Probabilistic Interface Automata

Esteban Pavese, Víctor Braberman, and Sebastian Uchitel

Abstract—System specifications have long been expressed through automata-based languages, which allow for compositional construction of complex models and enable automated verification techniques such as model checking. Automata-based verification has been extensively used in the analysis of systems, where they are able to provide yes/no answers to queries regarding their temporal properties. Probabilistic modelling and checking aim at enriching this binary, qualitative information with *quantitative* information, more suitable to approaches such as reliability engineering. Compositional construction of software specifications reduces the specification effort, allowing the engineer to focus on specifying individual component behaviour to then analyse the composite system behaviour. Compositional construction also reduces the validation effort, since the validity of the composite specification should be dependent on the validity of the components. These component models are smaller and thus easier to validate. Compositional construction poses additional challenges in a probabilistic setting. Numerical annotations of probabilistically independent events must be contrasted against estimations or measurements, taking care of not compounding this quantification with exogenous factors, in particular the behaviour of other system components. Thus, the validity of compositionally constructed system specifications requires that the validated probabilistic behaviour of each component continues to be preserved in the composite system. However, existing probabilistic automata-based formalisms do not support specification of non-deterministic and probabilistic component behaviour which, when observed through logics such as pCTL, is preserved in the composite system. In this paper we present a probabilistic extension to Interface Automata which preserves pCTL properties under probabilistic fairness by ensuring a probabilistic branching simulation between component and composite automata. The extension not only supports probabilistic behaviour but also allows for weaker prerequisites to interfacing composition, that supports delayed synchronisation that may be required because of internal component behaviour. These results are equally applicable as an extension to non-probabilistic Interface Automata.

Index Terms—Behaviour models, probability, interface automata, model checking

1 Introduction

Modelling languages are envisioned with the objective of capturing and conveying relevant aspects of a system design, many times resorting to diverse languages to describe separate aspects of the system. In the realm of software engineering in particular, many such languages have been introduced into general use, including automata-based languages which have the advantage of being simple enough to be used as a means to exhibit design and documentation, but also formal enough to be used as artefacts amenable to automated validation and verification.

Techniques that automatically explore automata-based models in order to gain increased assurance regarding the absence of errors have been investigated for some time. A notable example is model checking [1] where an exhaustive search of the model yields, in its most basic and widespread form, a yes/no response to the question of whether the model satisfies a specific property.

Although obtaining binary results from model checkers has been shown to be useful for validation and verification, when the model checker returns a negative answer this can represent insufficient information. This is acknowledged, for instance, in the software reliability community where the interest is not in whether certain properties hold, for example because they are known to be unavoidable (e.g., failures due to uncontrollable network transmissions), impractical to fix (e.g., hardware-based failures on deployed satellites, or similarly inaccessible systems), or simply because fixing them is uneconomical (e.g., mass product recalls). Alternatively, the focus is on measuring the likelihood of the properties being violated.

Compositional automata-based model construction allows building complex models by specifying the behaviour of system components, and then computing automatically the behaviour of the system resulting from having components execute concurrently and synchronising over their public interfaces. Compositionality allows structuring the specification similarly to how complex systems are built and greatly simplifies the specification effort

In theory, compositional construction also reduces the validation effort as the validity of the composite specification should be dependent on the validity of the components, which are smaller and thus easier to validate. However, this requires the models to fulfil two fundamental conditions. First, it must be possible to isolate the behaviour of a component in such a way that its behaviour can be described independently of the environment it may execute in. Second, a composition operation is required that can guarantee to preserve in the composite system the independent behaviour of each component.

In constructing probabilistic automata compositionally, measuring the likelihood of a component making a choice independently of the behaviour of its environment can be

- *E. Pavese is with the Departamento de Computación, Universidad de Buenos Aires. E-mail: epavese@dc.uba.ar.*
- *V. Braberman is with the Departamento de Computación, Universidad de Buenos Aires and CONICET. E-mail: vbraber@dc.uba.ar.*
- *S. Uchitel is with the Departamento de Computación, Universidad de Buenos Aires; Imperial College London and CONICET. E-mail: suchitel@dc.uba.ar.*

Manuscript received 11 Mar. 2015; revised 17 Sept. 2015; accepted 2 Jan. 2016. Date of publication 7 Feb. 2016; date of current version 23 Sept. 2016. Recommended for acceptance by C. Pasareanu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2527000

0098-5589 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

notoriously difficult [2]. Doing so may require careful decomposition of probabilities that were estimated or measured from an existing composite system's execution log. These compound probabilities need to be decomposed into conditional ones, that can be incorporated into a component description that will form part of a new composite system that is expected to replace the system used for measurement. For instance, the likelihood of a user selecting coffee in a vending machine must be extracted conditionally from the different available choices the user may have had at the moment of selecting the beverage.

Having specified and validated probabilistic behaviour component-wise, the expectation is that the probabilistic properties that are guaranteed by the numerical annotations of probabilistically independent events in the component descriptions continue to be preserved over the composite system. For example, a vending machine user's overall likelihood of selecting coffee, independently of the system it is interacting with, is guaranteed to be between 0 and $\max$; where 0 represents the worst case in which the vending machine never offers coffee, while $\max$ is the maximum likelihood of choosing coffee for some combination of offered drinks. The expectation is that, no matter what vending machine is used, the system level probability of coffee being selected will continue to be between 0 and $\max$, as this range of probabilities is dependent on the user behaviour alone.

Although many variations of probabilistic automata (e.g., [3], [4], [5], [6]) and composition operators (see [7] for a survey) have been proposed, none of these support specification of non-deterministic and probabilistic component behaviour which, when observed through appropriate logics, is preserved in the composite system. The problems they exhibit can be characterised as a lack of an appropriate treatment of the notion of action controllability in combination with probabilistic descriptions. This leads to a number of problems including *i)* unclear semantics of the probabilities of the environment model, *ii)* unintuitive probability distributions in the composite model, and as a consequence *iii)* a lack of preservation of components' probabilistic properties over the composition [6], [8]. These shortcomings work against the goal of being able to reason about separate components, and have an assurance that the individually validated behaviours still hold once the composite system is built.

In this paper we propose a novel formalism for reasoning quantitatively in such a way that individual component behaviour is guaranteed to be preserved over a composition. This approach achieves the goal by combining, and adding to, notions taken from Input-Output Probabilistic Automata [3] and Interface Automata [9]. Thus, the main contribution of the paper is a formalism that supports compositional construction and validation of probabilistic models. We establish the correctness of our approach by showing that parallel composition of our models is such that it preserves a weak probabilistic simulation [10], thus preserving the desired behaviour over the composite system.

A second contribution of this paper is that we weaken the constraints imposed by Interface Automata to specify when two automata may be composed. Interface Automata requires a component to be able to accept environmental inputs immediately, disallowing internal computation even if the component is guaranteed to always allow the input at the end of such internal computation. In this paper, we propose a delayed synchronisation mechanism that relaxes the Interface Automata composition precondition and allows for richer modelling of internal computation.

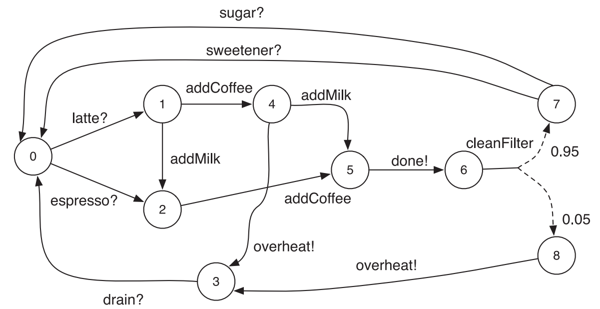


Fig. 1. A simple coffee machine.

The remainder of the paper is organised as follows: in Section 2 we use an example to motivate our approach, comparing to existing ones. In Section 3 we present some building blocks for our work, while in Section 4 we present our novel approach to the problem of probabilistic component-based verification. In Section 4.2 we show the main results of the paper; that is, composition over this new formalism is correct in terms of the semantics intended and property preservation. Along the paper, we illustrate our ideas with a motivating example and expand the ideas on a case study in Section 5. Finally, we discuss the relation of our work with previous efforts in Section 6 and offer our conclusions and prospects of future work on the subject.

2 Motivation

In this section we present a simple example to motivate the problem of compositional construction and analysis of probabilistic models. We also highlight the main issues related to the modelling of non-determinism and probabilities that threaten the compositional construction approach.

We first discuss the model for the machine presented in Fig. 1. This coffee machine has a digital tactile screen with which it interacts with the user, showing the user various options at different times during operation. First, the coffee machine offers the user, through the screen, a beverage choice between either an espresso or a latte. Once the user chooses her selection, the machine clears its screen and possibly shows a message telling the user to wait for beverage preparation. At this point, in a way unknown to the user, the machine prepares the beverage. Then, the machine informs the user it has finished the preparation. Now, the screen prompts for the addition of sugar or sweetener, and finally delivers the prepared drink when the choice is made. However, this coffee machine is known to sometimes overheat, requiring manual drainage. We have some information about the conditions under which the machine overheats, so we add this information to the model.

Without the need of having a model of the user, we can already validate some behaviour on this coffee machine model. For example, we may be interested in knowing whether the machine can overheat *after* it has added coffee to the cup, as at this point the coffee may boil and spill

violently towards the user, posing a safety hazard. By observing the trace that traverses states $0, 2, 5, 6, 8, 3$, we see that such an error is clearly possible. Moreover, note that there is always *at least* a 0.05 chance that this behaviour will manifest independently of user choices. It could be even worse if the machine always overheats at state 4, but we do not have the probabilistic information to quantify this claim. All we know is that the likelihood of the unsafe behaviour lies between 0.05 and 1.

For the sake of argument, assume for a moment that it is uneconomical to fix this behaviour unless the likelihood of it surpasses some probability threshold $p_{overheat} > 0.05$. Once we have the user model and compose it with our coffee machine, we could answer whether this threshold is met or not. For example, if the user were such that she never orders a latte, then the probability of overheating is exactly 0.05 and therefore there is no need for a fix. However, if she does order lattes, then it could overheat every time this happens.

In other words, we are interested in quantifying the occurrence of this error based on the expected behaviour of the environment interacting with the coffee machine.

In order to achieve this objective, we set out to produce a probabilistic model of the user's behaviour. However, not every modelling formalism will suit our compositional construction approach. Some choices may lead to problems which may not be immediately obvious, and these may arise from both the probabilistic aspect of the modelling and the non-deterministic as well.

2.1 Issues Arising from Probabilistic Modelling

There exist two main approaches for modelling probabilities over transitions of a behaviour model; namely modelling them via a *generative* [4] approach or a *reactive* [5] one.

Generative models are characterised by having a transition relation that defines, for each source state, a distribution on the cartesian product of the set of states and actions. That is, for each transition, both an action and a destination state are probabilistically selected. This choice of distributions leads to some well-known problems when trying to compose a generative model with another [6].

First, the generative paradigm forces all transitions to be probabilistically annotated. This is true even in the case of states that may transition because of both input and output actions. Probabilistically quantifying such choices would encode the probabilities of the resolution of this race between actions, an aspect that is usually outside the control of either component. A second problem arises if a component specifies a certain probability for an output action that is not accepted, or an input action that may never be received. In such a case, the probability of that action being triggered is obviously zero in the composition, yet the component specified a non-zero probability. This contradiction needs to be resolved at composition time. Although some solutions have been proposed to redistribute this missing probability [6], they are all arbitrary in that they need to *guess* what the component would have done if the action were not present.

These problems can be explained technically in terms of a lack that generative models have in modelling non-determinism, and a lack of clear semantics for the concurrent composition in such cases. Not allowing non-determinism means that these models are at a loss when it comes to modelling external actions the environment must act in response to.

Alternatively, the environment can be modelled under the *reactive* paradigm [5], under which each action on each state has a probabilistic distribution that defines the next state. Under the reactive paradigm, the action at each state is chosen in a non-probabilistic fashion (even allowing for non-determinism between different distributions for a same action), and only then the destination state is determined probabilistically. Reactive models, contrary to their generative counterpart, do allow for non-determinism, but do not allow probabilistic choice between different actions. There is a workaround for this, however, using hidden internal actions. State 6 in the coffee machine model of Fig. 1 shows an example of this workaround.

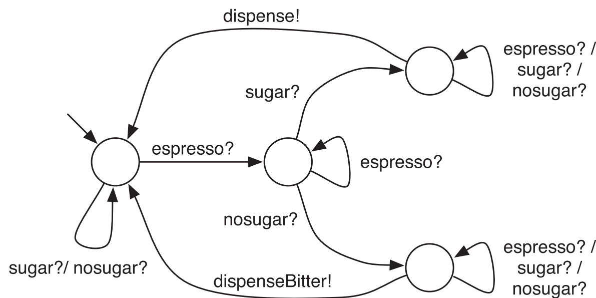
The use of a reactive probabilistic model solves many of the issues of the generative paradigm. However, in general, reactive probabilistic models allow for behaviour that does not necessarily consider input/output restrictions between components. Recall the property that the machine may overheat after dispensing coffee. We have already seen that this property holds with probability at least 0.05 for our modelled system. Yet, we can model a user environment that chooses to *never* synchronise on the `overheat` action, effectively blocking it. Oddly, the result obtained using standard composition [5] and analysis [11] is that the probability of the composite system overheating in an unsafe way is now zero, which means the error has probability 0 which is below the lower bound to error (0.05) that we had established when validating the machine model in isolation. The reason for such an unintuitive result is that the environment constrains the occurrence of a transition that should be controlled by the coffee machine.

The result in the previous analysis is quite unintuitive. There is a property that holds for the machine, and that does not depend on the environment to hold; but when composed with a certain environment it does not hold any more. Such a contradiction indicates that something is wrong with the way we have modelled either the system or the environment; in the way we composed them together, or in the probability computation. Again, this lack of behaviour preservation makes our goal of performing early probabilistic validation impossible.

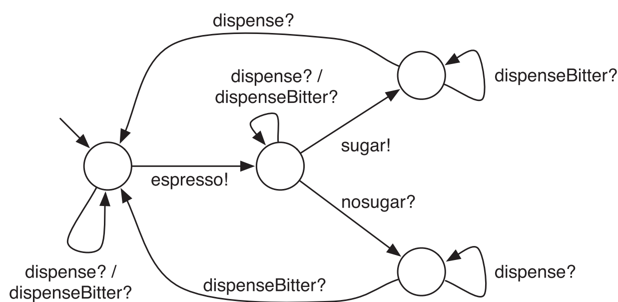
It is important to note that, contrary to the case of generative modelling, these problems do not relate strictly to the probabilistic annotations. Rather, they arise as a consequence of the inappropriate treatment of the notion of controllability. However, they do have an impact in terms of preservation of component properties. As such, we will make use of reactive modelling for the introduction of probabilities into the environment, but will need to resolve the synchronisation issues to ensure that components cannot restrict what other components are intended to control.

