input) action will not change the state of the robot.

Definition 7 (*Refusal*)**.** Let $p = (Q, q_0, I, O, T)$ be a ptockLTS, with $L = I \cup O$, $q \in Q$ be a state and $X \subseteq I$ be a set of inputs. We say that q *refuses* X, denoted by $q \xrightarrow{R(X)} q$, if the following two conditions hold:

1. $q \xrightarrow{\ominus}$ and
2. for all $a \in X$ we have that $q \overset{a}{\not\longrightarrow}$.

We use $R(X)$ to denote the refusal of set X and $\mathcal{R}(L)$ to denote the set of all possible refusals, that is, the set $\{R(X) | X \subseteq I\}$.

Note that the previous definition implicitly extends the set of transitions of a ptockLTS with the appropriate self-loop transitions labelled by refusals.

It is worth noting that this notion of refusals does not consider only maximal sets of refusals: it may happen that we have both $q \xrightarrow{R(X)} q$ and $q \xrightarrow{R(Y)} q$ even if $Y \subset X$. Indeed, refusals are downwardly closed: if $q \xrightarrow{R(X)} q$ and $Y \subseteq X$ then we must have that $q \xrightarrow{R(Y)} q$. When we define implementation relations that use refusals it will be clearer why restricting ourselves to maximal refusals would lead to some undesired results.

Finally, we can extend the definition of trace: a sequence of inputs and outputs, possible interspersed with $\ominus$ actions, with occurrences of refusals as long as these occurrences are followed by a $\ominus$ action. In addition, since the observation of a refusal takes time, we will impose the requirement that a trace cannot end with a refusal; it can end with the subsequent $\ominus$ action. Therefore, in contrast with usual sets of traces, this set is not prefix closed. Note that similar to the notion of probabilistic trace given in Definition 4 we allow traces having probability equal to 0 of being performed.

Definition 8 (*Probabilistic Timed Refusal Traces*)**.** Let $p = (Q, q_0, I, O, T)$ be a ptockLTS, with $L = I \cup O$. The set of *probabilistic timed refusal traces* of p is defined as

$$\mathcal{PR}\texttt{traces}(p) = \{(\sigma, prob(q_0, \sigma)) \mid \sigma \in (L \cup \{\ominus\} \cup (\mathcal{R}(I)\{\ominus\}))^*\}$$

Example 5. Let us consider p_{robot} given in Fig. 4. We can add refusals as self-loops in all input states since all of them have an outgoing transition labelled by $\ominus$. For example, $s_0 \xrightarrow{R(\{?f\})} s_0$, $q_0 \xrightarrow{R(\emptyset)} q_1$, $q_3 \xrightarrow{R(\{?c\})} q_3$ and $j_{0,0} \xrightarrow{R(I)} j_{0,0}$ among others.

The following sequences are timed refusal traces of p_{robot}:

$\sigma_1 = ?c!r\ominus?f!m \ominus R(\{?c\})\ominus$

$\sigma_2 = R(\{?f\})\ominus?c!n\ominus?c!r\ominus?f!s\ominus$

Finally, we can easily compute the associated probabilistic timed refusal traces: $(\sigma_1, \frac{2}{9}), (\sigma_2, \frac{2}{27}) \in \mathcal{PR}\texttt{traces}(p_{robot})$.

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 only requires that every behaviour (trace) of the SUT is also a trace of the specification. It therefore does not consider probabilities.

Definition 9. Let p and q be two ptockLTSs. We say that p conforms to q under *timed trace inclusion* if and only if $\texttt{traces}(p) \subseteq \texttt{traces}(q)$. We denote this $p \preceq q$.

The following property is immediate from Definition 9.

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

Timed trace inclusion is a natural notion of correctness if a specification or model is being used to define the set of allowed behaviours (timed traces) of a system. However, this notion of correctness can be too restrictive and to see this, consider the situation in which no behaviour of the specification includes a particular input $?i$ after a trace σ: σ is a trace of the specification but $\sigma?i$ is not a trace of the specification. Under timed trace inclusion, if the SUT produces the timed trace σ, then it cannot follow this by the input of $?i$. Such a restriction, on $?i$ not being allowed after σ, makes sense if an input being undefined in the specification corresponds to that input not being allowed. This situation might occur, for example, if there is a requirement that the field or option corresponding to $?i$ does not appear on the interface or a particular sensor has been turned off. However, sometimes we have that $?i$ not being specified after σ corresponds to 'do not care': all behaviours are allowed after $\sigma?i$. In such situations, timed trace inclusion is not the right implementation relation.

As a result of the above observation, we define two additional implementation relations. Both of these take the approach that any behaviour is allowed if an unspecified input is received. They differ, however, in when they consider an input $?i$ to not be defined after a trace σ and are consistent with the (untimed) implementation relations *ioco* and *uioco* [18].

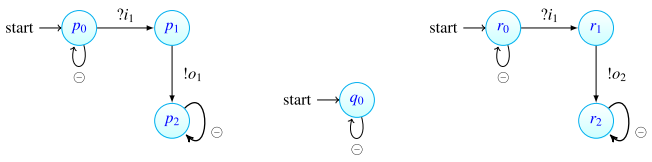

Fig. 5. Models (un)related by variants of trace inclusion.

Under one approach, which corresponds to *uioco*, $?i$ is said to be unspecified after σ if σ can take the specification to a state q in which $?i$ is not defined. Where this is the case, any behaviour is allowed if the SUT receives $?i$ after σ. The intuition is that, after σ, the specification might have been in a state where $?i$ is not defined and so any behaviour should then be allowed.

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

The following result is immediate from Definition 10.

Proposition 3. *The relation $\preceq^u$ is reflexive but need not be symmetric or anti-symmetric.*

It is interesting to note that, as the following example shows, $\preceq^u$ need not be transitive. This lack of transitivity is essentially a result of not requiring that models are *input-enabled*. Since we have an alternating model, *input-enableness* has a slightly different meaning than the usual one: given a state of a system, if the state is an input state then there are outgoing transitions labelled by each input of the alphabet.

Example 6. Consider the three ptockLTSs p, q and r given in Fig. 5. We have that $p \preceq^u q$ since q does not indicate what a correct implementation (in this case, p) should do concerning $?i$ and the subsequent behaviours. In addition, we have $q \preceq^u r$ because, in particular, we have $q \preceq r$. However, it is obvious that $p \preceq^u r$ does not hold.

However, if we restrict ourselves to input-enabled models then $\preceq^u$ is transitive.

Proposition 4. *Let p, q, and r be input-enabled ptockLTSs. If $p \preceq^u q$ and $q \preceq^u r$ then $p \preceq^u r$.*

Proof. Let σ be a trace of p. We will prove that σ must be a trace of r. By Definition 10, since q is input-enabled, $p \preceq^u q$, and σ is a trace of p, we have that σ is also a trace of q. Similarly, since r is input-enabled, $q \preceq^u r$, and σ is a trace of q, we have that σ is also a trace of r.

This shows that every trace of p is also a trace of r and it immediately follows that $p \preceq^u r$.

We can interpret the way in which $\preceq^u$ handles undefined inputs in terms of completing the specification so that all undefined behaviours are included. Since we are comparing traces of the specification and the SUT, we will remove probabilities in this *extended* specification.

Definition 11. Let $p = (Q, q_0, I, O, T)$ be a ptockLTS. The *completion* of p, denoted $\mathcal{C}(p)$, is the LTS $(Q \cup \{q_c\}, q_0, L', T')$ in which $q_c \notin Q$ is a fresh state, $L' = I \cup O \cup \{\ominus\}$ and $T' = T_0 \cup T_1 \cup T_2$ in which:

1. $T_0 = \{(q, a, q') | (q, (a, \pi), q') \in T\} \cup \{(q, \ominus, q') \in T\}$.
2. $T_1 = \{(q, ?i, q_c) | q \in Q \wedge ?i \in I \wedge q \stackrel{?i}{\not\Rightarrow}\}$
3. $T_2 = \{(q_c, a, q_c) | a \in I \cup O \cup \{\ominus\}\}$.

Note that the above defines an LTS whose set of actions is the set of the inputs and outputs of the original ptockLTS and $\ominus$.

Example 7. For the ptockLTS model given in Fig. 4, we build its completion LTS $\mathcal{C}(p_{robot})$ by keeping all the states of p_{robot} and adding a fresh new state q_c. For every (probabilistic) transition in the p_{robot} ptockLTS, we will have a corresponding transition in the LTS $\mathcal{C}(p_{robot})$, with the probability removed (T_0 set). Then, for those states that do not accept some inputs, T_1 will include transitions from these state to the new q_c state, labelled with all these inputs. For example, we will have $(s_0, ?f, q_c), (q_1, ?c, q_c), (j_0, ?c, q_c), (j_{0.0}, ?f, q_c) \in T_1$. Last, the new state q_c has self-loops labelled with all the visible actions: $(q_c, a, q_c) \in T_2$, where $a \in \{?c, ?f, !m, !n, !r, !s, \ominus\}$.

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

Theorem 1. *Given ptockLTS p and q with the same alphabets, $p \preceq^u q$ if and only if $\mathtt{traces}(p) \subseteq L(\mathcal{C}(q))$.*

Note that in the above we did not use $\preceq$ to compare p and $\mathcal{C}(q)$ since the latter is an LTS and $\preceq$ relates ptockLTSs. Therefore, we compared the set of traces of the original SUT and of the completed specification.

As previously noted, the $\preceq^u$ implementation relation deals with undefined inputs in a similar way to the (untimed) implementation relation *uioco*. It is also similar to how undefined inputs are considered in work on testing from a finite state machine (see, for example, [54–57]). The implementation relation *ioco*, however, takes a different approach that essentially says that the response to input $?i$ after σ is defined in specification q if there is some state q_1, that can be reached through σ ($q \stackrel{\sigma}{\Rightarrow} q_1$) and $?i$ is defined in q_1. The following adapts timed trace inclusion by following this approach.

Definition 12. Let p and q be two ptockLTSs. We say that $p \preceq^i q$ if and only if for all $\sigma \in \mathtt{traces}(p)$ either $\sigma \in \mathtt{traces}(q)$ or there exists a prefix $\sigma_1?i$ of σ, with $?i \in I$, such that $\sigma_1 \in \mathtt{traces}(q)$ and for all q_1 such that $q \stackrel{\sigma_1}{\Longrightarrow} q_1$ we have that $q_1 \stackrel{?i}{\not\Rightarrow}$.

We cannot represent $\preceq^i$, using completion, in the way we did with $\preceq^u$, because completion adds behaviours not allowed under $\preceq^i$. However, we can first convert the specification ptockLTS into a deterministic version. This process will use an approach that is similar to the classical transformation from non-deterministic to deterministic finite automata [58]. Given ptockLTS q, we let $\mathtt{det}(q)$ denote the *determinised* version; states of $\mathtt{det}(q)$ will be sets of states of q reached by a common trace. The idea is that all transitions departing from a given state and labelled by the same action are *unified*. This goal is achieved by using the `after` function introduced in Definition 5.

Definition 13. Let $p = (Q, q_0, I, O, T)$ be a ptockLTS. We write $\mathtt{det}(p)$ to denote the LTS $(\mathcal{P}(Q), \{q_0\}, L', T')$ in which $L' = I \cup O \cup \{\ominus\}$ and $(Q_1, a, Q_2) \in T'$ for $Q_1, Q_2 \in \mathcal{P}(Q)$ and $a \in I \cup O \cup \{\ominus\}$ if and only if $Q_2 = Q_1$ `after` a.

Example 8. Let us consider the p_{robot} model given in Fig. 4. We will provide a few states and transitions of its determinised LTS. For the deterministic transitions, the construction is straightforward: probabilities disappear from the transitions and states

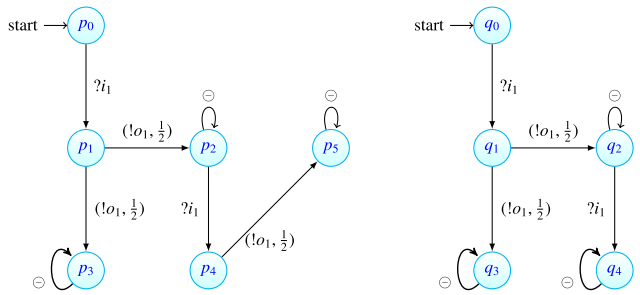

Fig. 6. $\preceq^u$ and $\preceq^i$ are not equivalent: $p \preceq^u q$ but $p \not\preceq^i q$.

like s_0 or c_0 are replaced by singleton sets $\{s_0\}$ and $\{c_0\}$, respectively. Then, we have some sources of non-determinism. For example, the two transitions labelled by $!m$, departing from j_0. The two states $j_{0,1}$ and $j_{0,3}$ will be replaced in $\mathtt{det}(p_{robot})$ by a new state $\{j_{0,1}, j_{0,3}\}$, and the transition $\{j_0\} \xrightarrow{!m} \{j_{0,1}, j_{0,3}\}$ will appear in $\mathtt{det}(p_{robot})$. Further on, after computing the set $\{j_{0,1}, j_{0,3}\}$ $\mathtt{after}$ $\ominus$, we have the transition $\{j_{0,1}, j_{0,3}\} \xrightarrow{\ominus} \{q_1, q_3\}$. Following the same approach we obtain, for example, $\{j_1, j_3\} \xrightarrow{!m} \{j_{1,0}, j_{1,2}, j_{3,0}, j_{3,2}\}$, then $\{j_{1,0}, j_{1,2}, j_{3,0}, j_{3,2}\} \xrightarrow{\ominus} \{q_0, q_2\}$, $\{q_0, q_2\} \xrightarrow{?f} \{j_0, j_2\}$ and $\{q_0, q_2\} \xrightarrow{?c} \{c_0\}$.

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

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

Similar to Theorem 1, we did not use $\preceq$ to compare p and $\mathcal{C}(\mathtt{det}(q))$ since the latter is an LTS and $\preceq$ relates ptockLTSs. Again, we compared the set of traces of the original SUT and of the extended specification but now we determinise the specification before completing it.

It is important to note that the application of $\mathtt{det}(q)$ could lead to states of the new LTS corresponding to infinite sets of states of the original ptockLTS q. If this happens then we might not be able to construct $\mathtt{det}(q)$, even using bounded approaches. In order to overcome this problem, it is usually assumed that we work with *finitely-branching* systems.

Definition 14. Let $q = (Q, q_0, I, O, T)$ be a ptockLTS. We say that q is *finitely-branching* if for every state $q_1 \in Q$ and action $a \in I \cup O \cup \{\ominus\}$ we have that q_1 $\mathtt{after}$ a is finite.

The following result, whose proof follows easily by induction on the length of the considered trace σ, says that the states of the LTS $\mathtt{det}(q)$ that are reached by finite traces are finite.

Proposition 5. *Let q be a ptockLTS, q_1 a state of q, and $\sigma \in \mathtt{traces}(q)$ be a trace of q. If q is finitely-branching then q_1 $\mathtt{after}$ σ is finite.*

The following result gives relationships between the implementation relations introduced in this section.

Theorem 3. *Given ptockLTSs $p = (Q, q_0, I, O, T)$ and $q = (Q', q'_0, I, O, T')$:*

1. *If $p \preceq q$ then we also have both $p \preceq^u q$ and $p \preceq^i q$.*
2. *Neither $p \preceq^u q$ nor $p \preceq^i q$ implies $p \preceq q$.*
3. *If $p \preceq^i q$ then we also have $p \preceq^u q$.*
4. *$p \preceq^u q$ does not imply $p \preceq^i q$.*

Proof. The first result is an immediate consequence of the definitions of $\preceq$, $\preceq^u$ and $\preceq^i$.

For the second result, let q given in Fig. 5 (centre) be a specification ptockLTS and p given in Fig. 5 be an SUT ptockLTS. First, it is obvious that we have both $p \preceq^u q$ and $p \preceq^i q$ since under these implementation relations, a valid implementation of q can do anything after the input $?i_1$. However, $p \preceq q$ does not hold because, for example, $?i_1!o_1 \in \mathtt{traces}(p)$ but $?i_1!o_1 \notin \mathtt{traces}(q)$.

For the third result, note that $p \preceq^i q$ implies that given a sequence $\sigma \in \mathtt{traces}(p)$, we have that either $\sigma \in \mathtt{traces}(q)$ or there exists a prefix $\sigma_1?i$ of σ, with $?i \in I$, such that $\sigma_1 \in \mathtt{traces}(q)$ and for all q_1 such that $q \stackrel{\sigma_1}{\Longrightarrow} q_1$ we have that $q_1 \stackrel{?i}{\not\rightarrow}$. Next, the second part of the previous condition implies that there exists a prefix $\sigma_1?i$ of σ, with $?i \in I$, such that $\sigma_1 \in \mathtt{traces}(q)$ and there is a state q_1 such that $q \stackrel{\sigma_1}{\Longrightarrow} q_1$ and $q_1 \stackrel{?i}{\not\rightarrow}$. This means that σ is allowed under both $p \preceq^u q$ and $p \preceq^i q$. Therefore, we conclude with the desired result.

For the last part of the result, we have to find two ptockLTSs p and q such that $p \preceq^u q$ but $p \preceq^i q$ does not hold. Consider the ptockLTSs p and q depicted in Fig. 6. The only difference between these processes is that they can perform different actions after the sequence $?i_1!o_1?i_1$: p can perform $!o_1$ and q can perform $\ominus$. First, we have that $p \preceq^u q$ since the sequence $?i_1!o_1$ can take q to a state (q_3) in which $?i_1$ is not specified. Therefore, using the definition of $\preceq^u$, all behaviours are allowed after $?i_1!o_1?i_1$. Second, $p \preceq^i q$ does not hold because the sequence $?i_1!o_1$ can take q to a state (q_2) in which $?i_1$ is specified and so, taking into account the definition of the $\preceq^i$ implementation relation, the only behaviours allowed after $?i_1!o_1?i_1$ are those that are traces of q.

Finally, we study the relationships between our first three implementation relations if we do not have unspecified inputs.

Theorem 4. *Let $p = (Q, q_0, I, O, T)$ and $q = (Q', q'_0, I, O, T')$ be ptockLTSs. If q is input-enabled then our implementation relations relate the same ptockLTSs, that is, we have*

$$p \preceq q \iff p \preceq^u q \iff p \preceq^i q$$

The previous trace-based implementation relations where probabilities have been abstracted are a simple way to relate specifications and SUTs and have some nice properties. Moreover, they can be alternatively characterised as inclusion of two sets of traces: the one corresponding to the ptockLTS representing the SUT and the one representing either the specification (for $\preceq$) or an LTS constructed from the specification (for $\preceq^u$ and $\preceq^i$). However, probabilities have been completely abstracted because we use traces where the only relevant quantity is whether the trace can be performed with probability greater than zero. In the next section we study how the previous implementation relations can be extended to consider additional probabilistic information.