2.2 Issues Arising from Action Controllability

Most of the aforementioned synchronisation semantics problems have been tackled by introducing a semantic distinction between *input*, *output* and *internal* (also called *hidden*) actions. These sets of actions represent those that the component can listen to (in the case of input actions) and emit (in the case of output actions). The set of internal actions represents those

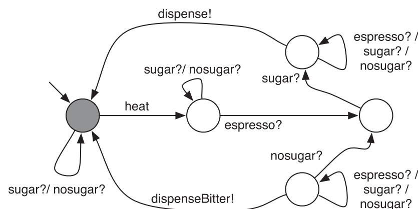


(a) Coffee machine model

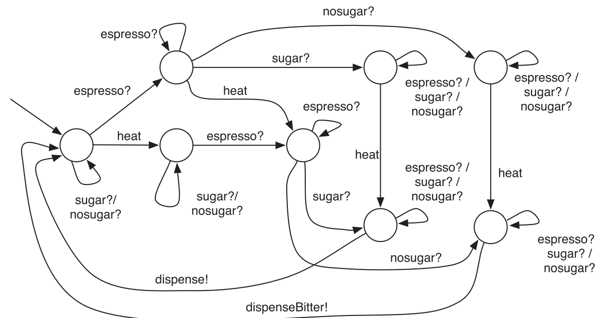


(b) Environment model

Fig. 2. Input/Output models for the simple coffee machine.



(a) A refined model that violates Input/Output automata rules



(b) A complex refinement satisfying Input/Output requirements

Fig. 3. Approaches to refinement of the coffee machine model.

that cannot be observed from outside the component, and do not take part of the *interface* of the component. The most well-know approaches to modelling that take this action segregation idea are those of Input/Output Automata [12] and Interface Automata [9]. Input/Output automata require that each component is input-enabled, that is, that they accept every possible input at every state. Interface Automata relaxes this condition a little by only enforcing that input synchronisations are always possible, but do not force an input to be enabled at a given state if it is known that it will not be triggered at that state.

Input enabledness introduces two modelling problems. First, it clutters models with unnecessary transitions. For example, we can look at the models in Figs. 2a and 2b. In this figure, the Input/Output automaton 2a models a coffee machine somewhat simpler to the one discussed above, while Input/Output automaton 2b models a potential environment that will interact with the coffee machine. It is noteworthy that the requirement for input enabledness does make the modelling more cumbersome.

The second problem is that input-enabledness restrictions are unrealistic for modelling some systems. It is usually the case that a component will accept some inputs in one state, while it will accept a different set of inputs in another. In fact, it may not accept any inputs at all until it finishes some internal computation, at which point it will accept new inputs. The need for immediate synchronisation with intended output actions hampers an iterative refinement approach where this internal behaviour is gradually modelled. As an example of how this problem arises, refer back to Figs. 2a and 2b. An engineer may now decide that the level of abstraction used to depict the behaviour of the coffee machine is too high, and she may decide to model some of the internal behaviour of the component. In particular, the engineer decides it would be interesting to note that the machine needs to heat the water for the beverages prior to preparing them. The result of this decision is a new model depicted in Fig. 3a. However, this new model is now not an Input/Output automaton respective to the environment model, as the grey state is blocking inputs from the environment that, at this point, may choose the beverage, and later choose whether to add sugar or not.

In order to turn this model into a valid Input/Output automaton it becomes necessary to take into account that the environment model expects a single push of the `espresso` button to prepare the drink, and a second one for the sweetener choice. Simply adding loops and ignoring the environment `espresso`, `sugar` and `nosugar` actions is insufficient, as the environment would now be expecting the beverage to be dispensed, and such an action would never happen. The model depicted by the automaton shown in Fig. 3b fulfils both this requirement and Input/Output synchronisation. It is easy to see that it is overly complex because of this need to remember user choices that may have happened during the internal actions of the machine. This complexity arises even for the very simple behaviour exhibited for this machine. Of course, an alternative modelling could consider signalling the environment that although the input actions are enabled, they are being ignored. However, such a decision involves a rework on the environment itself. Worse, such changes are a result of trying to fit a methodology rather than an attempt at modelling the actual interaction.

2.2.1 Interface Automata

Interface Automata [9] have been proposed as an alternative formalism, but one that still retains the notion of segregating interfacing actions. The Interface Automata formalism stipulates that the composition of a pair of components will be legal only if components do not block each other, that is, if every time that one component intends to exercise one of its output

actions, the other component enables such action (as part of its own input actions). In this case, it is not necessary to spuriously enable input actions, as only those that are actually needed are mandatory to be enabled. In this sense, Interface Automata allow for succinct modelling of interfacing protocols than their Input/Output counterpart, which assumes input-enabledness. However, similarly to Input/Output automata, they do require that the non-blocking behaviour be immediate, that is, whenever a component wants to emit one of its output actions, the corresponding input action must be immediately enabled at its counterpart component.

Except for the immediacy restrictions depicted above, Interface Automata seem to be a natural choice for modelling synchronisation and controllability. From an engineering point of view, it is natural to model the restriction of certain actions at selected states as long as these restrictions are compatible with the behaviour of the component that controls them.

In this way, assumptions about the behaviour of cooperating models can be encoded directly, easing the task of modelling interactions such as protocols enforcing ordered method calls, internal uninterruptible behaviour or system exceptions, among other useful system properties. This results in more concise models, as the engineer is released from the obligation of having to explicitly model responses for interactions that are known to not occur in the reality being modelled.

It is important to note, however, that specifying a similar formalism to the one we will present, but using Input/Output automata-like modelling is feasible. The choice of Input/Output Automata over Interface Automata is of no consequence regarding the solutions to the problems described in the previous sections, and the way to resolve them would be similar in both cases.

Regarding the immediate enabledness requirements discussed above, a formalism that allows for modelling such delayed synchronisation is thus desirable. Of course, an important requirement for such a model is that it can be guaranteed that for every possible future behaviour, the synchronisation point will always be available. Such guarantees will require some restrictions on unfair behaviour of the system under analysis that may hamper such guarantees. We will study these guarantees when we present our modelling formalism in Section 4.

2.3 Combining Probabilities Modelling and Synchronisation Semantics

Summarising the previous paragraphs, in order to model the probabilistic behaviour of the environment and compose it with a non-probabilistic behaviour model of the system to obtain meaningful quantitative results, a formalism is needed that can *i)* allow for modelling of both non-deterministic behaviour and probabilistic behaviour, *ii)* address notions of controllability and monitorability of actions by the environment and system (including synchronisation notions and delayed behaviour), and *iii)* preserve probabilistic properties of the environment after composition.

In the following sections we propose a formalism which distinguishes output/controlled and input/monitored actions, and also supports probabilistic and non-deterministic behaviour. Our formalism is inspired on *probabilistic reactive* models for introducing probabilities, as we discussed above. Synchronisation will be modelled inspired on Interface Automata. This combination allows for satisfying objective *i)* in the above paragraph, as well as *ii)*.

However, challenges arise from the combination of these two formalisms. The previous discussion hints at some of these challenges, and we elaborate on our solution on the next sections. We focus especially on the mechanisms that allow us to ensure that *iii)* is satisfied.

We will also tackle the problem of the need for immediate synchronisation. To this end, we will introduce a notion of fairness for executions of these automata that allows us to distinguish those cases where future synchronisation of delayed actions is guaranteed from those where it is not. Further, we will also present a suitable composition operator for these automata and in Theorem 4.1 we demonstrate the required results of property preservation.

3 Background

In this section we present some building blocks for our work. We will recall Interface Automata and related notions, as well as the base probabilistic model in which our new formalism is based.

We will also make use of various concepts related to measure and probability theory when referring to probabilistic models and its characteristics. Although various concepts will be summarily introduced, the interested reader is referred to [13] for in-depth discussion. We will also introduce some other concepts relating to probabilistic automata theory as the need arises.

3.1 Interface Automata Notions

3.1.1 Model and Executions

Definition 3.1 (Interface Automata [9]). *An* Interface Automaton *is a tuple* $P = < S_P, s_P^0, A_P^I, A_P^O, A_P^H, R_P >$ *where:*

- S_P *is a finite set of states.*
- $s_P^0 \in S_P$ *is a distinct* initial *state.*
- A_P^I, A_P^O, A_P^H *are finite and mutually disjoint sets of* input, output *and* hidden *actions respectively. We denote the set of all actions* $A_P = A_P^I \cup A_P^O \cup A_P^H$.
- $R_P \subseteq S_P \times A_P \times S_P$ *is the transition relation.*

We will write $A_P^I(s)$, $A_P^O(s)$ and $A_P^H(s)$ for a state $s \in S_P$ to denote the subset of actions in A_P^I, A_P^O and A_P^H, respectively, that are *enabled* at s. An action $a \in A_P$ is said to be enabled at state $s \in S_P$ if there exists $t \in S_P$ such that $(s, a, t) \in R_P$. Alternatively, we may say that the transition (s, a, t) itself is enabled if the previous condition holds. Analogously, we denote $A_P(s)$ the subset of actions enabled at state s, regardless of them being input, output or hidden actions. Without loss of generality, we require that for each state $s \in S_P$, there exists $s' \in S_P, a \in A_P$ such that $(s, a, s') \in R_P$.

In essence, an Interface Automaton is a labelled transition system (LTS) [14] where its action set has been further subdivided to distinguish the input, output and hidden actions. As we will see, this does not make a syntactic difference, but it does semantically. Also, note that we have reduced the original set of initial states to a single one without loss of generality.

Definition 3.2 (Execution fragment and executions). *An* execution fragment *of an Interface Automaton P is a (possibly infinite) sequence $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ of alternating states and action labels. Execution fragments always start with a state and, if finite, also end with a state. Each subsequence $s_i a_{i+1} s_{i+1}$ within an execution fragment of P is such that $(s_i, a_{i+1}, s_{i+1}) \in R_P$.*

Given an execution fragment α, $first(\alpha)$ denotes the first state of the fragment, while $tail(\alpha)$ denote the execution fragment from its second state. $tail(\alpha)$ might be empty if α is finite and consists of only one state. If α is finite, $last(\alpha)$ denotes its final state.

An execution *of an Interface Automaton P is an execution fragment α of P such that $first(\alpha) = s^0_P$, the initial state of P. As executions are execution fragments themselves, they can also be finite or infinite.*

We will also note $fragments(P)$ and $fragments{*}(P)$ to denote the set of execution fragments of P and the set of *finite* execution fragments of P, respectively. Accordingly, we will note $execs(P)$ and $execs{*}(P)$ for the set of executions and finite executions of P. For convenience, we also define $length : fragments(P) \to \mathbb{N} \cup \infty$ to be the number of states traversed by the execution fragment. Finally, we define projectors α^s_i and α^a_i that return the ith state and ith transition label respectively. Note that α^s_i is defined from 0 through $length(\alpha) - 1$, while α^a_i is defined from 1 through $length(\alpha) - 1$. Finally, we will note $\alpha \le \alpha'$ to indicate that the execution fragment α is a *prefix* of execution fragment α'; that is, for each $0 \le i \le length(\alpha) - 1$, $\alpha^s_i = {\alpha'}^s_i$ and for each $1 \le j \le length(\alpha) - 1$, $\alpha^a_j = {\alpha'}^a_j$. Accordingly, $suffix(\alpha, i)$ is defined for every $i < length(\alpha)$ and obtains the execution fragment that results of dropping the first i states and actions from an execution fragment. Therefore, for an execution fragment $\alpha = s_0 a_1 s_1 a_2 s_2 a_3 s_3 \ldots$, $suffix(\alpha, 0) = \alpha$, $suffix(\alpha, 1) = s_1 a_2 s_2 a_3 s_3 \ldots$ and so on.

3.1.2 Parallel Composition

The notion of action segregation in Interface Automata allows for the notion of *composability* of Interface Automata:

Definition 3.3 (Composability [9]). *Let P and Q be two Interface Automata. We say P and Q are* composable *if it holds simultaneously that $A^H_P \cap A_Q = \emptyset$, $A_P \cap A^H_Q = \emptyset$, $A^I_P \cap A^I_Q = \emptyset$, and $A^O_P \cap A^O_Q = \emptyset$.*

Furthermore, when referring to the interaction of two Interface Automata P and Q, it is usual to allude to its shared set of actions, $Shared(P,Q) = A_P \cap A_Q$. Note that if P and Q are composable, then $Shared(P,Q) = (A^I_P \cap A^O_Q) \cup (A^O_P \cap A^I_Q)$. We recall the definition of Interface Automata product and illegal states.

Definition 3.4 (Product [9]). *Let P and Q be two composable Interface Automata. Their product $P \otimes Q$ is another Interface Automaton defined by states $S_{P\otimes Q} = S_P \times S_Q$; initial state $s^0_{P\otimes Q} = (s^0_P, s^0_Q)$; action sets $A^I_{P\otimes Q} = (A^I_P \cup A^I_Q) \setminus Shared(P,Q)$; $A^O_{P\otimes Q} = (A^O_P \cup A^O_Q) \setminus Shared(P,Q)$ and $A^H_{P\otimes Q} = A^H_P \cup A^H_Q \cup Shared(P,Q)$. Its transition relation $R_{P\otimes Q}$ is defined by the set*

$$\left\{ \begin{array}{c} \{((s,t),a,(s',t)) \text{ such that } (s,a,s') \in R_P \wedge \\ t \in S_Q \wedge a \notin Shared(P,Q)\} \cup \\ \{((s,t),a,(s,t')) \text{ such that } (t,a,t') \in R_Q \wedge \\ s \in S_P \wedge a \notin Shared(P,Q)\} \cup \\ \{((s,t),a,(s',t')) \text{ such that } a \in Shared(P,Q) \wedge \\ (s,a,s') \in R_P \wedge (t,a,t') \in R_Q\} \end{array} \right\}.$$

Since the behaviour of a composite Interface Automaton is directly related to the behaviour of each of its components, there is a close relationship between the executions (and executions fragments) of a composite system, and those of its components. However, this depends on the semantics of the interface. The action segregation introduced in the definition of Interface Automata is essentially a description language tool. Although it has no bearing in the previous formal definitions, it introduces the notion of *illegal composition states*. Intuitively, a composition state will be regarded as illegal if, somehow, it violates the enabledness of the intended actions of each component.

Definition 3.5 (Illegal states [9]). *Given two composable Interface Automata P and Q, their product's illegal states are defined by the set $Illegal(P,Q) \subseteq S_P \times S_Q$. For any $s \in S_P$, $q \in S_Q$, $(s,q) \in Illegal(P,Q)$ if $\exists a \in Shared(P,Q)$ such that $a \in A^O_P(s) \wedge a \notin A^I_Q(q)$, or conversely $\exists a \in Shared(P,Q)$ such that $a \notin A^I_P(s) \wedge a \in A^O_Q(q)$.*

Informally, the idea behind illegal states is that, for a composition to be legal, component systems should not be able to block each other's enabled *output* actions. We will abuse notation and say that the product $P \otimes Q$ of two Interface Automata P and Q is *legal* if the product has no *reachable* illegal states.