5. Implementation relations with probabilities

In this section we assume that the environment can still only observe traces of visible actions and time (i.e. it cannot observe refusals yet) but we will take into account the probability with which these traces can be performed. Our first implementation relations follows the idea of trace containment: if a sequence is a trace of the SUT then it must be a trace of the specification. However, we have to appropriately deal with probabilities. Intuitively, we would like to allow valid SUTs to not perform some traces of the specification but if the trace can be performed, then it should do it with the specified probability or a higher probability. The idea is that the allowed options *absorb* the probability associated with the missing options. For example, consider the

models depicted in Fig. 2. We have $r \preceq q$ and we would like that this relation stays in the probabilistic case. Since r_1 and r_3 have *fewer outputs*, respectively, than q_1 and q_3, the traces associated with the available outputs will have a higher probability. This is reflected in the following definition.

Definition 15. Let A, B be two sets of probabilistic traces. We write $A \preccurlyeq B$ if for all $(\sigma, \pi) \in A$ we have that either $\pi = 0$ or there exists $(\sigma, \pi') \in B$ such that $\pi \geq \pi' > 0$.

Let p and q be two ptockLTSs. We say that p conforms to q under *probabilistic timed trace inclusion* if and only if $\mathcal{P}\texttt{traces}(p) \preccurlyeq \mathcal{P}\texttt{traces}(q)$. We denote this $p \preceq_{\mathcal{P}} q$.

The proof of the following result is immediate from Definition 15.

Proposition 6. *The probabilistic timed trace inclusion relation $\preceq_{\mathcal{P}}$ is reflexive and transitive but need not be symmetric or anti-symmetric.*

Our next step is to adapt the $\preceq^u$ and $\preceq^i$ relations to the new probabilistic setting. The role of these implementation relations is to provide a satisfactory answer when there are unspecified inputs in the specification, that is, when no behaviour of the specification includes a particular input $?i$ after a trace σ. In $\preceq^u$ we considered that this is the case if σ can take the specification to a state q in which $?i$ is not defined. Given a model p, a probabilistic interpretation of this is that $(\sigma, \pi), (\sigma?i, \pi') \in \mathcal{P}\texttt{traces}(p)$, with $\pi > 0$ and $\pi > \pi'$.[3] In particular, note that if $p \stackrel{\sigma?i}{\Longrightarrow}$, then this case is also covered because $\pi' = 0$, that is, $(\sigma?i, 0) \in \mathcal{P}\texttt{traces}(p)$. Intuitively, if we have $\pi = \pi'$, then we can claim that there are outgoing transitions labelled by $?i$ departing from all the states that we reach after σ (note that we always have $\pi \geq \pi'$). Therefore, if we have $\pi > \pi'$, then we know that in at least one of the states reached after σ we have that $?i$ is unspecified. In the case of $\preceq^i$ we request that none of the states reached after σ can perform $?i$. In probabilistic terms we can say that $(\sigma, \pi), (\sigma?i, 0) \in \mathcal{P}\texttt{traces}(p)$, with $\pi > 0$.

Definition 16. Let p and q be two ptockLTSs. We say that $p \preceq^u_{\mathcal{P}} q$ if and only if for all $(\sigma, \pi) \in \mathcal{P}\texttt{traces}(p)$ with $\pi > 0$ either $(\sigma, \pi') \in \mathcal{P}\texttt{traces}(q)$ with $\pi \geq \pi' > 0$ or there exists a prefix $\sigma_1?i$ of σ, with $?i \in I$, such that $(\sigma_1, \pi_1), (\sigma_1?i_1, \pi_2) \in \mathcal{P}\texttt{traces}(q)$ with $\pi_1 > \pi_2$.

We say that $p \preceq^i_{\mathcal{P}} q$ if and only if for all $(\sigma, \pi) \in \mathcal{P}\texttt{traces}(p)$ with $\pi > 0$ either $(\sigma, \pi') \in \mathcal{P}\texttt{traces}(q)$ with $\pi \geq \pi' > 0$ or there exists a prefix $\sigma_1?i$ of σ, with $?i \in I$, such that $(\sigma_1, \pi_1), (\sigma_1?i_1, 0) \in \mathcal{P}\texttt{traces}(q)$ with $\pi_1 > 0$.

We can adapt Theorem 3 to consider the new implementation relations. The proof of the result follows in the same way and, therefore, we omit it.

Theorem 5. *Let $p = (Q, q_0, I, O, T)$ and $q = (Q', q'_0, I, O, T')$ be ptockLTSs. Then we have:*

1. *If $p \preceq_{\mathcal{P}} q$ then we also have both $p \preceq^u_{\mathcal{P}} q$ and $p \preceq^i_{\mathcal{P}} q$.*
2. *Neither $p \preceq^u_{\mathcal{P}} q$ nor $p \preceq^i_{\mathcal{P}} q$ implies $p \preceq_{\mathcal{P}} q$.*
3. *If $p \preceq^i_{\mathcal{P}} q$ then we also have $p \preceq^u_{\mathcal{P}} q$.*
4. *$p \preceq^u_{\mathcal{P}} q$ does not imply $p \preceq^i_{\mathcal{P}} q$.*

We also adapt Theorem 4 to the probabilistic setting concerning the case when specifications are input-enabled.

[3] We could omit $\pi > 0$ because this is a consequence of $\pi > \pi'$, since $\pi' \geq 0$, but we prefer to explicitly mention it so that it is clear that p can perform σ.

Theorem 6. *Let $p = (Q, q_0, I, O, T)$ and $q = (Q', q'_0, I, O, T')$ be ptockLTSs. If q is input-enabled then the following three statements are equivalent: $p \preceq_{\mathcal{P}} q$; $p \preceq^u_{\mathcal{P}} q$; and $p \preceq^i_{\mathcal{P}} q$.*

We can now compare the implementation relations previously defined with those introduced in Section 4. The proof of the following result easily follows from the fact that in addition to the relation between the traces of specification and SUT that we considered in Section 4, we now take into account the probabilities with which traces are performed.

Proposition 7. *Let $p = (Q, q_0, I, O, T)$ and $q = (Q', q'_0, I, O, T')$ be ptockLTSs. Then the following hold.*

1. *If $p \preceq_{\mathcal{P}} q$ then $p \preceq q$.*
2. *If $p \preceq^u_{\mathcal{P}} q$ then $p \preceq^u q$.*
3. *If $p \preceq^i_{\mathcal{P}} q$ then $p \preceq^i q$.*

As expected, the converse of the previous result does not hold.

Proposition 8. *Let I and O be countable disjoint sets of inputs and outputs, respectively. There exist processes $p, q \in pTockLTS(I, O)$ such that:*

1. *$p \preceq q$ but $p \preceq_{\mathcal{P}} q$ does not hold.*
2. *$p \preceq^u q$ but $p \preceq^u_{\mathcal{P}} q$ does not hold.*
3. *$p \preceq^i q$ but $p \preceq^i_{\mathcal{P}} q$ does not hold.*

The previous probabilistic implementation relations allowed an SUT to show less behaviours that the specification. In this case, in particular if we have *missing* outputs, it makes sense that the remaining options have a higher probability and this fact motivates the definition of $\preceq_{\mathcal{P}}$ and its two variants. However, under some circumstances we require that all of the specified outputs are possible and these must have the probabilities given in the model. To see this, consider the running example. Here, navigation is probabilistic and it is important that all moves that are possible in the model are also possible in the SUT: otherwise a valid implementation might choose to stay in a room indefinitely or to cycle between only a few of the rooms. These considerations lead to a new set of implementation relations.

Definition 17. Let A, B be two sets of probabilistic traces. We write $A \sqsubseteq B$ if for all $(\sigma, \pi) \in A$ we have that either $\pi = 0$ or $(\sigma, \pi) \in B$.

Let p and q be two ptockLTSs. We say that p strictly conforms to q under *probabilistic timed trace inclusion* if and only if $\mathcal{P}\texttt{traces}(p) \sqsubseteq \mathcal{P}\texttt{traces}(q)$. We denote this $p \lesssim_{s\mathcal{P}} q$.

The following result, whose proof is immediate from Definition 17, shows the main properties of the $\lesssim_{s\mathcal{P}}$ implementation relation.

Proposition 9. *The strict probabilistic timed trace inclusion relation $\lesssim_{s\mathcal{P}}$ is reflexive and transitive but need not be symmetric or anti-symmetric.*

Next, we can adapt Definition 16 to the new implementation relation.

Definition 18. Let p and q be two ptockLTSs. We say that $p \lesssim^u_{s\mathcal{P}} q$ if and only if for all $(\sigma, \pi) \in \mathcal{P}\texttt{traces}(p)$ with $\pi > 0$ either $(\sigma, \pi) \in \mathcal{P}\texttt{traces}(q)$ or there exists a prefix $\sigma_1?i$ of σ, with $?i \in I$, such that $(\sigma_1, \pi_1), (\sigma_1?i_1, \pi_2) \in \mathcal{P}\texttt{traces}(q)$ with $\pi_1 > \pi_2$.

We say that $p \lesssim^i_{s\mathcal{P}} q$ if and only if for all $(\sigma, \pi) \in \mathcal{P}\texttt{traces}(p)$ with $\pi > 0$ either $(\sigma, \pi) \in \mathcal{P}\texttt{traces}(q)$ or there exists a prefix $\sigma_1?i$ of σ, with $?i \in I$, such that $(\sigma_1, \pi_1), (\sigma_1?i_1, 0) \in \mathcal{P}\texttt{traces}(q)$ with $\pi_1 > 0$.

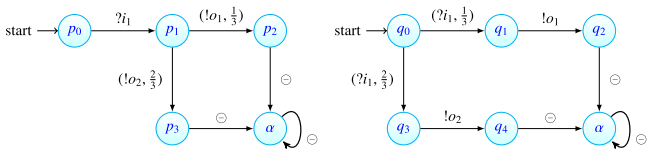

Fig. 7. Models related by probabilistic timed (refusal) trace inclusion.

The following result shows the relation between $\preceq_{\mathcal{P}}$ and $\lesssim_{s\mathcal{P}}$ and their variants. The proof is trivial.

Proposition 10. *Let p, q be ptockLTSs. We have that*

1. *$p \lesssim_{s\mathcal{P}} q$ implies $p \preceq_{\mathcal{P}} q$ but the converse might not hold.*
2. *$p \lesssim^{u}_{s\mathcal{P}} q$ implies $p \preceq^{u}_{\mathcal{P}} q$ but the converse might not hold.*
3. *$p \lesssim^{i}_{s\mathcal{P}} q$ implies $p \preceq^{i}_{\mathcal{P}} q$ but the converse might not hold.*

Although the implementation relations defined in this section appropriately take into account probabilistic information, it is interesting to study whether we can strengthen them by allowing refusals to be observed and obtain finer implementation relations. The following example illustrates the type of behaviours that we might try to distinguish.

Example 9. Let us consider the models given in Fig. 7. We have that p and q are related under all the implementation relations previously presented. In particular, they have the same probabilistic timed traces. We have a similar situation for r and s in Fig. 8. However, the intuitive behaviour of these pairs of processes is different. On the one hand, we expect p and q to be related in both directions of all the implementation relations presented in this paper because outputs cannot be controlled by the environment. In other words, a choice between outputs should work exactly as the corresponding internal choice between inputs. For example, even though $!o_1$ and $!o_2$ are simultaneously available at p_1, a user cannot choose which of them will be performed. On the other hand, r and s should not be equivalent. The issue is that a user of these systems should be able to choose between different available inputs as is the case, for instance, in a vending machine. However, the sets of available inputs differ depending on how the internal choice appearing in states r_1 and s_1 is resolved. For example, from r_1 and after performing $!o_1$, with probability $\frac{1}{2}$ we will reach a state where neither $?i_3$ nor $?i_4$ are available, while this situation has probability 0 if we start from s_1. The implementation relations that we present in the next section address this issue.

6. Implementation relations with refusals and probabilities

Finally, we consider implementation relations where refusals can be observed. We take as a starting point the implementation relations defined in the previous section. First, we consider the extension with refusals of $\preceq_{\mathcal{P}}$, the relation where we allowed the SUT to exhibit a subset of the behaviours of the specification.

Definition 19. Let p and q be two ptockLTSs. We say that p conforms to q under *probabilistic timed refusal trace inclusion* if and only if $\mathcal{PR}\mathtt{traces}(p) \preccurlyeq \mathcal{PR}\mathtt{traces}(q)$. We denote this $p \preceq_{\mathcal{PR}} q$.

Observe that the above provides the natural extension of Definition 15, produced by moving from $\mathcal{P}\mathtt{traces}(p)$ to $\mathcal{PR}\mathtt{traces}(p)$.

We now compare models with variants of the classical set inclusion operator. Since we can only 'lose' traces by 'losing' inputs and the refusal of inputs is observable in a state where tock is available, one might ask whether it is possible to have $\mathcal{PR}\mathtt{traces}(p) \subseteq \mathcal{PR}\mathtt{traces}(q)$ but not $\mathcal{PR}\mathtt{traces}(p) = \mathcal{PR}\mathtt{traces}(q)$. In other words, can we defined two models p and q such that $\mathcal{PR}\mathtt{traces}(p) \subset \mathcal{PR}\mathtt{traces}(q)$? The answer is positive. Consider s and t given in Fig. 8. We have $\mathcal{PR}\mathtt{traces}(s) \subset \mathcal{PR}\mathtt{traces}(t)$. Although these two processes have the same probabilistic timed traces, there is an important distinction concerning refusals. Specifically, we can refuse any non-empty set of inputs at state t_3 while this is not the case if we consider the sets that can be refused at s_2 and s_3.

We now examine this implementation relation before considering alternatives that treat unspecified inputs in different ways.

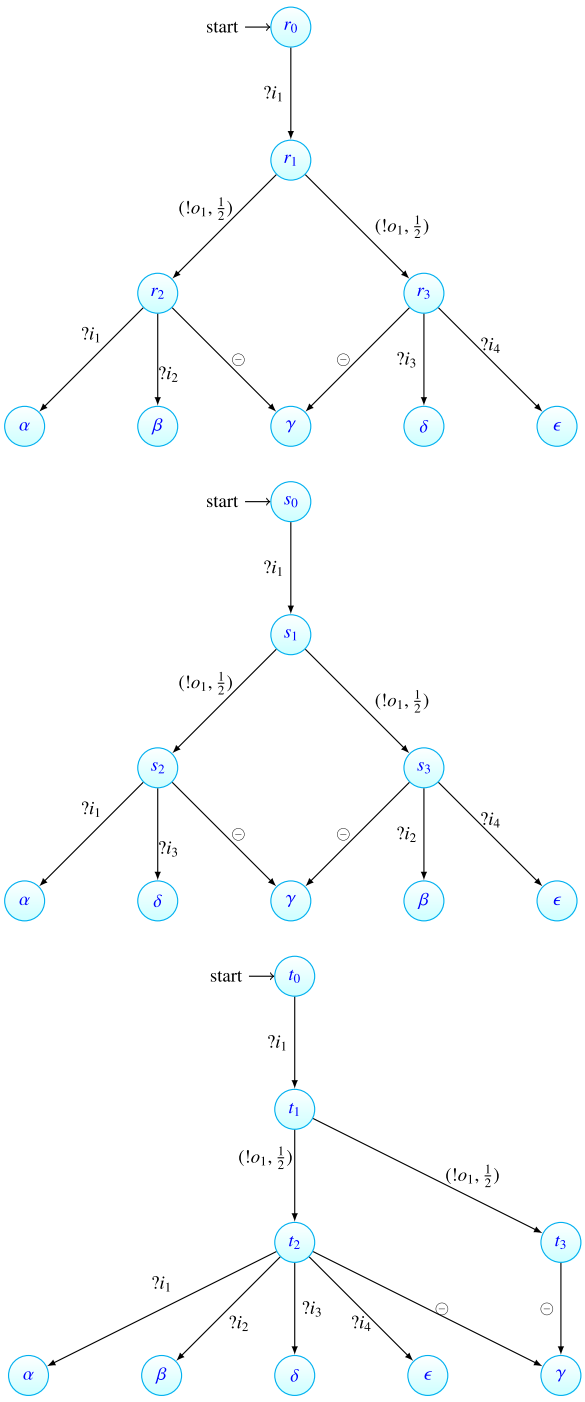

Fig. 8. Models unrelated by probabilistic timed refusal trace inclusion.

First, we present an example showing some relations between models. As part of this, we see why one cannot define a suitable implementation relation in terms of the maximal refusals.

Example 10. Consider again the models p and q given in Fig. 7. We cannot add refusals to traces in states p_1, q_1 and q_3 because they are not stable. Therefore, we have $\mathcal{PR}\texttt{traces}(p) = \mathcal{PR}\texttt{traces}(q)$ and this implies $p \preceq_{\mathcal{PR}} q$ and $q \preceq_{\mathcal{PR}} p$.

Consider now r, s and t given in Fig. 8. We have that these three processes have the same probabilistic timed traces. Therefore, they are equivalent under all the implementation relations defined in Sections 4 and 5. Assuming that $I = \{?i_1, ?i_2, ?i_3, ?i_4\}$ and $O = \{!o_1\}$, we have the following sets[4] of probabilistic timed refusal traces:

$$\mathcal{PR}\texttt{traces}(r) = \left\{\begin{array}{l}(?i_1!o_1\ominus, 1),\\ (?i_1!o_1R(\emptyset)\ominus, 1),\\ (?i_1!o_1R(\{?i_1\})\ominus, \frac{1}{2}),\\ (?i_1!o_1R(\{?i_2\})\ominus, \frac{1}{2}),\\ (?i_1!o_1R(\{?i_1, ?i_2\})\ominus, \frac{1}{2}),\\ \ldots\end{array}\right\}$$

$$\mathcal{PR}\texttt{traces}(s) = \left\{\begin{array}{l}(?i_1!o_1\ominus, 1),\\ (?i_1!o_1R(\emptyset)\ominus, 1),\\ (?i_1!o_1R(\{?i_1\})\ominus, \frac{1}{2}),\\ (?i_1!o_1R(\{?i_3\})\ominus, \frac{1}{2}),\\ (?i_1!o_1R(\{?i_1, ?i_3\})\ominus, \frac{1}{2}),\\ \ldots\end{array}\right\}$$

$$\mathcal{PR}\texttt{traces}(t) = \left\{\begin{array}{l}(?i_1!o_1\ominus, 1),\\ (?i_1!o_1R(\emptyset)\ominus, 1),\\ (?i_1!o_1R(\{?i_1\})\ominus, \frac{1}{2}),\\ (?i_1!o_1R(\{?i_2\})\ominus, \frac{1}{2}),\\ (?i_1!o_1R(\{?i_1, ?i_2\})\ominus, \frac{1}{2}),\\ (?i_1!o_1R(\{?i_1\})\ominus, \frac{1}{2}),\\ (?i_1!o_1R(\{?i_3\})\ominus, \frac{1}{2}),\\ (?i_1!o_1R(\{?i_1, ?i_3\})\ominus, \frac{1}{2}),\\ (?i_1!o_1R(\{?i_1, ?i_2, ?i_3\})\ominus, \frac{1}{2}),\\ (?i_1!o_1R(\{?i_1, ?i_2, ?i_4\})\ominus, \frac{1}{2}),\\ \ldots\end{array}\right\}$$