The notions of composability and illegal states make it possible to define what a *valid environment* for a given Interface Automaton is.

Definition 3.6 (Valid environment [9]). *Given a non-empty Interface Automaton P (that is, P has at least one state), another Interface Automaton Q is a* valid environment *for P if all the following hold: i) P and Q are composable; ii) $A^I_Q = A^O_P$; iii) no state in $Illegal(P,Q)$ is reachable in $P \otimes Q$; and iv) $P \otimes Q$ is non-empty as well.*

3.1.3 Non-Determinism and Schedulers

Finally, it is important to note that the distinct execution fragments generated by an Interface Automata depend on how the choice between different transitions is resolved. That is, whenever two or more actions can be chosen in a state, the choice of which action to take is left unspecified, and can only be resolved by an external agent. In order to distinguish this choice from the probabilistic choices that will appear later in the paper, we will refer to these choices as *non-deterministic* choices. Note that this is slightly different from a common meaning of *non-determinism* which is limited to the choice between different transitions with the same label. In this paper we refer to non-deterministic choices to those that are *not probabilistic* in nature.

In order to characterise this external agent, and thus the different non-deterministic choices and the execution

fragments that they induce, we will introduce the notion of a *scheduler*.

Definition 3.7 (Scheduler). *A* scheduler σ *for an Interface Automaton* $P = < S_P, s_P^0, A_P, R_P >$ *(also called an* adversary*) is a total function* $\sigma : execs*(P) \rightarrow R_P$*, such that* $\sigma(\alpha)$ *is a transition starting from* $last(\alpha)$*; and whenever* $\sigma(\alpha) = (last(\alpha), a, s)$ *it must be that* $(last(\alpha), a, s) \in R_P$*. The notation* $Sched(P)$ *refers to the set of all possible schedulers for the automaton* P*; while* $\sigma(\alpha)_a$ *and* $\sigma(\alpha)_s$ *refer to the scheduled action and destination state given an execution* α*, respectively.*

The idea behind schedulers is that they drive the execution of the automaton by resolving all possible non-determinism. As such, they restrict the set of possible execution fragments. Extending this notion, a set of schedulers defines a set of possible executions and execution fragments.

Definition 3.8 (Scheduler-generated executions). *Given an Interface Automaton* P*, a scheduler* σ_P *and an execution* $\alpha \in execs(P)$*, we say that* σ_P generates α *over* P *if and only if for each* $0 \leq i < length(\alpha)$ *it holds that* $\sigma_P(\alpha_0^s \alpha_1^a \ldots \alpha_i^s) = (\alpha_i^s, \alpha_{i+1}^a, \alpha_{i+1}^s)$*.*

Note that once a scheduler σ is set for an Interface Automaton P, this scheduler eliminates all possible branching. That is, it generates a single infinite execution fragment, along with its infinite set of finite prefixes.

Some schedulers will not be very useful to our approach, as they may model invalid behaviours. In particular, we are interested in schedulers that are *fair* in their choices of non-determinism resolution, as they have desirable properties which will be discussed later. The following definitions deal with our requirements for fairness, which have been adapted from [15], [16], [17].

Definition 3.9 (Fair executions). *Let* α *be an infinite execution over an Interface Automaton* P*. For each* $s \in S_P$*, let* $Traversals(\alpha, s) = \{i \in \mathbb{N}_0 \cdot \alpha_i^s = s\}$*, that is* $Traversals(\alpha, s)$ *denotes the indexes in* α *where state* s *is traversed. Similarly, define* $Traversals(\alpha, (s, a, s'))$ *to be the indexes in* α *where the transition* (s, a, s') *is taken.*

We say that α *is a* fair *execution if for each* $s \in S_P$ *such that* $Traversals(\alpha, s)$ *is an* infinite *set it holds that whenever* (s, a, s') *is an enabled transition from* s *(that is,* $(s, a, s') \in R_P$*), then the set* $Traversals(\alpha, (s, a, s'))$ *is also infinite.*

Informally, an execution is *fair* if, every time that it passes through a state t infinitely often, then it also progresses over each of its enabled transitions infinitely often. In other words, whenever a transition is enabled and the execution has the opportunity to take it, a fair execution cannot indefinitely avoid taking it. We will extend this notion of fairness to schedulers.

Definition 3.10 (Strictly fair schedulers [1]). *A scheduler* σ *is* strictly fair *(also called* strong fair*) if the infinite execution it generates is itself* fair*.*

The reasons behind the choice of words on defining schedulers as *strictly* fair in Definition 3.10 will be made more clear once we examine schedulers for probabilistic models.

3.1.4 Logics for Property Description

Several temporal logics have been put forth for reasoning about the protocols described by automata-like formalisms. As we will see later when we discuss property preservation, we need to preserve the branching structure of components within the composition. We will therefore express these behaviour properties with the logic CTL (*Computational Tree Logic*) [18], or some variants of it. ACTL [19] (not to be confused with the universal fragment of CTL) in particular is a temporal logic equivalent to CTL. The main difference is that, while CTL focuses its predicates on *states*, ACTL does so on the set of *actions*. ACTL will become useful to us, as it allows us to express directly the restrictions that pertain to the availability of actions for synchronisation, which will allow us to expand the notion of composability in Section 4.

Definition 3.11 (ACTL Syntax [19]). *The set of ACTL formulae is defined as the smallest set of* state formulae *such that*

- $True$ *is a state formula;*
- *if* ϕ_1 *and* ϕ_2 *are state formulae, then* $\neg\phi_1$ *and* $\phi_1 \wedge \phi_2$ *are also state formulae;*
- *if* ψ *is a* path formula*, then* $\neg\psi$ *is also a path formula;*
- *if* ψ *is a path formula then* $\exists\psi$ *is a state formula;*
- *if* ϕ_1 *and* ϕ_2 *are state formulae and* a *is an action label, then* $X_a\phi_1$*,* $\phi_1 \mathcal{U} \phi_2$ *and* $\phi_1 \mathcal{U}_w \phi_2$ *are path formulae.*

Definition 3.12 (ACTL Semantics [19]). *Let* $M = < S_M, s_M^0, A_M^I, A_M^O, A_P^H, R_M >$ *be an Interface Automaton. The semantics of an ACTL formula are given by a* satisfaction relation*, which is defined for* M *over execution fragments* $\alpha \in fragments(M)$ *for path formulae* ψ *(noted* $M, \alpha \models \psi$*), and over states* $s \in S_M$ *for state formulae* ϕ *(noted* $M, s \models \phi$*). The satisfaction relation is defined inductively as follows, where* ϕ_1, ϕ_2 *denote state formulae and* ψ *denotes a path formula, and* $a \in A_M$*:*

$$
\begin{array}{lcl}
M, s \models True & & always\ holds \\
M, s \models \neg\phi & \Leftrightarrow & \neg(M, s \models \phi) \\
M, s \models \phi_1 \wedge \phi_2 & \Leftrightarrow & M, s \models \phi_1 \wedge M, s \models \phi_2 \\
M, s \models \exists\psi & \Leftrightarrow & \exists\alpha \in fragments(M)\ such\ that \\
 & & \alpha_0^s = s \wedge \alpha \models \psi \\
M, \alpha \models \neg\psi & \Leftrightarrow & \neg(M, \alpha \models \psi) \\
M, \alpha \models X_a\phi & \Leftrightarrow & length(\alpha) > 1 \wedge \alpha_0^a = a \wedge \\
 & & M, \alpha_1^s \models \phi \\
M, \alpha \models \phi_1 \mathcal{U} \phi_2 & \Leftrightarrow & (\exists 0 \leq j < length(\alpha))(\forall 0 \leq i < j) \\
 & & M, \alpha_i^s \models \phi_1 \wedge M, \alpha_j^s \models \phi_2 \\
M, \alpha \models \phi_1 \mathcal{U}_w \phi_2 & \Leftrightarrow & (M, \alpha \models \phi_1 \mathcal{U} \phi_2) \vee \\
 & & (\forall 0 \leq i < length(\alpha)) M, \alpha_i^s \models \phi_1
\end{array}.
$$

We will abuse notation and, given a finite set of actions A, note $X_A\phi$ as an equivalent to $\bigvee_{a \in A} X_a\phi$. Also, we can further refine the satisfaction relation to ask whether a formula ϕ is satisfied by an Interface Automaton M when under a given scheduler σ. The satisfaction semantics are kept almost the same, except that whenever we need to check for fragments in $fragments(M)$, we must restrict them to those *generated* by σ.

3.2 Probabilistic Automata

The previously presented definitions push us halfway towards our goal of providing a suitable language for the specification of *probabilistic* user environments. The

probabilistic semantics are introduced via a well-known reactive probabilistic formalism, that of Segala's Simple Probabilistic Automata [10], [20]. As we will see, this model extends classic LTSs by modifying the transitions so that they no longer reach a single state, but a probabilistic distribution over a set of destination states instead.

Definition 3.13 (Segala's Simple Probabilistic Automaton (SPA)). *A* Simple Probabilistic Automaton *is defined by a tuple* $M =< S_M, s^0_M, A_M, R_M >$ *where*

- S_M *is a finite set of states.*
- $s^0_M \in S_M$ *is a distinct initial state.*
- A_M *is a finite set of actions.*
- $R_M \subseteq S_M \times A_M \times D(S_M)$ *is a transition relation, where* $D(S_M)$ *denotes the set of* probabilistic distributions *over the set of states* S_M*. Further,* R_M *is required to be finite.*

We will note $R_M(s)$ to denote the set of all transitions that originate on state s, that is, those tuples in R_M where the first component is s. Similarly, we will note $R_M(s, a)$ to note the set of transitions originating in s through action a. For convenience and without loss of generality, we will assume that for all states $s \in S_M$, the transition relation is such that $R_M(s) \neq \emptyset$ [21].

In a manner similar to other automata-based behaviour description formalisms, Simple Probabilistic Automata can be constructed compositionally as the product of other, smaller Simple Probabilistic Automata.

Definition 3.14 (Simple Probabilistic Automata product [10]). *Let* $M_1 =< S_1, s^0_1, A_1, R_1 >$ *and* $M_2 =< S_2, s^0_2, A_2, R_2 >$ *be two Simple Probabilistic Automata. Their product* $M_1 \otimes M_2$ *is another Simple Probabilistic Automaton* $M =< S_{M_1\otimes M_2}, s^0_{M_1\otimes M_2}, A_{M_1\otimes M_2}, R_{M_1\otimes M_2} >$*, such that*

- $S_{M_1\otimes M_2} = (S_1 \times S_2)$
- $s^0_{M_1\otimes M_2} = (s^0_1, s^0_2)$
- $A_{M_1\otimes M_2} = A_1 \cup A_2$
- *given* $(s,t) \in S_1 \otimes S_2$, $a \in A_1 \cup A_2$ *and* $\delta \in D(S_{M_1\otimes M_2})$, $R_{M_1\otimes M_2}$ *is such that* $((s,t), a, \delta) \in R_{M_1\otimes M_2}$ *if and only if any of the following is satisfied:*
  1) $a \in A_1 \wedge a \notin A_2 \wedge \forall s' \in S_1\ (\exists \delta_1 \in D(S_1)$ *such that* $(s, a, \delta_1) \in R_1 \wedge \forall s' \in S_1, \delta((s', t)) = \delta_1(s'))$
  2) $a \in A_2 \wedge a \notin A_1 \wedge \forall t' \in S_2\ (\exists \delta_2 \in D(S_2)$ *such that* $(t, a, \delta_2) \in R_2 \wedge \forall t' \in S_2, \delta((s, t')) = \delta_2(t'))$
  3) $a \in A_1 \cap A_2 \wedge \exists \delta_1 \in R_1(s, a) \wedge \exists \delta_2 \in R_2(t, a)$ *such that* $\forall s' \in S_1, t' \in S_2, \delta((s', t')) = \delta_1(s') \times \delta_2(t')$.

As was the case for Interface Automata earlier in the paper, SPAs are composed through an asynchronous product, but synchronising on shared actions. This distinction is made clear when defining the transition relation for the product SPA. Clauses 1 and 2 state that, whenever an action is not shared by both processes, the possible distributions governing transitions in the product are exactly those that come from each component process. Clause 3 describes the synchronising nature of the Simple Probabilistic Automata product. The distributions for transitions where the action label is shared are computed as the product of the distributions for each of the components. Note that, when composing states from different components, if at any of these states the shared action is not enabled (i.e., the state does not provide an outgoing transition through the shared action), then no distribution is present and the product cannot be computed. In that case, the product state does not have an outgoing transition on the shared action—it does not synchronise.

The definitions for execution fragments and complete executions still apply to Simple Probabilistic Automata, as we are still interested in the possible traces of the Simple Probabilistic Automaton.

Definition 3.15 (SPAs' execution fragments and executions). *An* execution fragment *of a Simple Probabilistic Automaton* M *is a (possibly infinite) sequence* $\alpha = s_0(a_1, p_1)s_1(a_2, p_2)s_2 \ldots$ *of alternating states and transitions, where these transitions are annotated by their governing action and associated probability. Execution fragments always start with a state and, if finite, also end with a state. Each sequence* $s_i(a_{i+1}, p_{i+1})s_{i+1}$ *within an execution fragment of* M *is such that there exists a probabilistic distribution* δ *such that* $(s_i, a_{i+1}, \delta) \in R_P$*, and* $\delta(s_{i+1}) = p_{i+1}$.

Given an execution fragment α*,* $first(\alpha)$ *denotes the first state of the fragment, while* $tail(\alpha)$ *denotes the execution fragment from its second state.* $tail(\alpha)$ *might be empty if* α *is finite and consists of only one state. If* α *is finite,* $last(\alpha)$ *denotes its final state.*

An execution *of a Simple Probabilistic Automaton* M *is an execution fragment* α *of* M *such that* $first(\alpha) = s^0_M$*, the initial state of the automaton. As executions are execution fragments themselves, they can also be finite or infinite.*

As was the case for Interface Automata, we will also note $fragments(M)$ and $fragments*(M)$ to denote the set of execution fragments of M and the set of *finite* execution fragments of M, respectively. Additionally, we will note $execs(M)$ and $execs*(M)$ for the set of executions and finite executions of M. We also define $length : fragments(M) \rightarrow \mathbb{N} \cup \infty$ to be the number of states traversed by the execution fragment. For additional convenience, we define projectors α^s_i, α^a_i and α^p_i that return the ith state, ith transition label and ith associated probability respectively. Note that α^s_i is defined from 0 through $length(\alpha) - 1$, while α^a_i and α^p_i are defined from 1 through $length(\alpha) - 1$. Finally, we will note $\alpha \leq \alpha'$ to indicate that the execution fragment α is a finite *prefix* of execution fragment α'. Again, $suffix(\alpha, i)$ is defined for every $i < length(\alpha)$ and obtains the execution fragment that results of dropping the first i states and probability-action pairs from an execution fragment. Therefore, for an execution fragment $\alpha = s_0(a_1, p_1)s_1(a_2, p_2)s_2(a_3, p_3)s_3 \ldots$, $suffix(\alpha, 0) = \alpha$, $suffix(\alpha, 1) = s_1(a_2, p_2)s_2(a_3, p_3)s_3 \ldots$ and so on.

The notion of schedulers for resolving non-determinism is also preserved, but note that instead of scheduling an action and a destination state, it schedules a distribution on destination states instead.

Definition 3.16 (Scheduler for Simple Probabilistic Automata). *A* scheduler σ *for a Simple Probabilistic Automaton* $M =< S_M, s^0_M, A_M, R_M >$ *(also called an* adversary*)*

is a total function $\sigma : execs*(M) \to A_M \times D(S_M)$*, such that if* $\sigma(\alpha) = (a, \delta)$ *it must be that* $(last(\alpha), a, \delta) \in R_M$.

For ease of reading, we will note $\sigma(\alpha)_a$ to refer to the scheduled action label, and $\sigma(\alpha)_\delta$ to refer to the chosen distribution. That is, if $\sigma(\alpha) = (a_0, \delta_0)$, then $\sigma(\alpha)_a = a_0$ and $\sigma(\alpha)_\delta = \delta_0$.

It is noteworthy, however, that resolving non-determinism via a scheduler for an SPA does not, as was the case for Interface Automata, produce a unique execution. Rather, resolving non-determinism induces a fully probabilistic process, specifically a Discrete Time Markov Chain (DTMC) which, in turn induces a set of execution fragments. For more insight on these probabilistic processes the reader may refer to [22], [23].

This combination of a scheduler σ and an SPA M defines a probability measure δ on the σ-algebra generated by the *cones* (also called *cylinder sets* in the literature) of execution fragments.

Definition 3.17 (Cones and probability measure [20]). *Given a finite execution fragment* α *of an SPA* M*, the cone of* α *is the set of execution fragments* $C_\alpha = \alpha' \in fragments(M) \cdot \alpha \leq \alpha'$*. The measure of a cone* C_α *under scheduler* σ *is defined as*

$$\delta(C_\alpha, M, \sigma) = \prod_{i=1}^{length(\alpha)} IsSched(\sigma, \alpha, i-1, \alpha_i^a) \times \delta_{Sched}(\sigma, \alpha, i-1)(\alpha_i^s),$$

where $\delta_{Sched} : Sched(M) \times fragments*(M) \times \mathbb{N} \to D(S_M)$, *and* $IsSched : Sched(M) \times fragments*(M) \times \mathbb{N} \times A_M \to \{0, 1\}$ *are such that* $\delta_{Sched}(\sigma, \alpha, n) = \sigma(\alpha_0 \dots \alpha_n)_\delta$ *and*

$$IsSched(\sigma, \alpha, n, a) = \begin{cases} 1 & \text{if } \sigma(\alpha_0 \dots \alpha_n)_a = a \\ 0 & \text{otherwise.} \end{cases}$$

In other words, δ_{Sched} *obtains the distribution corresponding to the next scheduled transition, while* $IsSched$ *checks whether in fact* a *is the next scheduled action.*

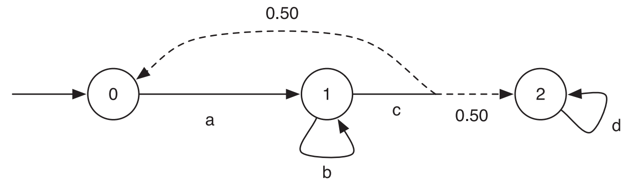
Cone measure as defined in Definition 3.17 can easily be extended for sets of non-overlapping cones. Given a SPA M, a scheduler σ, and a set Γ of finite execution fragments such that for every $\alpha_i, \alpha_j \in \Gamma$ neither is a prefix of the other, we can define the measure of the set Γ (noted $\delta(\Gamma, M, \sigma)$) as follows:

$$\delta(\Gamma, M, \sigma) = \sum_{\alpha \in \Gamma} \delta(C_\alpha, M, \sigma).$$

The notion of cones is essential for the definition of the σ-algebra underlying SPAs, since it gives us a way to *measure* sets of traces. As we will see later, this concept will have a strong relation with the logics we will employ to reason about SPA behaviour.

Leveraging on the previous definitions, we can characterise the set of execution fragments generated by a scheduler σ on an SPA M.

Definition 3.18 (Simple Probabilistic Automaton scheduled fragments). *Let* M *be a Simple Probabilistic Automaton, and* σ *a scheduler for* M*. The set of* scheduled execution fragments *of* M *through* σ *is the set of execution fragments* $fragments(M, \sigma) \subseteq fragments(M)$ *such that* $\alpha \in fragments(M, \sigma) \Leftrightarrow (\forall \alpha' \in fragments*(M) \cdot \alpha' \leq \alpha \Rightarrow \delta(C_{\alpha'}, M, \sigma) > 0)$.



$$\begin{cases} \sigma_1(0, \alpha) & = & a \\ \sigma_1(1, \alpha) & = & b \\ \sigma_1(2, \alpha) & = & d \end{cases}$$

$$\begin{cases} \sigma_2(0, \alpha) & = & a \\ \sigma_2(1, \alpha) & = & c \\ \sigma_2(2, \alpha) & = & d \end{cases}$$

Fig. 4. A simple probabilistic automaton and two unfair schedulers. σ_2 is *probabilistically fair.*

In other words, $fragments(M, \sigma)$ is the set of the execution fragments of SPA M that may be generated probabilistically given a scheduler. Each scheduler for an SPA generates a (possibly infinite) set of executions and execution fragments, instead of a single execution as was the case for automata that do not exhibit probabilities. Therefore, schedulers alone are not enough to exercise complete control over the executions of an SPA, as probabilities also have an influence on possible behaviour. In particular, this implies that the notion of scheduler fairness needs to be adjusted. Consider for example the case of the SPA depicted in Fig. 4, and two possible schedulers σ_1 and σ_2 that behave roughly as described beside the automaton. In both cases, a nonfair execution is possible – $0a1b1b1 \dots b1b1b1 \dots$ in the case of scheduler σ_1, and $0a1c0a1c0a1 \dots c0a1c0a1 \dots$ in the case of scheduler σ_2. Under the previous definition, neither of these schedulers are themselves fair. However, note that the probability of the nonfair executions under σ_2 is actually zero, while those under σ_1 have nonzero probability. This important distinction leads to the definition of *probabilistically fair* schedulers. Once again, this definition has been put forth previously in [15], [16], [17].

Definition 3.19 (Probabilistically fair schedulers). *A scheduler* σ *is* probabilistically fair *for an SPA* M *if it either is strictly fair, or else the measure of the subset of nonfair executions within its scheduled fragments set* $fragments(M, \sigma)$ *(that is, the measure of the cones that describe the set) is zero.*

In other words, a probabilistically fair scheduler generates fair execution fragments *almost surely*, while they *almost never* produce unfair execution fragments. For the remainder of this paper, when we refer to *fair* schedulers for SPAs, we will be implicitly referring to *probabilistically fair* ones, unless specifically noted.

3.2.1 Simulations for Probabilistic Automata

The notion of simulations [24] is useful to compare the behaviours of automata, and is a step forward to establishing equivalence between them. In the context of probabilistic

automata the concept of simulations has also been studied [7]. In this work we will leverage on the particular notion of probabilistic branching simulations [20]. We will employ these simulations to show that our approach to composability preserves behaviour, in the form of establishing these kind of simulations between different automata.

Before we can define probabilistic branching simulations properly, we need to understand the basic blocks with which they are built. Probabilistic branching simulations must show that the probabilistic information is simulated between different automata. The main mechanism through which this is achieved is by showing that a probability distribution on the simulated system can be embedded into a probability distribution over the system that simulates it.

Definition 3.20 (Distribution embedding [10]). *Let $\mathcal{R} \subseteq S \times T$ be a relation between two sets S and T; and let $\delta_S \in D(S)$ and $\delta_T \in D(T)$ be two distributions on each of those sets. We say δ_S and δ_T are in relation $\sqsubseteq_{\mathcal{R}}$, noted $\delta_S \sqsubseteq_{\mathcal{R}} \delta_T$ if there exists a weight function $w : S \times T \to [0, 1]$ such that*

1) *for each $s \in S$, $\sum_{t \in T} w(s, t) = \delta_S(s)$;*
2) *for each $t \in T$, $\sum_{s \in S} w(s, t) = \delta_T(t)$;*
3) *for each $(s, t) \in S \times T$, $w(s, t) > 0 \Rightarrow s\mathcal{R}t$.*

The notion of distribution embedding bears a close relationship to embedding a probabilistic transition of one system into a combination of several transitions on the other, and vice versa. The notion of combined steps captures this relationship.

Definition 3.21 (Combined step [10]). *Let M be an SPA and $s \in S_M$ an arbitrary state. Let $\delta_C \in D(A_M \times S_M)$. We say (s, δ_C) is a* combined step *of M if there exists a weight function $w : R_M(s) \to \mathbb{R}$ such that for each action a in A_M the following hold:*

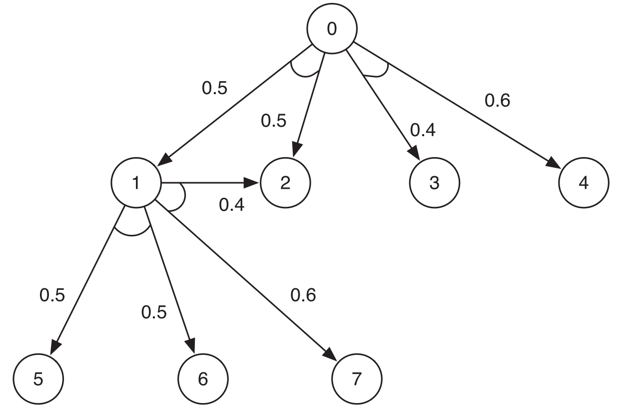
- *$\sum_{(t,a,\delta) \in R_M(s)} w((t, a, \delta)) = 1$; and*
- *for every $s' \in S_M$ it holds that $\delta_C(a, s') = \sum_{(t,a,\delta) \in R_M} w(t, a, \delta)\delta(s')$.*

In other words, a combined step of M at state s is a convex combination of the transitions allowed by M at state s. We will note $s \xrightarrow{a,p}_C s'$ every time that there exists a combined step $C = (s, \delta_C)$ such that $\delta_C(a, s') = p$.

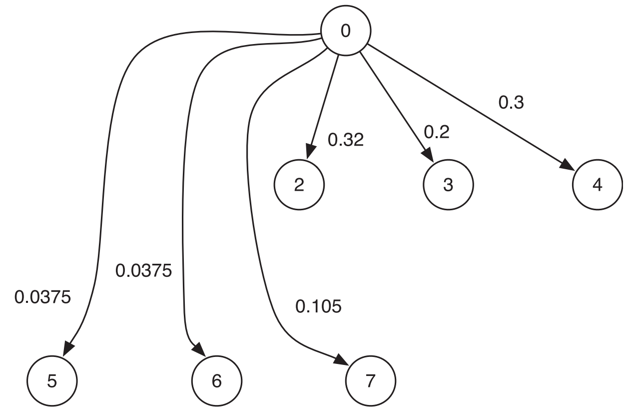
A related notion is that of *weak* combined steps. A weak combined step is essentially a product of many combined steps where at most one of them is via a non-internal action, while the rest are internal.

Definition 3.22 (Internal combined step [10]). *Let M be an SPA, $s \in S_M$ and $\delta_{IC} \in D(S_M)$. (s, δ_{IC}) is an* internal combined step *if either*

1) *$\delta_{IC}(s) = 1$; or*
2) *there exists a combined step (s, δ_C) such that for every $(a, t) \in A_M \times S_M$ such that $\delta_C(a, t) > 0$ it holds that*
   a) *$a \in A^H_M$;*
   b) *there exists an internal combined step $(t, \delta_{(s,a,t)})$ noted $step(s, a, t)$; and*
   c) *for every state $s' \in S_M$, $\delta_{IC}(s') = \sum_{(a,t) \in A_M \times S_M} \delta_C(a, t) * \delta_{(s,a,t)}(s')$; where $\delta_{(s,a,t)}$ is the distribution given by the combined step $step(s, a, t)$.*



(a) Original transition distributions



(b) Combining internal distributions

Fig. 5. An internal combined step.

Essentially, an internal combined step is a combination of subsequent combined steps where each combined step is such that it assigns non-zero probabilities only to internal actions. Fig. 5 shows an example of an internal combined step. In this case, all actions are hidden so no labels on transitions are necessary. Different transition distributions are told apart by the arc between the transitions. The combined transition depicted is obtained through an embedded distribution. This embedded distribution is the result of combining the distributions from state 0 with a factor of 0.5 on each distribution; and from state 1 using factors 0.3 (distribution shown on left) and 0.7 (distribution shown on right). In this case the combined step "skips" state 1.

There is a combination of combined steps and internal combined steps that is of important interest, which is the case when a state can be reached by any combination of exactly one action in $A^I_M \cup A^O_M$ and countably many interleavings of actions in A^H_M in between. We shall denote these as *weak combined steps*.

Definition 3.23 (Weak combined step [10]). *Let M be an SPA, $s \in S_M$ and $a \in (A^I_M \cup A^O_M)$. (s, a, δ_C) is a* weak combined step *if and only if there exists a combined step (s, δ'_C) such that every time that $\delta_C(action, state) > 0$ the following hold:*

1) *action* $= a \vee$ *action* $\in A_M^H$; *and*
2) *if action* $= a$ *then either* $\delta'_C(state) > 0$ *or else there exists an* internal *combined step denoted* $step(s, a, state) = (state, \delta'_{IC})$;
3) *otherwise, if action* $\in A_M^H$, *there exists a weak combined step denoted* $step(s, action, state) = (state, a, \delta_C)$;
4) *and finally, for every state* $t \in S_M$ *it holds that* $\delta_C(t) = \sum_{(action,state)\in A_M \otimes S_M} \delta'_C(state) * \delta_{s,action,state}(t)$, *where* $\delta_{s,action,state}$ *is the distribution of* $step(s, action, state)$.