It is easy to check that we have neither $r \preceq_{\mathcal{PR}} s$ nor $s \preceq_{\mathcal{PR}} r$. We have $\mathcal{PR}\texttt{traces}(r) \subseteq \mathcal{PR}\texttt{traces}(t)$, so that $r \preceq_{\mathcal{PR}} t$ but the converse is not the case. For example, $(?i_1!o_1R(\{?i_1, ?i_3\})\ominus, \frac{1}{2}) \in \mathcal{PR}\texttt{traces}(t)$ but we have $(?i_1!o_1R(\{?i_1, ?i_3\})\ominus, 0) \in \mathcal{PR}\texttt{traces}(r)$. Therefore, $\mathcal{PR}\texttt{traces}(t) \preccurlyeq \mathcal{PR}\texttt{traces}(r)$ does not hold and we conclude that $t \preceq_{\mathcal{PR}} r$ does not hold. Similarly, we have $\mathcal{PR}\texttt{traces}(s) \subseteq \mathcal{PR}\texttt{traces}(t)$ bu $t \preceq_{\mathcal{PR}} s$ does not hold.

We can use r and t to show why we cannot restrict ourselves to only computing the maximal refusal sets. If we were to do this, the timed probabilistic refusal traces of r would be

$$\left\{\begin{array}{l}(?i_1!o_1\ominus, 1),\\ (?i_1!o_1R(\{?i_1, ?i_2\})\ominus, \frac{1}{2}),\\ (?i_1!o_1R(\{?i_3, ?i_4\})\ominus, \frac{1}{2}),\\ \ldots\end{array}\right\}$$

but the ones corresponding to t would be

$$\left\{\begin{array}{l}(?i_1!o_1\ominus, 1),\\ (?i_1!o_1R(\{?i_1, ?i_2, ?i_3, ?i_4\})\ominus, \frac{1}{2}),\\ \ldots\end{array}\right\}$$

and we would no longer have the $\preccurlyeq$ relation between the traces of r and the ones of t.

The proof of the following result is immediate from Definition 19.

Proposition 11. *The probabilistic timed trace inclusion relation $\preceq_{\mathcal{PR}}$ is reflexive and transitive but need not be symmetric or anti-symmetric.*

Next, we present the adaptions of the implementation relations introduced in Definition 16.

Definition 20. Let p and q be two ptockLTSs. We say that $p \preceq^u_{\mathcal{PR}} q$ if and only if for all $(\sigma, \pi) \in \mathcal{PR}\texttt{traces}(p)$ with $\pi > 0$ either $(\sigma, \pi') \in \mathcal{PR}\texttt{traces}(q)$ with $\pi \geq \pi' > 0$ or there exists a prefix $\sigma_1?i$ of σ, with $?i \in I$, such that $(\sigma_1, \pi_1), (\sigma_1?i_1, \pi_2) \in \mathcal{PR}\texttt{traces}(q)$ with $\pi_1 > \pi_2$.

We say that $p \preceq^i_{\mathcal{PR}} q$ if and only if for all $(\sigma, \pi) \in \mathcal{PR}\texttt{traces}(p)$ with $\pi > 0$ either $(\sigma, \pi') \in \mathcal{PR}\texttt{traces}(q)$ with $\pi \geq \pi' > 0$ or there exists a prefix $\sigma_1?i$ of σ, with $?i \in I$, such that $(\sigma_1, \pi_1), (\sigma_1?i_1, 0) \in \mathcal{PR}\texttt{traces}(q)$ with $\pi_1 > 0$.

We can adapt Theorem 5 to consider these last three implementation relations. The proof of the result follows in the same way as the one corresponding to Theorem 3.

Theorem 7. *Let $p = (Q, q_0, I, O, T)$ and $q = (Q', q'_0, I, O, T')$ be ptockLTSs. Then we have:*

1. *If $p \preceq_{\mathcal{PR}} q$ then we also have both $p \preceq^u_{\mathcal{PR}} q$ and $p \preceq^i_{\mathcal{PR}} q$.*
2. *Neither $p \preceq^u_{\mathcal{PR}} q$ nor $p \preceq^i_{\mathcal{PR}} q$ implies $p \preceq_{\mathcal{PR}} q$.*
3. *If $p \preceq^i_{\mathcal{PR}} q$ then we also have $p \preceq^u_{\mathcal{PR}} q$.*
4. *$p \preceq^u_{\mathcal{PR}} q$ does not imply $p \preceq^i_{\mathcal{PR}} q$.*

We also adapt Theorem 6 to the new setting where refusals are taken into account.

Theorem 8. *Let $p = (Q, q_0, I, O, T)$ and $q = (Q', q'_0, I, O, T')$ be ptockLTSs. If q is input-enabled then the following three statements are equivalent: $p \preceq_{\mathcal{PR}} q$; $p \preceq^u_{\mathcal{PR}} q$; and $p \preceq^i_{\mathcal{PR}} q$.*

We can now compare the implementation relations previously defined with those introduced in Section 5. The proof of the following result easily follows from the fact that if we have $\mathcal{PR}\texttt{traces}(p) \subseteq \mathcal{PR}\texttt{traces}(q)$ and remove those traces with occurrences of refusals we trivially have $\mathcal{P}\texttt{traces}(p) \subseteq \mathcal{P}\texttt{traces}(q)$.

Proposition 12. *Let $p = (Q, q_0, I, O, T)$ and $q = (Q', q'_0, I, O, T')$ be ptockLTSs. Then the following hold.*

1. *If $p \preceq_{\mathcal{PR}} q$ then $p \preceq_{\mathcal{P}} q$.*
2. *If $p \preceq^u_{\mathcal{PR}} q$ then $p \preceq^u_{\mathcal{P}} q$.*
3. *If $p \preceq^i_{\mathcal{PR}} q$ then $p \preceq^i_{\mathcal{P}} q$.*

As expected, the converse of the previous result does not hold.

Proposition 13. *There exist ptockLTSs p and q such that:*

1. *$p \preceq_{\mathcal{P}} q$ but $p \preceq_{\mathcal{PR}} q$ does not hold.*
2. *$p \preceq^u_{\mathcal{P}} q$ but $p \preceq^u_{\mathcal{PR}} q$ does not hold.*
3. *$p \preceq^i_{\mathcal{P}} q$ but $p \preceq^i_{\mathcal{PR}} q$ does not hold.*

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

Proof. In order to prove this it is sufficient to give an example of such ptockLTSs. Consider r and s depicted in Fig. 8. It is obvious that $\mathcal{P}\texttt{traces}(r) = \mathcal{P}\texttt{traces}(s)$. Therefore, $s \preceq_{\mathcal{P}} r$. However, as explained in Example 10, we do not have $s \preceq_{\mathcal{PR}} r$. Similarly, $s \preceq^{u}_{\mathcal{P}} r$ but $s \preceq^{u}_{\mathcal{PR}} r$ does not hold and $s \preceq^{i}_{\mathcal{P}} r$ but $s \preceq^{i}_{\mathcal{PR}} r$ does not hold.

The combination of Propositions 12 and 13 show that if we can observe refusals then we have more powerful implementation relations than the ones we obtain when we only consider probabilistic timed traces.

Finally, we present adaptions of Definitions 17 and 18 to include refusals.

Definition 21. Let p and q be two ptockLTSs. We say that p strictly conforms to q under *probabilistic timed refusal trace inclusion* if and only if $\mathcal{PR}\texttt{traces}(p) \sqsubseteq \mathcal{PR}\texttt{traces}(q)$. We denote this $p \lesssim_{s\mathcal{PR}} q$.

We write $p \lesssim^{u}_{s\mathcal{PR}} q$ if and only if for all $(\sigma, \pi) \in \mathcal{PR}\texttt{traces}(p)$ with $\pi > 0$ either $(\sigma, \pi) \in \mathcal{PR}\texttt{traces}(q)$ or there exists a prefix $\sigma_1?i$ of σ, with $?i \in I$, such that $(\sigma_1, \pi_1), (\sigma_1?i_1, \pi_2) \in \mathcal{PR}\texttt{traces}(q)$ with $\pi_1 > \pi_2$.