Fig. 6a shows an example of distributions that can be combined as a weak combined step. Inside the arc corresponding to a distribution we note the triggering action. a is an action that is presumably shared with an external environment, while h is an internal action to the component we are modelling in this case. Fig. 6b shows the resulting weak combined step. In this case, we obtained this step by combining the first two transitions (originating from state 0) with factors of 0.5 each; on state 3 we use factors 0.3 and 0.7. In this case, the combination is far more complex, as hidden actions may appear before or after the action a, and even multiple times. However, it can easily be seen that the resulting step is much more simpler as well.

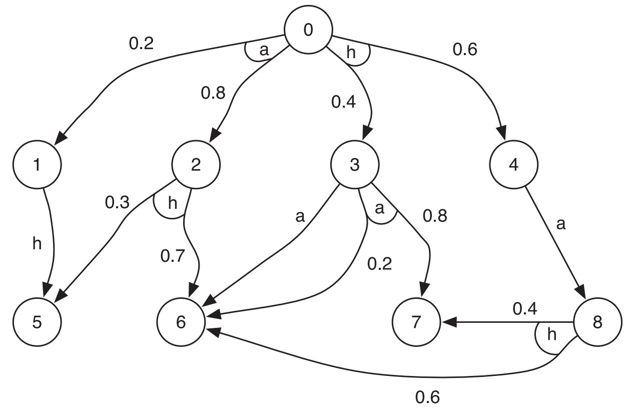
Definition 3.24 (Probabilistic branching simulation (PBS) [10]). *Given two Simple Probabilistic Automata* M_1 *and* M_2, *a* probabilistic branching simulation *is a relation* $\mathcal{R} \subseteq S_{M_1} \times S_{M_2}$ *such that*

1) *the initial state of* M_1 *is related through* $\mathcal{R}$ *with the initial state of* M_2;
2) *for each* $s_1 \mathcal{R} s_2$ *and each possible transition* $(s_1, a, \delta_1) \in R_1$ *then:*
   a) *if* $a \in A_{M_2}$, *there exists a weak combined step* (s_2, a, δ_2) *such that the distribution* δ_1 *can be embedded into* δ_2 *through* $\mathcal{R}$, *that is,* $\delta_1 \sqsubseteq_{\mathcal{R}} \delta_2$.
   b) *if* $a \notin A_{M_2}$, *there exists an internal combined step* (s_2, δ_2) *such that* $\delta_1 \sqsubseteq_{\mathcal{R}} \delta_2$.
3) *every time that* $s_1 \mathcal{R} s_2$, *it must be that if* $s_1 \xrightarrow{a_i}$ *for a set of actions* $a_i \in A_{M_1}$, *then* $s_2 \overset{a}{\Longrightarrow}$ *as well for at least one of these actions* a_i; *where* $s \xrightarrow{a}$ *denotes that there is a transition from* s *with action* a; *and* $s \overset{a}{\Longrightarrow}$ *denotes that* s *can* weakly transition *to some other state on action* a. *That is, it either has* a *enabled, or there is a path of internal transitions to a state where* a *is enabled. In other words, whenever* s_2 *weakly enables some actions, at least one of them must be weakly enabled in* s_1. *This establishes a liveness condition.*[1]

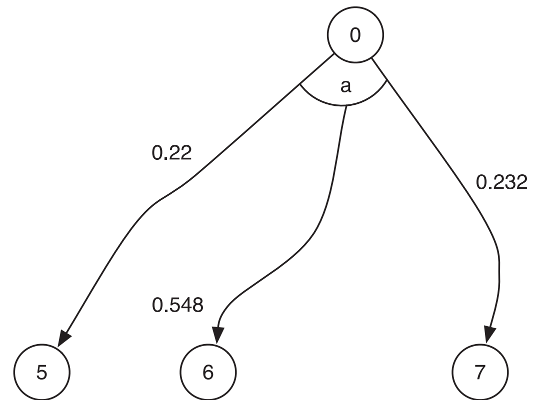
Whenever there exists such a simulation relation $\mathcal{R}$ *between* M_1 *and* M_2 *we will say that* M_2 simulates M_1, *and note it* $M_1 \sqsubseteq_{\mathcal{R}} M_2$ *(or succinctly* $M_1 \sqsubseteq M_2$ *if we do not care about the particular relation* $\mathcal{R}$*).*

Property 3.1 (Reflexivity and Transitivity [10]). *Probabilistic branching simulations are both reflexive and transitive.*

1. In [10] liveness is required on *every* action, although it is mentioned that it can be relaxed in the way we state here.



(a) Original transition distributions



(b) Combining a action and internal distributions

Fig. 6. A weak combined step on action a.

3.2.2 Logics for Property Description

In order to express and analyse properties over probabilistic models such as SPAs, these automata are coupled with modal logics whose formulae express said properties. For the specific case of probabilistic models, the temporal logic pCTL* [25] has been introduced as an extension of the well known temporal logic CTL*. Essentially, pCTL* replaces path quantifiers present in CTL* for *probabilistic quantification bounds* on the related path formulae.

pCTL* Syntax and Semantics. pCTL* formulae are built from state and path formulae, just as CTL*. Let AP be a finite set of atomic propositions. If ϕ stands for a state formula, and ψ for a path formula, then pCTL* formulae are built as follows

$$\begin{array}{c} \phi \rightarrow true \,|\, a \in AP \,|\, \neg\phi \,|\, \phi \wedge \phi \,|\, P_{\sim p}\psi \\ \psi \rightarrow X\phi \,|\, \phi U \phi \,|\, \phi U^{\leq k}\phi \end{array}.$$

In the above, $\sim \in \{<, \leq, =, \geq, >\}$ and $p \in \mathbb{R}, p \in [0, 1]$. Given an SPA Q and a mapping of states to atomic propositions $V : S_Q \rightarrow 2^{AP}$ defining the subset of atomic propositions that are valid for each state, we can define the satisfaction of pCTL* formulae by a state $s \in S_Q$, a scheduler $\sigma \in Sched(Q)$ and an execution fragment $\alpha \in fragments(Q)$ as follows

$$
\begin{aligned}
s,\sigma \models true &\Leftrightarrow \text{true}\\
s,\sigma \models a &\Leftrightarrow a \in V(s)\\
s,\sigma \models \neg\phi &\Leftrightarrow \neg(s,\sigma \models \phi)\\
s,\sigma \models \phi_1 \wedge \phi_2 &\Leftrightarrow (s,\sigma \models \phi_1) \wedge (s,\sigma \models \phi_2)\\
s,\sigma \models P_{\sim p}\psi &\Leftrightarrow \sum_{\alpha\in\psi_{sat}} \delta(C_\alpha,\sigma,Q) \sim p,\\
&\quad \text{where } \alpha \in \psi_{sat} \text{ iff } \alpha,\sigma \models \psi \text{ and}\\
&\quad \text{for every other } \alpha' \in \psi_{sat} \text{ neither}\\
&\quad \alpha \leq \alpha' \text{ nor } \alpha' \leq \alpha.\\
\alpha,\sigma \models X\phi &\Leftrightarrow \alpha^s_1,\sigma \models \phi\\
\alpha,\sigma \models \phi_1 U^{\leq k}\phi_2 &\Leftrightarrow \exists 0 \leq i \leq k \cdot \alpha^s_i,\sigma \models \phi_2 \wedge\\
&\qquad \forall 0 \leq j < i \cdot \alpha^s_j,\sigma \models \phi_1\\
\alpha,\sigma \models \phi_1 U \phi_2 &\Leftrightarrow \exists 0 \leq k \cdot \alpha,\sigma \models \phi_1 U^{\leq k}\phi_2.
\end{aligned}
$$

For convenience, we will note that a SPA Q and scheduler σ satisfy a formula (noted $Q,\sigma \models \phi$) if the initial state s^0_Q is such that $s^0_Q,\sigma \models \phi$. We can also generalise the satisfaction relation to set of schedulers. Given a set of schedulers $\Sigma \subseteq Sched(Q)$, we say Q satisfies ϕ under Σ, noted $Q,\Sigma \models \phi$ if for every $\sigma \in Sigma$, $Q,\sigma \models \phi$. Further, whenever $\Sigma = Sched(Q)$ we simply note $Q \models \phi$.

It is interesting to note that satisfaction verification of a pCTL* formula can be reduced to a reachability problem coupled with an optimization problem if more than one scheduler is possible [25]. Informally, given a path formula ϕ, a typical pCTL* state formula takes the form of a restricted classic CTL* state formula, but where path quantifiers have been replaced by the probabilistic operator $P_{\sim a}$. Thus, a state formula $P_{\leq a}\phi$ (resp. $P_{\geq a}\phi$), is true at a given state of the system if its possible evolutions from that state satisfy the formula ϕ with probability at most (resp. at least) a.

Note that whether a formula is satisfied or not by a SPA depends heavily on schedulers. Under two different schedulers, the same pCTL* formula may be satisfiable or not. This plays a critical role especially in the case of probabilistic operator formulae (that is $P_{\sim p}\psi$) as two different schedulers may assign distinct probabilities. In general, the scheduler is unknown when evaluating the satisfaction of a formula. Therefore, it is more interesting to know if a formula holds for *any* possible scheduler. In that case, for a probabilistic formula ψ, there will exist a scheduler σ^{ψ}_{min} that induces a *minimum* probability on the formula being satisfied; and another one σ^{ψ}_{max} (not necessarily distinct) that induces a *maximum* probability. Then, we will usually employ a different form of the probabilistic operator to query whether the minimum or maximum probabilities satisfy our requirements. We will usually replace the operator $P_{\sim p}$ by two other operators $P^{min}_{\sim p}$ and $P^{max}_{\sim p}$, which are evaluated globally for every scheduler. Satisfaction of these operators will be defined as follows:

$$
\begin{aligned}
s \models P^{min}_{\sim p}\psi \Leftrightarrow s,\sigma^{\psi}_{min} \models P_{\sim p}\psi\\
s \models P^{max}_{\sim p}\psi \Leftrightarrow s,\sigma^{\psi}_{max} \models P_{\sim p}\psi
\end{aligned}.
$$

It is important to note that there is a close relationship between pCTL* satisfaction and the notion of cones defined in Definition 3.17. We can see from the semantics definition of pCTL* that $s,\sigma \models P_{\sim p}\psi$ if the measure of the set of traces that satisfy ψ holds the relation $\sim p$. We have already established that cones induce a σ-algebra (in particular, a measure). The set of traces that satisfy ψ can be characterised by a (possibly infinite, but numerable) set of disjoint cones, based on the prefixes of the traces. Therefore, the set of traces that satisfy ψ has a definite measure induced by the cones that characterise it.

Finally, note that in the context of this work we will focus on a restriction of pCTL*, namely its weak fragment, which we denote as WpCTL*. A WpCTL* formula is restricted in the sense that the X and $U^{\leq p}$ operators are prohibited. Such a restriction is reasonable when the aim of the approach is to allow further refinement by modelling internal computation of components. The *next* and *bounded until* operators, which we choose to avoid, distinguish models based on these internal computations. However, from the point of view of an external observers, such internal computation should not be discernible.

3.2.3 Simulations and Property Preservations

There is a close relationship between automata that can be shown to be in a probabilistic branching simulation, and the sets of WpCTL* formulae that they satisfy. However, since an automata that simulates another will probably have more behaviour than the simulated one, it is necessary to take into account some precautions regarding fairness if we wish to study these sets of properties. As we will see, this idea has a close relationship to that of probabilistically fair schedulers 3.19.

Definition 3.25 (Probabilistically convergent automata [10]). *A Simple Probabilistic Automaton M is* probabilistically convergent *under a set of schedulers Sch if for every state $s \in S_M$ and $\sigma \in Sch$, the probability of diverging (that is, performing infinitely many internal actions and no input or output actions) from state s is 0.*

Definition 3.26 (Induced subgraph, (Bottom) strongly connected component). *Given a digraph $G = (V, E)$ where V is the set of vertices and E the set of directed edges, the* subgraph induced *by $V_0 \subseteq V$ is the graph $G_0 = (V_0, E_0)$ such that $E_0 \subseteq E$ includes all edges between vertices in V_0, and no other edges.*

An induced subgraph $G_0 = (V_0, E_0)$ of G is a strongly connected component *(SCC) of G if every vertex in V_0 is reachable from every other vertex in V_0 through edges in E_0.*

An SCC $G_0 = (V_0, E_0)$ of G is a bottom SCC *(BSCC) if no vertex from V_0 can reach a vertex in $V \setminus V_0$ through edges in E.*

Proposition 3.1 (Convergence of SPAs). *Let M be a Simple Probabilistic Automaton such that its underlying graph has no BSCC whose edges contain only internal actions. Let Sch a set of probabilistically fair schedulers for M. Then, it holds that M is probabilistically convergent under Sch.*

Proof. The proof is immediate from the definition of probabilistically fair schedulers. The only way for an infinite sequence of internal actions to have a measure larger than zero is that there is only a finite number of probabilistic choices with probability less than 1. For having such a situation be possible, there should be only a finite number of non-deterministic choices made in favour of

input/output actions instead of internal actions. However, such a choice would be in direct violation of probabilistically fair schedulers, therefore no probabilistically fair scheduler may result in a divergent automaton.

Note that the requirement of M having no purely internal BSCCs is reasonable and stems from the fact that they are trivially divergent, and largely uninteresting from a modelling point of view, since they would model only unobservable behaviour. In the remainder of the paper assume that the SPAs under study comply with this requirement. □

Finally, we recall a central theorem from [20] regarding probabilistic branching simulations and convergent SPAs.

Theorem 3.1 (PBSs preserve WpCTL* [10]). *Let M_1 and M_2 be two SPAs and such that $M_1 \sqsubseteq M_2$. Let $\phi = P_{\geq p}\psi$ be a WpCTL* formula. Then, it holds that $M_2, \Sigma_2 \models \phi$ implies that $M_1, \Sigma_1 \models \phi$ as well, where the formula satisfaction is considered only under the subsets $\Sigma_1 \subseteq Sched(M_1)$ and $\Sigma_2 \subseteq Sched(M_2)$ of fair schedulers .*

In other words, Theorem 3.1 states that, under the conditions described, if the minimum probability of M_1 satisfying ψ is p, then the minimum probability of M_2 satisfying ψ is at least as much. Note that the theorem also applies to maximum probabilities, since the minimum probability $p_{\min}$ of satisfying a given formula is equal to $1 - p^{\neg}_{\max}$ where $p^{\neg}_{\max}$ is the maximum probability of satisfying the negation of that same formula. In the remainder of this paper, we will focus on fair schedulers, so we will note $M \models \phi$ to implicitly refer to satisfaction under the subset of fair schedulers Σ (that is $M, \Sigma \models \phi$).

This notion of probabilistic branching simulations and property preservation is central, as we will show that our approach is such that the composition establishes a probabilistic branching simulation between the components and the composite system, and therefore preserves WpCTL* behaviour. This will be stated in Theorem 3.1.

4 Probabilistic Interface Automata

In this section we present our new modelling formalism designed to overcome the shortcomings other probabilistic modelling formalisms have, as was discussed in Section 2.

4.1 Definitions, Relations with IA and SPA

Leveraging on the definitions presented in previous sections, we can attain our aim of merging the notion of SPAs with that of Interface Automata. As a way to attain this objective, we define *Probabilistic Interface Automata* based on SPAs.

Definition 4.1 (Probabilistic Interface Automata). *A* Probabilistic Interface Automaton *(PIA) is a tuple of the form $M = < S_M, s^0_M, A^I_M, A^O_M, A^H_M, R_M >$ where the sets A^I_M, A^O_M and A^H_M are mutually disjoint, and such that defining $A_M = A^I_M \cup A^O_M \cup A^H_M$ yields a Simple Probabilistic Automaton $M_{SPA} = < S_M, s^0_M, A_M, R_M >$.*

Therefore, a Probabilistic Interface Automaton is an SPA that shares the input, output and hidden action semantics from Interface Automata. Note that since a Probabilistic Interface Automaton must induce an SPA, then $R_M \subseteq S_M \times A_M \times D(S_M)$. Note also that a Probabilistic Interface Automaton A has an *underlying Interface Automata*, noted $A\downarrow$ and defined as follows:

Definition 4.2 (Underlying IA). *Given a Probabilistic Interface Automaton E, we define its* underlying Interface Automaton *as the classic Interface Automaton $E\downarrow = < S_{E\downarrow}, s^0_{E\downarrow}, A_{E\downarrow}, R_{E\downarrow} >$ such that $S_{E\downarrow} = S_E$, $s^0_{E\downarrow} = s^0_E$, $A_{E\downarrow} = A_E$ and for all $s, s' \in S_{E\downarrow}$, $a \in A_{E\downarrow}$, $(s, a, s') \in R_{E\downarrow}$ if and only if there exists a distribution $\delta \in R_E(s, a)$ such that $\delta(s') > 0$.*

Simply put, the underlying Interface Automaton of a Probabilistic Interface Automaton is a non-deterministic automaton with the same state and edge structure, where all probabilities have been *forgotten* and replaced by non-deterministic transitions, leaving all other information unchanged. Conversely, it is also worth noting that a classic Interface Automata can be embedded in a Probabilistic Interface Automata by restricting R_M to Dirac distributions. This definition is akin to that of *underlying graph* of Markov chains [20], but this definition makes explicit the fact that the obtained graph is an Interface Automaton.

The notion of underlying Interface Automaton turns out to be useful for a natural way to define Probabilistic Interface Automata composability.

4.1.1 Composability and Product

Definition 4.3 (Composability). *Given P and Q two Probabilistic Interface Automata, we will say that P and Q are* composable *if their underlying Interface Automata $P\downarrow$ and $Q\downarrow$ are themselves composable (see Definition 3.3).*

The concepts of execution fragments and schedulers still apply to Probabilistic Interface Automata. Since these automata can be directly embedded into an SPA, we will refer to the SPA definitions for these concepts while working with PIAs. Probabilistic Interface Automata product, however, does express some differences regarding the composition of the transition relation.

Definition 4.4 (Product). *Given P and Q two composable Probabilistic Interface Automata, their product $P \otimes Q$ is defined by the Probabilistic Interface Automaton:*

$$P \otimes Q = < S_{P\otimes Q}, s^0_{P\otimes Q}, A^I_{P\otimes Q}, A^O_{P\otimes Q}, A^H_{P\otimes Q}, R_{P\otimes Q} >,$$

where $S_{P\otimes Q}$,$s^0_{P\otimes Q}$, $A^I_{P\otimes Q}$, $A^O_{P\otimes Q}$ and $A^H_{P\otimes Q}$ are defined in the same way as Interface Automata composition. Its transition relation $R_{P\otimes Q} \subseteq S_{P\otimes Q} \times A_{P\otimes Q} \times D(S_{P\otimes Q})$ however, is constructed in the same way as it was constructed for SPAs (Definition 3.14).

Note that we are overloading the operator $\otimes$ to refer to all of IA, SPA and PIA composition. The specific meaning in each case, however, can be easily understood from the context in which we use the operator. Refer to Fig. 7 for an example of two-state composition, where $a?$ denotes a is an input action for the automaton, and $a!$ denotes it is an output. Unannotated actions are internal. Recall that we would like the notion of Probabilistic Interface Automata to exceed a syntactic notion and actually have an interesting semantic bearing, as otherwise its usefulness would be drastically reduced. We will see to this objective in Theorem 4.1.

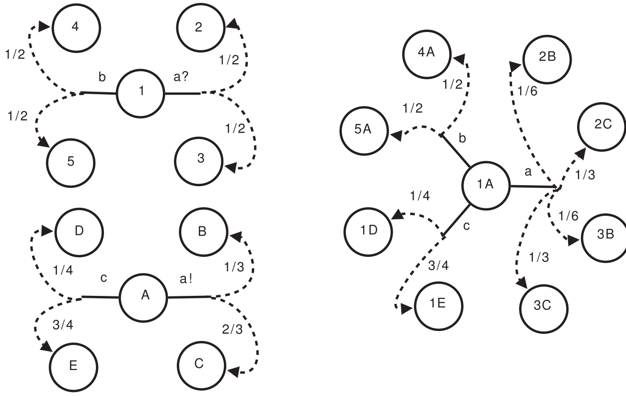


Fig. 7. Probabilistic interface automata (partial) product. Only the composite state 1A is shown.

Property 4.1 (Commutativity). *As is the case for Interface Automata, composition of PIAs is commutative, that is* $A \otimes B = B \otimes A$.

Property 4.2 (Composition and $\downarrow$ operator). *The probabilistic composition operator and the underlying Interface Automata operator are distributable over one another. That is, if P and Q are two Probabilistic Interface Automata, then* $(P \otimes Q) \downarrow = P \downarrow \otimes\, Q \downarrow$.

Property 4.3 (Composition preserves refinement). *Let A, B, C be PIAs and R a PBS such that $A \sqsubseteq_{\mathcal{R}} B$. Also, let $A \otimes C$ be legal as well as $B \otimes C$. Then, it holds that $A \otimes C \sqsubseteq_{\mathcal{R}} B \otimes C$.*

4.1.2 Illegal States and Valid Environments

The notions of illegal states and valid environments can also be extended for Probabilistic Interface Automata. In essence, they share the same definition, except for an important difference in the illegal states concept. As we discussed in Section 2, the original criteria for defining illegal states in the case of Interface Automata is too stringent, as it requires *immediate* enabledness of output actions in the component to be composed with.

In the following definition, we will make use of ACTL formulae over the underlying Interface Automaton of a given Probabilistic Interface Automaton P.

Definition 4.5 (Illegal states). *Given two composable Probabilistic Interface Automata P and Q, their product's illegal states are defined by the set $Illegal_{ProbIA}(P,Q) \subseteq S_P \times S_Q$. For any $s \in S_P$, $t \in S_Q$, $(s,t) \in Illegal_{ProbIA}(P,Q)$ if it is the case that either*

1) *for any action $a \in A^O_P \cap Shared(P,Q)$ enabled in s (respectively, actions $b \in A^O_Q \cap Shared(P,Q)$ enabled in state t) the ACTL formula $\forall(X_a) \vee (X_{A_Q \setminus Shared(P,Q)} True) \mathcal{U} (X_a True)$ does not hold for $Q \downarrow$ at state t under fair schedulers (respectively $\forall(X_b) \vee (X_{A_P \setminus Shared(P,Q)} True) \mathcal{U} (X_b True)$ does not hold on $P \downarrow$ at state s); or*
2) *s is such that its only enabled actions on P are a subset A_s of $A^I_P \cap Shared(P,Q)$ (respectively, enabled actions at t on Q are a subset A_t of $A^I_Q \cap Shared(P,Q)$) and the ACTL formula $\forall(X_{A_Q \setminus Shared(P,Q)} True) \mathcal{U} (X_{A_s} True)$ does not hold on $Q \downarrow$ at state t (respectively the formula $\forall(X_{A_P \setminus Shared(P,Q)} True) \mathcal{U} (X_{A_t} True)$ does not hold on P at state s) when being evaluated, restricting evaluation only to fair schedulers.*

Note that the semantics of the $\mathcal{U}$ operator above is that of a *strong until*. The difference between *weak until* ($\mathcal{U}_w$) and *strong until* is subtle and merits a reminder: an execution α satisfies the path formula $\psi \mathcal{U}_w \phi$ (that is, $\alpha \models \psi \mathcal{U}_w \psi$) if there exists an index i such that $\alpha^s_i \models \psi$ and $\forall 0 \leq j < i \cdot \alpha^s_j \models \phi$; or alternatively $\alpha^s_k \models \phi$ for every $k \geq 0$. The *strong until* is more stringent in the sense that it does not allow the second alternative, and it needs the step α^s_i such that $\alpha^s_i \models \psi$ to exist. In other words, the *strong until* demands the formula ψ to be true at some point, while *weak until* does not, as long as ϕ is never violated.

The illegal state definition for Probabilistic Interface Automata relaxes that of Interface Automata, so that synchronisation does not need to be available at each state, but may be *finitely* delayed, under certain conditions. Intuitively, the first clause (i) enforces the claim that states will only be legal if they allow an output action to be taken immediately; or else, if the current state is momentarily blocking it, it is such that every possible continuation of the trace from that state involves only internal actions of the blocking component until it allows the blocked behaviour to happen. However, it still is required that the synchronisation be carried out, regardless of any internal actions the delaying component takes. It must be noted that this future synchronisation delayed by a component cannot depend on action requirements by its counterpart. That is, a component may delay synchronisation *only* through the execution of internal actions, and every possible fair continuation of such execution fragments must eventually synchronise. Such restrictions are essential to further probabilistic analysis, because failure to eventually accept such behaviours would result in missing behaviour from the environment, along with its probability. Note that we refer to *fair* executions in the sense of *probabilistic fairness*. In other words the probability distributions that govern the transitions may allow for an indefinite delay of the required synchronization, but the probability of selecting this delay indefinitely should be zero (i.e., such a situation should *almost never* arise).

Clause (ii) in turn, describes that states that only allow for *shared input* actions are such that they must eventually always receive one of these input actions in order to advance. These are states that need to receive an input in order to advance (because the states themselves do not generate outputs and do not perform internal actions), and must be guaranteed to eventually receive one of these inputs and cannot be kept stuck forever. This second restriction essentially imposes an advancing condition on quiescent states of the components. On the one hand, this forces the counterpart component to actually have one of those actions as an output to be processed by the blocked component. On the other hand, fairness conditions are vital to ensure, additionally to the fact that the action must be available, that again the action is taken at some point in the future and is never indefinitely delayed.

These restrictions allow us to relax the stringent immediate blocking semantics, and let us model components' internal behaviour in a way that doesn't interfere with the synchronising semantics. Also, note that these conditions are not necessarily exclusive to Probabilistic Interface Automata. They can be used to relax the Interface Automata illegal states condition as well.

There is a price to pay, however, in the complexity of checking for legality. In the case of Interface Automata, this check was straightforward since it depends only on the states being composed. Legality checks of PIA may require $i)$ the whole composition to be built, and $ii)$ checking ACTL formulae satisfaction. We expand on this in Section 6.

4.2 PIAs and Property Preservation

In the case of Probabilistic Interface Automata, WpCTL* is a viable logic for property observation, since we can leverage on their underlying SPA structure and the scheduler definition (recall Definition 3.7). The main contribution of this paper is to convey the notion that the product of two interfacing probabilistic models is not merely a syntactic convenience, but that it does maintain a semantic relationship between the individual models, their composition, and their observable properties. The following theorem and its corollary see to this objective.

Theorem 4.1 (WpCTL* property preservation). *Let A and B be two composable Probabilistic Interface Automata such that their product $A \otimes B$ is legal (that is, it contains no reachable illegal states). Let ϕ_A be a WpCTL* property such that ϕ_A is expressed only in terms of the alphabet of actions in A. Then, if $A \models \phi_A$ under fair schedulers, then it holds that $A \otimes B \models \phi_A$ under fair schedulers as well.*

Informally, the theorem provides a validation for the compositional view of the component-composite model relation, as properties formulated early in the validation process do not lose their meaning once the components are integrated into a whole composite model. Intuitively, this is true, since the composition does not add new behaviour and neither does it prohibit allowed behaviour by the environment.

We delay for a moment proving the theorem and present a useful corollary regarding the extreme probabilities (minimum and maximum) of satisfaction of a given WpCTL* property.

Corollary 4.1 (Maximum and minimum scheduler probability). *Let A and B be defined as in Theorem 4.1. Let ψ_A be a WpCTL* formula and σ_A^{max} be a fair scheduler for A such that $A, \sigma_A^{max} \models P_{=p}\psi_A$, where $p \in [0,1]$. Further, let every other fair scheduler σ_A be such that $A, \sigma_A \models P_{\leq p}\psi_A$. In other words, σ_A^{max} yields the maximum probability of satisfying ψ_A on A.*

Similarly, let $\sigma max_{A\otimes B}$ be the scheduler that yields the maximum probability q of satisfying ψ_A on $A \otimes B$. Then, it holds that $q \leq p$.

This same corollary applies analogously to the minimum *probabilities of satisfying ψ_A.*

Proof. Suppose that it is not the case, that is $q > p$, or equivalently $q = p + r$ with $r > 0$. Recall from earlier on the paper that if σ_A^{max} yields the maximum probability of satisfying ψ_A on A, then it also provides the *minimum* probability of satisfying $\neg\psi_A$ on A, and that it is equal to $1 - p$. Similarly, $\sigma_{A\otimes B}^{max}$ yields the minimum probability of satisfying $\neg\psi_A$ on $A \otimes B$, with a value of $1 - q$.

Since σ_A^{max} yields the minimum probability $1 - p$ of satisfying $\neg\psi_A$ on A, then $A \models P_{\geq 1-p}\neg\psi_A$. Because of Theorem 4.1, it must be then that $A \otimes B \models P_{\geq 1-p}\neg\psi_A$. Yet the scheduler $\sigma_{A\otimes B}^{max}$ is such that it satisfies ψ_A with probability $1 - p - r < 1 - p$ on $A \otimes B$. Contradiction. □

We can now go back to the proof of Theorem 4.1.

Proof. Recall Theorem 3.1 that states that for two SPAs M_1 and M_2, and a WpCTL* formula ϕ it holds that if $M_1 \sqsubseteq M_2$, then $M_2 \models \phi \Rightarrow M_1 \models \phi$. Since in our setting A, B and $A \otimes B$ are PIAs, they are also SPAs. If we were to show that there exists a probabilistic branching simulation $\mathcal{R}$ such that $A \otimes B \sqsubseteq_{\mathcal{R}} A$, the theorem would be proved as a consequence of Theorem 3.1. □

We will show that $\mathcal{R}$ indeed exists by construction. We define $\mathcal{R} \subseteq S_{A\otimes B} \times S_A$ such that $(s,t)\mathcal{R}r$ if and only if $s = r$. We informally recall the four conditions of PBSs definition (see Definition 3.24) and show they are satisfied by $\mathcal{R}$ and we will prove each formally.