We write $p \lesssim^{i}_{s\mathcal{PR}} q$ if and only if for all $(\sigma, \pi) \in \mathcal{PR}\texttt{traces}(p)$ with $\pi > 0$ either $(\sigma, \pi) \in \mathcal{PR}\texttt{traces}(q)$ or there exists a prefix $\sigma_1?i$ of σ, with $?i \in I$, such that $(\sigma_1, \pi_1), (\sigma_1?i_1, 0) \in \mathcal{PR}\texttt{traces}(q)$ with $\pi_1 > 0$.

The relation between the implementation relations presented in this section is given in the following result whose proof is trivial.

Proposition 14. *Let p, q be ptockLTSs. We have that*

1. *$p \lesssim_{s\mathcal{PR}} q$ implies $p \preceq_{\mathcal{PR}} q$ but the converse might not hold.*
2. *$p \lesssim^{u}_{s\mathcal{PR}} q$ implies $p \preceq^{u}_{\mathcal{PR}} q$ but the converse might not hold.*
3. *$p \lesssim^{i}_{s\mathcal{PR}} q$ implies $p \preceq^{i}_{\mathcal{PR}} q$ but the converse might not hold.*

7. Alternative characterisation based on observers

The earlier sections defined alternative implementation relations that varied in terms of what could be observed: they were informed by a notion of what the environment (e.g. a tester) could observe when interacting with the SUT. The idea is that we should not reject an SUT if, to the environment, its behaviours are consistent with those of the specification. We therefore aimed to describe the ability of the environment to distinguish between models. Here, we define the notion of an observer, that interacts with the SUT, and show that this leads naturally to the richest notion of observation we consider, where we have probabilities, time and refusals.

We build upon the use of observers for cyclic timed testing in which there are no probabilities [14]. An observer will be an LTS that interacts with a ptockLTS through a definition of synchronisation and, in doing so, it will capture a type of observation that can be made. Below we define observers and also how they interact with ptockLTSs but first we explain the approach we take.

As noted, observers are LTSs and their transitions are defined so that they mimic the observations that an environment might make. One approach would be to extend observers to include (expected/required) probabilities within transitions: the label of a transition would be a pair (a, π) for an event a and probability π. This would suggest that a tester could determine, or estimate, the probability of a particular event at a given point in a computation. However, if a tester wanted to estimate the probability of an event or transition at some point in a computation, a tester would have to observe an event, take the system back to the previous state, and repeat this process multiple times. Normally this cannot be done in testing and so, although we could include expected (required) probabilities within the definition of an observer, the resultant observers would not represent the ability of a tester to make observations regarding a system and so such a definition of an observer would be unsuitable.

As a result, we take a different approach in which one does not include probabilities within the definition of an observer and instead reasons about the probability of an observer making a particular observation. We can therefore reuse the previous definition of an observer [14], for cyclic systems without probabilities; differences will appear in the definition of how an observer and a ptockLTS interact.

Similar to the approach taken with ioco [18], observers include a special action θ to denote a refusal being observed.

Definition 22 (*Observer [14]*)**.** Let I and O be countable disjoint sets of inputs and outputs, respectively. An *observer* $u = (Q, q_0, I \cup O \cup \{\ominus, \theta\}, T)$ is an LTS that satisfies the following properties for each state $q \in Q$:

1. If $q \overset{\ominus}{\not\rightarrow}$ and there is at least one event $a \in I \cup O$ such that $q \overset{a}{\rightarrow}$ then for all $a \in O$ we have that $q \overset{a}{\rightarrow}$.
2. There exists at most one $q' \in Q$ such that $q \overset{\ominus}{\rightarrow} q'$.
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 if a refusal is observed (that is, if a θ transition is performed) then the next event must be a $\ominus$; this reflects the observation of a refusal taking time. The other rules ensure that outputs can always be observed if the observer has not terminated: for output $!o$ and state q, if there are transitions from q then there must be a transition with label $!o$ from q. The exception to this rule is where a $\ominus$ is possible: the synchronisation rules below will ensure that such a state can only be reached if the SUT cannot engage in any outputs.

It would be straightforward to update the definition of an observer to make it alternating. However, this would lead to a more complicated definition and would not be required; the alternating nature of ptockLTSs plus the definition of synchronisation given below ensure that if, for example, the ptockLTS is in an input state then outputs cannot be observed irrespective of the state of the observer.

Example 11. In Fig. 9 we present an observer for the ptockLTS mailing robot model from Fig. 4. It is worth mentioning that there are many possible observers and the graphical representation from Fig. 9 captures only a part of an observer. For example, the states 12, 17 or 18 could be further expanded. Also, due to space constraints, all the transitions labelled by output symbols, departing from the states 9 and 10 have been omitted and replaced in the figure with dots. The example highlights the observation of a refusal (the θ-labelled transitions), time passing ($\ominus$-transition, that is the only one possible after a refusal) and acceptance of all outputs.

The composition of an observer and a ptockLTS will define an LTS rather than a ptockLTS. We therefore first extend the notion of an LTS to include probabilities.

Definition 23 (*p-LTS*)**.** A *p-LTS* is a tuple $p = (Q, q_0, L, T)$ where

- 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;
- $T \subseteq Q \times (\{\ominus\} \cup (L \times (0, 1])) \times Q$ is the transition relation.

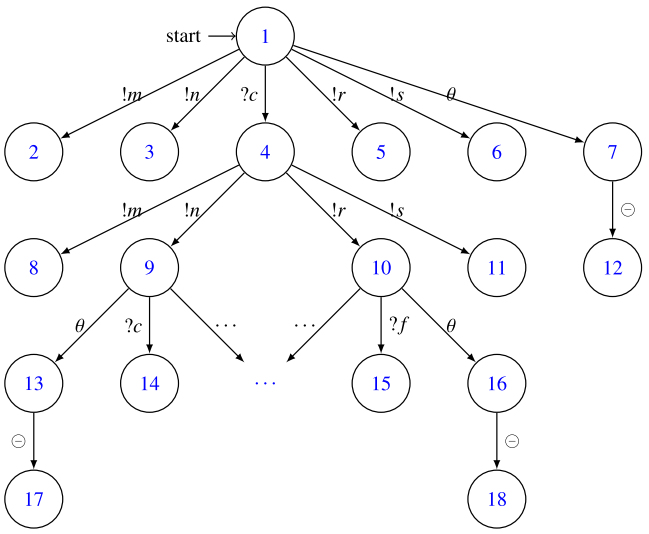

Fig. 9. Observer example.

We define the language of p, denoted by $L(p)$, as the set of (finite) sequences of actions $\{\sigma \in (L \cup \{\ominus\})^* | q_0 \stackrel{\sigma}{\Longrightarrow}\}$.

Note that $L(p)$ includes the set of sequences of visible actions and tocks that p can perform; probabilities are discarded. It is trivial to adapt Definition 4 to the new formalism so that we can compute the probability with which a p p-LTS performs a certain trace σ. We denote this probability by $prob(p, \sigma)$.

We now define a parallel composition operator, $\pitchfork$, that defines how a ptockLTS and an observer interact. Here, we include probabilities; since observers are not probabilistic, the probabilities come from the ptockLTS. Since p-LTSs include probabilities for all events except $\ominus$, we will need to consider a probability whenever we include the observation of a refusal; clearly this probability is always 1.

Definition 24 (*Synchronised Parallel Communication*)**.** Let I and O be countable disjoint sets of inputs and outputs, respectively. Let $p = (Q, q_0, I, O, T) \in pTockLTS(I, O)$ and $u = (Q', q'_0, I \cup O \cup \{\ominus, \theta\}, T') \in \mathcal{U}(I, O)$. The composition of the observer u and the model p, denoted by $u \pitchfork p$, is a p-LTS $(Q \times Q', (q_0, q'_0), I \cup O \cup \mathcal{R}(I), T'')$ in which T'' is defined as follows:

- If $(q_1, (a, \pi), q_2) \in T$ and $(q'_1, a, q'_2) \in T'$, with $a \in I \cup O$, then we have $((q_1, q'_1), (a, \pi), (q_2, q'_2)) \in T''$.
- If $(q_1, \ominus, q_2) \in T$ and $(q'_1, \ominus, q'_2) \in T'$, then we have $((q_1, q'_1), \ominus, (q_2, q'_2)) \in T''$.
- Let $X \subseteq I$. If $(q_1, \ominus, q_2) \in T$ and $(q'_1, \theta, q'_2) \in T'$ then $((q_1, q'_1), (R(X), 1), (q_1, q'_2)) \in T''$ if and only if the following conditions hold:
  - for all $a \in I$ 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 $(q'_1, a, q'_3) \in T'$.
  - for all $a \in X$ we have that there exists q'_3 such that $(q'_1, a, q'_3) \in T'$.

The application of an observer u to a ptockLTS can observe the following set of traces:

$$obs^{\theta}(u, p) = L(u \pitchfork p)$$

As one would expect, in the last rule, since q_1 may evolve via $\ominus$ and outputs are urgent in ptockLTSs, no outputs are enabled in q_1. As a result, the state of p does not change. The conditions associated with the third bullet, where refusals are produced, indicate, on the one hand, that the model and the observer cannot synchronise and, on the other hand, that all the actions included in the refusal X can be performed by the observer. Note that as a result of the first condition, if an action can be performed by the observer then it cannot be performed by the model and, therefore, the model refuses all the actions belonging to X.