First, we check that the initial state of $A \otimes B$ is related through $\mathcal{R}$ with the initial state of A. The initial state of $A \otimes B$ is (s_0^A, s_0^B), the product of the initial states of A (s_0^A) and B (s_0^B). By definition of $\mathcal{R}$, $(s_0^A, s_0^B)\mathcal{R}s_0^A$.

Second, we check the simulation conditions on internal actions of $A \otimes B$ and those shared with A. Now take an arbitrary reachable state $(s,t) \in S_{A\otimes B}$. By definition of $\mathcal{R}$ it holds that $(s,t)\mathcal{R}s$. Consider the possible steps originating on (s,t) at $A \otimes B$, that is $R_{A\otimes B}((s,t)) \subseteq A_{A\otimes B} \times D(S_{A\otimes B})$. Let (a,δ) be an arbitrary transition on this set.

4.2.1 *Proving for an Action a Invisible to A*

If $a \in A_{A\otimes B} \setminus A_A$, then a is an action invisible to A (internal to $A \otimes B$). In this case we need to see that there exists an internal combined step (s, δ_{IC}) for A, such that $\delta \sqsubseteq_{\mathcal{R}} \delta_{IC}$. Define $\delta_{IC} = Dirac(s)$, that is, $\delta_{IC}(s) = 1$ and 0 everywhere else. To prove $\delta \sqsubseteq \delta_{IC}$, we refer to Definition 3.20. We need to show the existence of a weight function $w : (S_A \times S_B) \times S_A \rightarrow [0,1]$ such that

1) $\forall r \in S_A$, $\sum_{(x,y)\in S_A\times S_B} w((x,y),r) = \delta_{IC}(r)$;
2) $\forall (x,y) \in S_A \times S_B$, $\sum_{r\in S_A} w((x,y),r) = \delta(x,y)$; and
3) $w((x,y),r) > 0 \Rightarrow (x,y)\mathcal{R}r$.

We define the weight function w as follows:

$$w((x,y),r) = \begin{cases} \delta(x,y) & \text{if } x = r \\ 0 & \text{otherwise} \end{cases}.$$

We prove each condition on w individually. First, let $r \in S_A$. We compute $\sum_{(x,y)\in S_A\times S_B} w((x,y),r)$.

$$\begin{aligned} &\textstyle\sum_{(x,y)\in S_A\times S_B} w((x,y),r) = \\ &= \textstyle\sum_{y\in S_B} w((r,y),r) \text{ as } w \text{ is defined as 0 otherwise}. \\ &= \textstyle\sum_{y\in S_B} \delta(r,y) \end{aligned}$$

Now, recall that δ is a distribution arising from a transition on an action invisible to A. Therefore if the originating state was (s,t), only states of the form (s,t_i) will have

nonzero probability for δ. So, if $r \neq S$, $\sum_{y \in S_B} \delta(r, y) = 0 = \delta_{IC}(r)$ as δ_{IC} was 0 everywhere but s. If $r = s$, then $\sum_{y \in S_B} \delta(r, y) = \sum_{y \in S_B} \delta(r, y)$ which sums over the whole support set of δ, so equals to 1, which in turn is $\delta_{IC}(s)$.

Conversely, take an arbitrary $(x, y) \in S_A \times S_B$. Now, $\sum_{r \in S_A} w((x, y), r) = w((x, y), x)$ as w is zero otherwise. And $w((x, y), x) = \delta(x, y)$ by definition.

Finally, it is easy to see that if $w((x, y), r) > 0$ it must be that $r = x$. By definition of $\mathcal{R}$, $(x, y)\mathcal{R}x$, so the final point is proven.

4.2.2 Proving for a Shared Action a

In this case, we need to show the existence of a weak combined step (s, a, δ_{WC}) on A. Action a is obviously enabled on s as otherwise a would not synchronise and a would not be enabled on (s, t) either. Since a is a shared action, the distribution δ on $A \otimes B$ must have arisen from the product of a distribution δ_A on a transition from A, and a distribution δ_B on B. That is, for any $(x, y) \in S_A \times S_B$, $\delta((x, y)) = \delta_A(x) \times \delta_B(y)$.

In this case, we define $\delta_{WC} = \delta_A$, while w is defined in the same way as it was defined before. The conditions on w are proven in the same way as in the previous case.

4.2.3 Proving the Liveness Condition on Simulations

Finally, in order for $\mathcal{R}$ to be a probabilistic branching simulation, we need to show that whenever $(s, t)\mathcal{R}s$ and s enables a set of actions $A_A(s)$, then (s, t) weakly enables a set of actions $A_{A \otimes B}(s, t)$ with at least one action in common. The proof is a direct consequence of the fact that $A \otimes B$ has no illegal states. Assume $s \xrightarrow{o} s'$ for at least one *output* action o. Because of condition $i)$ on illegal states, every internal-action path on B must eventually enable action o to be illegal-state-free. Therefore, o is weakly enabled on $A \otimes B$.

Alternatively, suppose that $s \stackrel{i}{\Longrightarrow}$ only for internal actions i. In this case, because of condition $ii)$ on illegal states, B must weakly enable at least one of them, so enabledness on $A \otimes B$ is also guaranteed.

As an additional note, it is worth noting that composition, while preserving WpCTL* properties, may not actually preserve the *exact* event probabilities for a given property. For example, assume environment E satisfies the property $P_{\leq 0.75}\psi$. Recalling the formula satisfaction definition, this means that E satisfies ψ with probability at most 0.75 under the control of *any* scheduler. There may, or may not, be an actual scheduler that, when controlling E actually witness probability 0.75 for formula ψ. The interesting issue is that even if there is such a scheduler, the existence of a scheduler for $E \otimes S$ witnessing probability 0.75 for ψ is not guaranteed; in fact every scheduler for $E \otimes S$ may witness an inferior probability.

This distinction, however, is only important from a more formal point of view. In practice, if the approach is being used in a software engineering context, this distinction is not as important. For example, an engineer may be interested in proving that a given component has at most a 0.05 chance of failing. That is, the engineer poses the formula $P_{\leq 0.05}\mathit{failure}$, where *failure* is a formula capturing the conditions under which the component actually fails. The engineer then validates this formula over the component and finds it to be true. Then, it is guaranteed that the probability of this same component failing over the whole composition is at most 0.05. Further, suppose that in fact the engineer observes that the probability of failure of the isolated component is *exactly* 0.05. However, it may very well be that, because of behaviour restriction imposed by the composition, the exact failing probability drops to, for example, 0.03 or even zero in the composition. In any case, the reliability objective posed by the engineer, although it does not preserve the exact probability, is only reinforced by the composition. The failing probability never *increases* because of the composition, it can only decrease (and in fact, can only decrease down to the *minimum* probability of failure of the isolated component, and no further).

5 Case Study

In order to illustrate our approach, we analyse quantitatively the behaviour of an existing software system. This system is not probabilistic in nature, however this setting is not a limitation for our approach, as we can understand it as a probabilistic model comprised only of Dirac distributions.

However, we do have a probabilistic model of the environment with which this system interacts. We then set out to analyse the whole system specification in a modular way. That is, we will analyse first the behaviour of each component (the software system and its environment) in isolation. After this, we will study the system composition, and we will compare the obtained results.

In this case study, we analyse the impact of variations of the expected probabilistic behaviour of this environment, which is described using a Probabilistic Interface Automata. These environmental variations are then composed with the non-deterministic model of the software, and we produce bounds on the probability of environment-specific and system-specific properties holding. In each case, we verify the composability of the system/environment ensemble, and analyse and validate the property preservation characteristics of Probabilistic Interface Automata.

The software system we analyse is an extension of the case study presented in [26], which was further refined in [8]. In this paper, we further refine the models; in particular we relax the restrictions that were previously present regarding the immediate need for synchronisation, and allow for it to be delayed. This refinement preserves the observable behaviour of the original system, but the refinement is more complex as it models internal behaviour of the system. Of course, our approach could also be applied to the original system, although the analyses and findings would not be as interesting in that case.

The software under analysis is the *TeleAssistance* (TA) system; a web-based application providing remote assistance to patients that, for any reason, need to remain at their homes and need constant monitoring. In its most basic interaction, the patient commences operation via a `startAssistance` command. This puts TA in an infinite loop accepting any of the following requests:

- `stopMsg`, signaling the user wishes to cancel TA service for now.

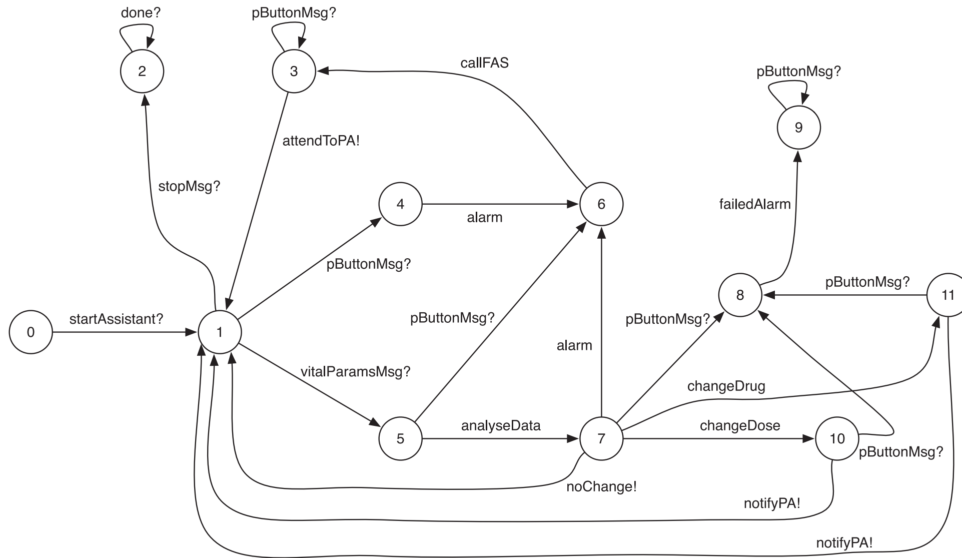


Fig. 8. The TeleAssistance software.

- `vitalParamsMsg`. This signal allows the user to send varied body readings via a supplied device. The patient's health parameters are analysed by the application server which, if necessary, may then suggest a course of action. The system may decide that a change in the patient's medication is needed, and communicates this decision via either the `changeDrug` or `changeDose` commands. If a successful adjustment is made, the patient is notified via the `notifyPA` message (with no details regarding the kind of adjustment made). If any anomalies are detected during the analysis, a First-Aid Squad (FAS) is requested and sent. In the case of a FAS being sent, the patient is informed via the `attendToPA` message.
- `pButtonMsg` allows the patient to activate a panic signal, if at any moment she begins to feel sick and cannot cope. This signal triggers an alarm in the TA service. A successful processing of the alarm results in a FAS being sent to the patient's home.

We have augmented the simplified model presented in [26] in two ways in order to introduce richer software-environment interactions. First, by specifying that for emergency reasons the panic button may be pushed at any operational state of the software, even if waiting for other results. Second, by refining the feedback provided by the software so that the patient is also told if no medication adjustment is needed. We depict an abstract model of the TA software in Fig. 8. Note that the model is as an Interface Automaton, which is a particular case of the Probabilistic Interface Automata introduced in this paper. As is customary, output actions are appended with '`!`', and input actions are appended with '`?`', while internal actions are left with no annotations.

The TA software exhibits a critical failure. this failure is reached by the triggering of the `failedAlarm` event. This happens if an alarm has been raised but it failed to be acknowledged or properly handled, thus not calling and sending the FAS. In this implementation, such an error (state 9) is reached if the user presses the panic button once the software has started analysing vital parameters' data. This event sequence was not properly foreseen by the implementation team. Relying on the software's model only, we can easily see that such a state is reachable. However, actual probability of reaching said state is highly dependent on both the environment's behaviour and timing races regarding the interaction between both the environment and the system. We now show how to model the probabilistic behaviour of the environment using Probabilistic Interface Automata, and how this resulting model and the theory presented in previous sections allow meaningful quantification of the probability of critical failures based on the modelled probabilistic assumptions of the environment.

5.1 Modelling the Environment

In Fig. 9 we depict a first attempt at modelling the probabilistic behaviour of the environment (in this case, the patient) of the TA software. The patient, when waiting for a vital parameters analysis response, probabilistically chooses to wait patiently or press the panic button. Also, it reflects a certain degree of anxiety in the patient's behaviour, since it behaves quite differently depending on whether the software determines to adjust her medication or not. If the medication is not adjusted, the patient reverts to its usual behaviour, however, if the medication is indeed adjusted, she becomes more prone to pressing the panic button.

Although seemingly a reasonable model of this environment, this is not the case. It is straightforward to see that Fig. 9 is a Probabilistic Interface Automaton (see Definition 4.1) and that it is composable (see Definition 3.3) with the TA system model (Fig. 8). However, Fig. 9 is not a valid environment (see Definition 3.6). The composite state consisting of state 9 in the TeleAssistance system model; and state 9 of the environment is an illegal pair (see Definition 4.5) that is reachable in the product (see Definition 4.4) of both models via the trace: $(0_s, 0_e)$ `startAssistant` $(1_s, 1_e)$ `choice` $(1_s, 2_e)$ `vitalParamsMsg` $(5_s, 5_e)$ `analyseData` $(7_s, 5_e)$ `choice` $(7_s, 8_e)$ `pButtonMsg` $(8_s, 5_e)$ `choice` $(8_s, 9_e)$ `failedAlarm`

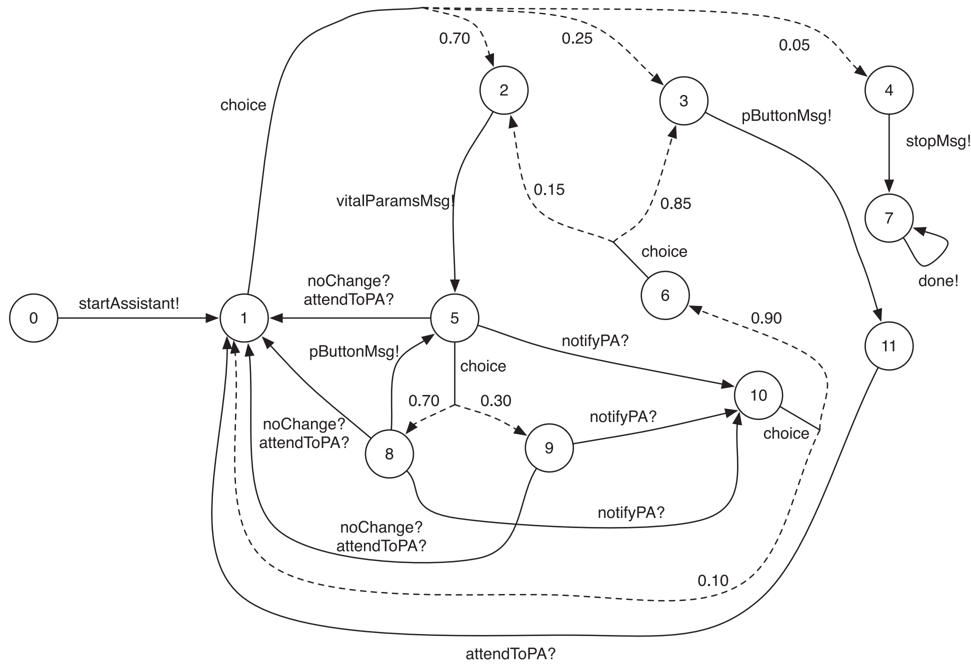


Fig. 9. An initial environment for the TA system.