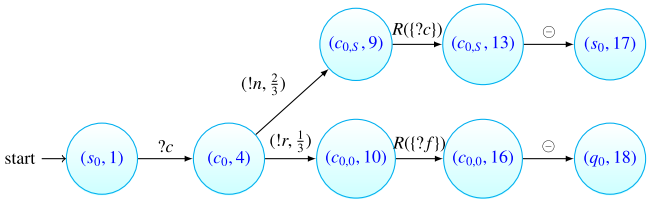

Fig. 10. Composition of a tockLTS with an observer.

Example 12. Fig. 10 illustrates a *fragment* of the composition of the observer given in Example 11 (see Fig. 9) with the ptockLTS model from Fig. 4. It aims to highlight the observation of a refusal, followed by time passing.

We now show how the observations that can be made by an observer relate to the semantics of the ptockLTS. The following is a direct consequence of the definition of $u \pitchfork p$; a proof follows easily by induction on the length of the trace considered.

Proposition 15. *Let I and O be countable disjoint sets of inputs and outputs, respectively. Given $\sigma \in (I \cup O \cup \{\ominus\} \cup \mathcal{R}(I))^*$ and p in $pTockLTS(I, O)$, there exists 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. In addition, if this is the case then $prob(p, \sigma) = prob(u \pitchfork p, \sigma)$.*

It is then possible to rephrase implementation relations in terms of the observations that can be made. The following is one such result, which is immediate from Proposition 15; similar results hold for the other implementation relations.

Theorem 9. *Let $p, q \in pTockLTS(I, O)$, for countable sets I and O. We have that $p \lesssim_{SPR} q$ if and only if for all $u \in \mathcal{U}(I, O)$ we have that the following hold:*

1. *$obs^{\theta}(u, p) \subseteq obs^{\theta}(u, q)$; and*
2. *for all $\sigma \in obs^{\theta}(u, p)$ we have that $prob(u \pitchfork p, \sigma) = prob(u \pitchfork q, \sigma)$.*

The above results tell us both that probabilistic timed refusal traces correspond to observations that could be made in testing and that our implementation relations are appropriate extensions of previously defined implementation relations.

Note that there is an implicit assumption that an environment (tester) can determine the true probability of a trace. However, the best one can actually do is to estimate the probability of a trace through experiments. We are working on a testing framework where this idea is implemented and we will provide additional details when we discuss lines for future work.

8. Conclusions and future work

If one has a formal model that describes the allowed behaviours of the SUT, then this model can form the basis of testing and this observation has led to the development of many test generation techniques. There has been particular interest in testing from transition systems, with a number of resultant techniques and tools. However, in order to test from a formal model in

a manner that is guaranteed to be sound, it is necessary to first define a suitable implementation relation between SUTs and specifications. This paper concentrated on defining and comparing implementation relations for a class of (cyclic state-based) models, motivated by their use in robotics.

The motivation for the work was the use of state-based models to model the required behaviour of the control software of robotic systems. We took particular inspiration from the RoboChart and RoboSim languages, which define cyclic state-based models and use a notation similar to that used by those developing robotic software. There is a tool (RoboTool [17]) that will map a model in RoboChart or RoboSim to a tock-CSP model. In turn, a tock-CSP model can be given a formal semantics as a labelled transition system that is probabilistic. There is then the potential to automate the generation of sound test cases that can be used to test the SUT within simulation. However, it is first necessary to define suitable implementation relations that define what behaviours (SUTs) are considered to be correct implementations of a model.

We started by defining the notion of a ptockLTS, which makes it possible to include probabilities and discrete time within cyclic models. The basic implementation relations were defined in terms of the possible traces of such models. These implementation relations are therefore suitable for circumstances in which one cannot determine the probability of an observation/trace. This is the case, for example, in both run-time monitoring and passive testing, since for both of these one passively observes the behaviour of a system but one cannot, for example, repeatedly apply a test case.

Having defined implementation relations in terms of traces, we then considered scenarios in which a tester can make a richer set of observations. First, we allowed probabilities to be considered and this makes sense if a tester is able to repeatedly apply a test case and then estimate the probabilities of observations made. Here, there were a number of ways in which one could treat the situation in which there is more than one allowed output in a given state. We considered two scenarios: the first was where these different outputs represent valid alternatives and it was not necessary to include all; the second was when a correct system must implement all of the allowed outputs. This led to groups of implementation relations, with the choice of implementation relation to use being dependent on the scenario. Finally, we extended the probabilistic implementation relations to consider the situation in which a tester can observe that certain inputs are not allowed by the SUT, with this being represented in terms of refusals. For example, a sensor or sub-system might be switched off or a field unavailable.

Having defined implementation relations, we have the question of whether they are suitable: whether the types of behaviours they considered might actually be observed by a tester. In order to validate this we defined the notion of an observer, with the resultant set of observations being exactly what one would expect.

The focus on this paper has been on defining suitable implementation relations and showing how these relate. The practical benefit is that the implementation relations can form the basis for sound, systematic testing based on models. The testing process becomes sound if a suitable implementation relation is used: if a failure is observed during testing then it must be a consequence of the SUT not being a correct implementation of the specification.

Now that we have implementation relations, it is possible to devise test generation algorithms based on these and this is one of our main lines of current and future work on the application of formal approaches to robotics. Test cases will not have probabilistic information. At each state, a test case might offer an input, observe a refusal, or let time pass. In addition, a test case must always be ready to observe an output produced by the SUT. There are at least two possible approaches to automated test generation: either one randomly generates test cases that are guaranteed to be sound, as is done for implementation relations such as ioco [18], or one uses a fault-based approach and generates test cases that are guaranteed to detect certain faults. For the latter approach, we intend to extend our fault-based technique that was developed for models that do not include probabilities. This technique involved using a tool to insert possible faults into a model and then another tool (a model-checker) to automatically generate test cases guaranteed to find such faults [12].

The next problem that we will tackle is to define how test cases are applied to the SUT. If we are testing deterministic systems, then it is enough to apply each test case once. If the system is non-deterministic, then the test case must be applied several times and it is necessary to use a *fairness condition* to assume that all potential paths are eventually traversed. In the probabilistic case, we do not need to consider a fairness condition but need to apply the same test case several times. Having done this, it is possible to produce estimates of probabilities based on the frequencies with which the different sequences of actions are observed. Then, we need to perform hypothesis contrasts to decide, up to a confidence degree, whether the estimated probability distributions conform to the specified probability distributions.

There are several additional possible lines of future work. First, once test generation techniques have been developed, there is a need to evaluate these in the testing of robotic software in simulation. As previously noted, there may be potential to use testing within simulation as the basis for generating test cases for deployment testing. One potential approach is to use fault seeding (mutation) and find test cases that expose seeded faults within simulation (see, for example, [12]): it may then be possible to use these as the starting point for generating test cases that would find corresponding faults in deployed systems. An additional avenue of future work is to incorporate information about the environment in order to avoid generating 'unrealistic' test cases. In addition, there may be scope to develop techniques that map a (class of) ptockLTS to a probabilistic finite state machine and then utilise test generation techniques that have been developed for such models.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

[1] Q. Song, E. Engström, P. Runeson, Concepts in testing of autonomous systems: Academic literature and industry practice, in: 1st IEEE/ACM Workshop on AI Engineering - Software Engineering for AI, WAIN@ICSE'21, IEEE, 2021, pp. 74–81.

[2] C.S. Timperley, A. Afzal, D.S. Katz, J.M. Hernandez, C. Le Goues, Crashing simulated planes is cheap: Can simulation detect robotics bugs early? in: 11th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST'18, IEEE, 2018, pp. 331–342.

[3] A. Afzal, C. Le Goues, M. Hilton, C.S. Timperley, A study on challenges of testing robotic systems, in: 13th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST'20, IEEE, 2020, pp. 96–107.

[4] E.P. Moore, Gedanken experiments on sequential machines, in: C. Shannon, J. McCarthy (Eds.), Automata Studies, Princeton University Press, 1956.

[5] J. Peleska, Model-based avionic systems testing for the airbus family, in: 23rd IEEE European Test Symposium, ETS'18, IEEE Computer Society, 2018, pp. 1–10.

[6] M. Shafique, Y. Labiche, A systematic review of state-based test tools, Int. J. Softw. Tools Technol. Transfer 17 (1) (2015) 59–76.
[7] W. Grieskamp, N. Kicillof, K. Stobie, V. Braberman, Model-based quality assurance of protocol documentation: tools and methodology, Softw. Test. Verif. Reliab. 21 (1) (2011) 55–71.
[8] 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, H. Zedan, Using formal specifications to support testing, ACM Comput. Surv. 41 (2) (2009) 9:1–9:76.
[9] A. Miyazawa, A. Cavalcanti, P. Ribeiro, W. Li, J. Woodcock, J. Timmis, RoboChart Reference Manual, Technical Report, University of York, 2019.
[10] A. Cavalcanti, P. Ribeiro, A. Miyazawa, A. Sampaio, M.C. Filho, A. Didier, RoboSim Reference Manual, Technical Report, University of York, 2019, URL https://www.cs.york.ac.uk/circus/RoboCalc/robosim/robosim-reference.pdf.
[11] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice Hall, 1998.
[12] A. Cavalcanti, J. Baxter, R.M. Hierons, R. Lefticaru, Testing robots using CSP, in: 13th Int. Conf. on Tests and Proofs, TAP'19, LNCS 11823, Springer, 2019, pp. 21–38.
[13] R. Lefticaru, R.M. Hierons, 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, Springer, 2019, pp. 393–409.
[14] R. Lefticaru, R.M. Hierons, M. Núñez, Implementation relations and testing for cyclic systems with refusals and discrete time, J. Syst. Softw. 170 (2020) 110738:1–20.
[15] K. Ye, A. Cavalcanti, S. Foster, A. Miyazawa, J. Woodcock, Probabilistic modelling and verification using RoboChart and PRISM, Softw. Syst. Model. 21 (2) (2022) 667–716.
[16] A. Cavalcanti, J. Baxter, G. Carvalho, RoboWorld: Where can my robot work? in: 19th Int. Conf. on Software Engineering and Formal Methods, SEFM'21, LNCS 13085, Springer, 2021, pp. 3–22.
[17] J. Baxter, A. Miyazawa, P. Ribeiro, K. Ye, RoboTool - RoboChart Tool Manual, Technical Report, University of York, 2022.
[18] J. Tretmans, Model based testing with labelled transition systems, in: Formal Methods and Testing, LNCS 4949, Springer, 2008, pp. 1–38.
[19] PRISM Model Checker, PRISM lab session, part B: Mail delivery robot. Available at http://www.prismmodelchecker.org/courses/aims1617/deliveryRobot.php.
[20] A.W. Roscoe, Understanding Concurrent Systems, in: Texts in Computer Science, Springer, 2010.
[21] J. Baxter, A. Cavalcanti, M. Gazda, R.M. Hierons, Testing using CSP models: time, inputs, and outputs, ACM Trans. Comput. Log. 24 (2) (2023) 1–40.
[22] A. Cavalcanti, R.M. Hierons, S.C. Nogueira, Inputs and outputs in CSP: a model and a testing theory, ACM Trans. Comput. Log. 21 (3) (2020) 24:1–24:53.
[23] L. Brandán Briones, 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, Springer, 2004, pp. 64–78.
[24] M. Krichen, S. Tripakis, Conformance testing for real-time systems, Form. Methods Syst. Des. 34 (3) (2009) 238–304.
[25] J. Schmaltz, J. Tretmans, On conformance testing for timed systems, in: 6th Int. Conf. on Formal Modeling and Analysis of Timed Systems, FORMATS'08, LNCS 5215, Springer, 2008, pp. 250–264.
[26] L. Cheung, M. Stoelinga, F. Vaandrager, A testing scenario for probabilistic processes, J. ACM 54 (6) (2007) 29.
[27] R. Cleaveland, Z. Dayar, S.A. Smolka, S. Yuen, Testing preorders for probabilistic processes, Inform. and Comput. 154 (2) (1999) 93–148.
[28] Y. Deng, R. van Glabbeek, M. Hennessy, C. Morgan, Characterising testing preorders for finite probabilistic processes, Log. Methods Comput. Sci. 4 (4) (2008).
[29] M. Núñez, Algebraic theory of probabilistic processes, J. Log. Algebr. Program. 56 (1–2) (2003) 117–177.
[30] R.M. Hierons, M.G. Merayo, Mutation testing from probabilistic and stochastic finite state machines, J. Syst. Softw. 82 (11) (2009) 1804–1818.
[31] I. Hwang, A.R. Cavalli, Testing a probabilistic FSM using interval estimation, Comput. Netw. 54 (7) (2010) 1108–1125.
[32] N. López, M. Núñez, I. Rodríguez, Specification, testing and implementation relations for symbolic-probabilistic systems, Theoret. Comput. Sci. 353 (1–3) (2006) 228–248.
[33] M. Gerhold, M. Stoelinga, Model-based testing of probabilistic systems, Form. Asp. Comput. 30 (1) (2018) 77–106.
[34] R.M. Hierons, M. Núñez, Using schedulers to test probabilistic distributed systems, Form. Asp. Comput. 24 (4–6) (2012) 679–699.
[35] R.M. Hierons, M. Núñez, Implementation relations and probabilistic schedulers in the distributed test architecture, J. Syst. Softw. 132 (2017) 319–335.
[36] I. Phillips, Refusal testing, Theoret. Comput. Sci. 50 (3) (1987) 241–284.
[37] C. Gregorio, M. Núñez, Denotational semantics for probabilistic refusal testing, in: 1st Int. Workshop on Probabilistic Methods in Verification, PROBMIV'98, ENTCS 22, Elsevier, 1999, pp. 111–137.
[38] L. Heerink, 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, Chapman & Hall, 1997, pp. 23–38.
[39] R. Michelmore, M. Wicker, L. Laurenti, L. Cardelli, Y. Gal, M. Kwiatkowska, Uncertainty quantification with statistical guarantees in end-to-end autonomous driving control, in: 27th IEEE Int. Conf. on Robotics and Automation, ICRA'20, IEEE, 2020, pp. 7344–7350.
[40] B.K. Aichernig, H. Brandl, F. Wotawa, Conformance testing of hybrid systems with qualitative reasoning models, in: 6th Int. Workshop on Formal Engineering Approaches To Software Components and Architectures, FESCA'06, ENTCS 253(2), 2009, pp. 53–69.
[41] T. Dang, Model-based testing of hybrid systems, in: Model-Based Testing for Embedded Systems, CRC Press, 2011, pp. 383–424.
[42] N. Khakpour, M.R. Mousavi, Notions of Conformance Testing for Cyber-Physical Systems: Overview and Roadmap (Invited Paper), in: 26th Int. Conf. on Concurrency Theory, CONCUR'15, in: Leibniz International Proceedings in Informatics 42, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 18–40.
[43] M. van Osch, Hybrid input-output conformance and test generation, in: 1st Combined Int. Workshops on Formal Approaches to Software Testing and Runtime Verification, FATES/RV'06, LNCS 4262, Springer, 2006, pp. 70–84.
[44] D. Harel, Statecharts: A visual formulation for complex systems, Sci. Comput. Program. 8 (3) (1987) 231–274.
[45] D. Harel, M. Politi, Modeling Reactive Systems with Statecharts: The STATEMATE Approach, McGraw-Hill, New York, 1998.
[46] N. López, M. Núñez, An overview of probabilistic process algebras and their equivalences, in: C. Baier, B.R. Haverkort, H. Hermanns, J. Katoen, M. Siegle (Eds.), Validation of Stochastic Systems - a Guide to Current Research, LNCS 2925, Springer, 2004, pp. 89–123.
[47] A. Sokolova, E.P. de Vink, Probabilistic automata: System types, parallel composition and comparison, in: C. Baier, B.R. Haverkort, H. Hermanns, J. Katoen, M. Siegle (Eds.), Validation of Stochastic Systems - a Guide to Current Research, LNCS 2925, Springer, 2004, pp. 1–43.
[48] R. van Glabbeek, S.A. Smolka, B. Steffen, Reactive, generative and stratified models of probabilistic processes, Inform. and Comput. 121 (1) (1995) 59–80.
[49] K. Larsen, A. Skou, Bisimulation through probabilistic testing, Inform. and Comput. 94 (1) (1991) 1–28.
[50] G. Fraser, F. Wotawa, P. Ammann, Issues in using model checkers for test case generation, J. Syst. Softw. 82 (9) (2009) 1403–1418.
[51] D. Peled, Model checking and testing combined, in: 30th Int. Colloquium on Automata, Languages and Programming, ICALP'03, LNCS 2719, Springer, 2003, pp. 47–63.
[52] M. Harman, P. McMinn, A theoretical and empirical study of search-based testing: Local, global, and hybrid search, IEEE Trans. Softw. Eng. 36 (2) (2010) 226–247.
[53] M.-C. Gaudel, Testing can be formal, too!, in: 6th Int. Joint Conf. CAAP/FASE, Theory and Practice of Software Development, TAPSOFT'95, LNCS 915, Springer, 1995, pp. 82–96.
[54] R.M. Hierons, Testing from partial finite state machines without harmonised traces, IEEE Trans. Softw. Eng. 43 (11) (2017) 1033–1043.
[55] R.M. Hierons, FSM quasi-equivalence testing via reduction and observing absences, Sci. Comput. Program. 177 (2019) 1–18.
[56] A. Petrenko, N. Yevtushenko, Testing from partial deterministic FSM specifications, IEEE Trans. Comput. 54 (9) (2005) 1154–1165.
[57] A. Petrenko, N. Yevtushenko, G.v. Bochmann, Testing deterministic implementations from their nondeterministic FSM specifications, in: 9th IFIP Workshop on Testing of Communicating Systems, IWTCS'96, Chapman & Hall, 1996, pp. 125–140.
[58] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, third ed., Addison-Wesley, 2006.

Manuel Núñez received a Ph.D. degree in Mathematics and an M.S. degree in Economics. He is a Professor of Computer Science with the Complutense University of Madrid, Spain. He belongs to the IEEE SMC Technical Committee on Computational Collective Intelligence; he is a member of several Editorial Boards and has served on more than 130 Programme Committees of international events in Computer Science.

Robert M. Hierons received a B.A. in Mathematics (Trinity College, Cambridge), and a Ph.D. in Computer Science (Brunel University). He then joined the Department of Mathematical and Computing Sciences at Goldsmiths College, University of London, before returning to Brunel University in 2000. He was promoted to full Professor in 2003 and joined The University of Sheffield in 2018.

Raluca Lefticaru received a B.Sc. degree in Mathematics and Computer Science, then M.Sc. and Ph.D. degrees in Computer Science. She has been a Lecturer at the University of Bucharest, Romania, and has held several research positions in the UK. She is currently an Assistant Professor at the University of Bradford and a Visiting Researcher at the University of Sheffield, UK.