$(9_s, 9_e)$. We have suffixed each state with either e or s to make clear whether we refer to the environmental or system state respectively.

The fact that $(9_s, 9_e)$ is an illegal state highlights that the environment is making incorrect assumptions on the behaviour of the system and renders the probabilistic environment behaviour modelled meaningless. For instance, analysing the behaviour of the probabilistic environment it is easy to conclude that the probability of sending a `vitalParamsMsg` to the system as the next message if being at state 9_e is at most 0.7, and at least 0.205 (the upper bound is obtained if the `noChange/attendToPA` transition is followed, while the lower bound is the result of the sum of the possible outcomes of taking the `notifyPA` transition). However, the same analysis on the product results in a probability inconsistent with the analysis on the environment alone. The inconsistency is that while, for the patient, the lower bound for the property was 0.205, the lower bound was decreased to zero (rather than increased) when composed with the system. The increase of the lower bound is due to the fact that the environment's behaviour specified in the environment's state 9_e is restricted when the system is in its own state 9_s, hence the environment probabilistic contribution that outgoing transitions from 9_e made to the lower bound of the property are no longer possible. However, this particular environment fails to makes a provision in modelling the possibility of such a restriction.

In summary, if the analyses performed to validate the probabilistic behaviour of the environment are not valid once the environment is composed with the software, then the model of the environment has a limited, if any, potential for sound analysis. The definition of legal environment, which the model in Fig. 9 does not satisfy, is aimed to guarantee sound analysis.

We can produce a legal environment for the TA by slightly modifying the current one. For example, a possible solution is to add `timeout` transitions from states 9_e and 11_e, modelling that the environment can give up waiting for the software response concluding that it has probably crashed in some way. That is to say, the previous model of the environment was establishing very strong assumptions on the system; the environment required the system to *always* generate an input at these states. This assumption, which turns out to be wrong, results in an illegal environment as it generates illegal states in the composition – see condition (ii) in Definition 4.5.

The property discussed with the initial probabilistic environment now evaluates to the interval $[0, 0.7]$ in this legal environment, and this is consistent with the evaluation of the property when composing the legal environment with the software. In fact, due to Theorem 4.1 we know that any property that has been used to validate the probabilistic behaviour in this legal environment will be preserved in its composition with the software. Asserting the validity of the conditions for legal environments essentially entails verification of several liveness properties. For this case study, however, we have chosen to check the legality of the environment directly by hand, given that the size of the model allows for such a manual analysis. Larger models would, of course, require an automated procedure for such validation.

5.2 Quantitative Analysis of the TA Software System

Since we have a legal environment for the TA software, we can now analyse quantitatively the behaviour of the TA software system by checking the probability of system properties holding when the TA software is composed with the legal probabilistic environment. The properties we considered for our quantitative analysis were taken from [26]. To perform the analysis we use the model checker PRISM [11], a well-known probabilistic verification tool.

TABLE 1
Varying $P^{error_1}_{\max}$ and $P^{error_2}_{\max}$ for Different Distributions

Run	R(1,choice)	R(5,choice)	R(6,choice)	R(10,choice)	$P^{error_1}_{\max}$	$P^{error_2}_{\max}$
1	$(2 \mapsto 0.7)$,$(3 \mapsto 0.25)$,$(4 \mapsto 0.05)$	$(8 \mapsto 0.7)$,$(9 \mapsto 0.3)$	$(2 \mapsto 0.15)$,$(3 \mapsto 0.85)$	$(1 \mapsto 0.1)$,$(6 \mapsto 0.9)$	0.9108	0.1435
2	$(2 \mapsto 0.7)$,$(3 \mapsto 0.25)$,$(4 \mapsto 0.05)$	$(8 \mapsto 0.7)$,$(9 \mapsto 0.3)$	$(2 \mapsto 0.99)$,$(3 \mapsto 0.01)$	$(1 \mapsto 0.1)$,$(6 \mapsto 0.9)$	0.9304	0.6727
3	$(2 \mapsto 0.7)$,$(3 \mapsto 0.25)$,$(4 \mapsto 0.05)$	$(8 \mapsto 0.7)$,$(9 \mapsto 0.3)$	$(2 \mapsto 0.15)$,$(3 \mapsto 0.85)$	$(1 \mapsto 0.95)$,$(6 \mapsto 0.05)$	0.9076	0.4707
4	$(2 \mapsto 0.7)$,$(3 \mapsto 0.25)$,$(4 \mapsto 0.05)$	$(8 \mapsto 0.2)$,$(9 \mapsto 0.8)$	$(2 \mapsto 0.15)$,$(3 \mapsto 0.85)$	$(1 \mapsto 0.1)$,$(6 \mapsto 0.9)$	0.7584	0.41
5	$(2 \mapsto 0.15)$,$(3 \mapsto 0.2)$,$(4 \mapsto 0.65)$	$(8 \mapsto 0.7)$,$(9 \mapsto 0.3)$	$(2 \mapsto 0.15)$,$(3 \mapsto 0.85)$	$(1 \mapsto 0.1)$,$(6 \mapsto 0.9)$	0.1441	0.105
6	$(2 \mapsto 0.15)$,$(3 \mapsto 0.2)$,$(4 \mapsto 0.65)$	$(8 \mapsto 0.7)$,$(9 \mapsto 0.3)$	$(2 \mapsto 0.15)$,$(3 \mapsto 0.85)$	$(1 \mapsto 0.95)$,$(6 \mapsto 0.05)$	0.1393	0.105
7	$(2 \mapsto 0.15)$,$(3 \mapsto 0.2)$,$(4 \mapsto 0.65)$	$(8 \mapsto 0.2)$,$(9 \mapsto 0.8)$	$(2 \mapsto 0.99)$,$(3 \mapsto 0.01)$	$(1 \mapsto 0.95)$,$(6 \mapsto 0.05)$	0.0458	0.0384

Consider the property that states that the FAS is always sent to the patient location when the alarm has been raised. Clearly, the TA software does not satisfy this property (see transition from 8 to 9 in the system model). However, it is interesting to quantify the probability of such error under the assumption of a particular probabilistic behaviour of the environment. To do this we first computed the product of the Interface Automaton for the TA software and the Probabilistic Interface Automaton modelling the environment and then using PRISM we quantify the occurrence of the error which can be characterised by the CTL formula $error = (\text{true}\,\mathcal{U}\,\text{TA.state} = 9)$. Recall that, because of non-determinism in the TA software, we will not obtain a single probability as the event measure, but rather an interval of where the probability lies.

In the product of the TA software and its legal environment, the lower and upper bounds for reaching $error = (\text{true}\,\mathcal{U}\,\text{TA.state} = 9)$ can be characterised as $P^{error_1}_{\min} = \max_x\{P_{\geq x}(\text{true}\,\mathcal{U}\,\text{TA.state} = 9)\}$ and respectively $P^{error_1}_{\max} = \min_x\{P_{\leq x}(\text{true}\,\mathcal{U}\,\text{TA.state} = 9)\}$. Computing these values in PRISM yields $P^{error_1}_{\min} = 0$ and $P^{error_1}_{\max} \sim 0.9108$. Note that the lower bound does not convey much information because of the highly non-deterministic nature of the TA software model. There is always a non-deterministic possibility that the error is *never* reached. In other words, there is a scheduler that always avoids the error. If information were available on the probabilistic behaviour of the internal choices of the TA software (e.g., reliability information) then the lower bound for this property could be non-zero.

The (rather high) value of $P^{error_1}_{\max}$ is sensitive to the probabilistic behaviour of the environment, namely the four probabilistic transitions labelled with `choice` (states 1, 5, 6 and 10). Varying the probabilities on these transitions helps understand the impact of the probabilistic environment on the system behaviour. Table 1 summarises a few analyses where these probabilities have varied and how these variations affect the final value of $P^{error_1}_{\max}$.

The table shows that the value of $P^{error_1}_{\max}$ decreases noticeably when the probability of exiting the assistance system (that is, transitioning from state 1 to 4 on `choice`) is increased (see row 5). This is sensible as the property being checked has an unbounded until operator, meaning that we are interested in the occurrence of errors no matter how long the system runs. Hence, the longer the system runs, the more likely it is to fail; and so the more likely the environment chooses to stop the software (transition 1 to 4) the less likely the probability of error, bringing $P^{error_1}_{\max}$ down.

Another interesting analysis, taken from [26] is to understand the probability of the following scenario occurring: "if a `changeDrug` or `changeDose` occurs, the next message received by the TA generates an alarm which fails". Rather than using a pCTL formula, we modelled this property by means of an observer Interface Automaton that flags this scenario as an error. Table 1 also summarises the results obtained for this error measure, $P^{error_2}_{\max}$ for the same distributions as before, showing the relationship between distributions, and the differences with $P^{error_1}_{\max}$.

Summarising, in this section we have shown how Probabilistic Interface Automata supports carrying out quantitative analysis of models in a compositional way. The notion of legal environment and its related theorems are crucial, as they constrain the acceptable models of the probabilistic behaviour of the environment to those that ensure that analysis performed to validate the environment's probabilistic behaviour is sound and preserved when analysing the composite system.

6 Discussion and Related Work

In this work we have dealt with the problem of enriching behaviour models in the shape of automata with probabilistic information. Our work is based on the Interface Automata [9] formalism, which introduces an *optimistic* approach to composition by making explicit assumptions over its environment. These assumptions allow for a notion of refinement, given by weaker input assumptions and stronger output guarantees.

In this work we have taken the action segregation semantics of Interface Automata as a starting point for introducing probabilities into non-deterministic systems. One of the main results presented by Interface Automata is the *refinement relation* between different versions of the same component. In that sense, a more concrete version of the component (e.g., an implementation) *refines* a more abstract one (e.g., a specification). Our work also benefits from this refinement notion. In our case, however, we require that this refinement also takes into account the probabilistic semantics of the component. The concept of probabilistic branching simulations (Definition 3.24) encompasses that of Interface Automata refinement and, additionally, it also establishes a refinement of probabilistic behaviour. In other words, if two Probabilistic Interface Automata A_1 and A_2 are in a refinement relation given by a probabilistic branching simulation (that is, $A_1 \sqsubseteq A_2$), then it is also the case that their underlying Interface Automata (Definition 4.2) $A_1\downarrow$ and $A_2\downarrow$ are also in a refinement relation as defined by Interface Automata (that is, $A_2\downarrow$ refines $A_1\downarrow$).

Additionally, regarding the notion of compatibility and composability of Interface Automata, we have taken only the

initial definitions from [9]. In that work, the notion of compatibility is also extended to *pairs* of components, in the sense that a pair (A, B) components are compatible if there exists a third Interface Automaton (the environment) that is composable with the product $A \otimes B$. In other words, they establish that A and B are compatible is there exists an environment that can link their behaviours together. We find this introduction of a third component unnecessary, hence we define compatibility in terms of the original two automata.

The idea of the third environmental component is significant if the aim of the work is, for example, to *synthesise* these environments. This synthesis is not the focus of the present work, although it is a subject of interest for future research.

Finally, it is important to note that the notion of weak probabilistic simulations subsumes that of alternating simulations in Interface Automata. Further, in this paper we have extended these simulations with weak semantics, which further extends the applicability of Interface Automata to an engineering process where model refinement with internal actions is desirable as a means to both converge to the design of a software component, and provide several views of the same component at different layers of abstraction.

Apart from these concepts inherited from Interface Automata, the focus of our work is the mechanism through which probabilistic information is added to non-deterministic models, in a way that it guarantees a sensible composition semantics. In this sense, it is worth pointing out that in the last few decades researchers have paid attention to the concept and consequences of operational profiles in system reliability specification and analysis [2], [27], [28], [29], [30].

One way to enrich models with probabilistic information is to analyse and quantify the influence that components exert over one another as well, and that can be observed as emergent behaviour on the composition. For example [26], [31] have proposed using a sample space of runs obtained from an existing similar system as the source of probabilistic behaviour. Then, using an algorithm that summarises this information, a non-probabilistic automata model of the composite system and environment is annotated with the obtained probabilities. The aim of this work is to be able to account for complex quantifiable interactions, in which the history of execution affects the probabilistic behaviour of an interacting component. Unlike our work, this approach is not suitable to building a system in a modular fashion, as the annotation is performed over the whole model. This yields a verification artefact that is a single model containing all the relevant probabilistic transition information, both pertaining to the environment and to the system. Also, Markov models such as these are purely probabilistic, which may not allow us to fully model concurrent systems' nondeterministic behaviour.

Although operationally intuitive, annotation approaches, such as the ones mentioned, lack a declarative characterisation of the resulting annotated composite model and its relation to the source of probabilities. In addition, and more importantly, they lack a notion of property preservation regarding the source of the annotations. This is crucial as, whatever the source of probabilities for the annotation is, this source must have been validated according to some criteria. If the annotation algorithm does not preserve such criteria, then little can be said about the validity of analysis performed on the probabilistically annotated composite model.

Another problem we tackle with our approach is that of mixing non-deterministic and probabilistic information on a single formalism. The problems arising from trying to mix this information in a meaningful way are not new, and approaches to either achieving or avoiding this mixed specification have been studied. For example, *generative* models [4] aim to avoid mixing non-determinism and probabilities. They do so by disallowing non-determinism to be present on transitions. However, an asynchronous parallel composition (á la CSP [32]) induces such non-determinism and must be dealt with. Works such as [6] advance in this direction resorting to redistributing probabilities when finding synchronising actions with no matching counterpart. It is unclear if this approach is suitable when the probabilities reflect system-environment interaction—the environment (in the most usual case, a user) may not actually redistribute probabilities on allowed actions when the desired one is not allowed. Regarding purely *reactive* models [5] the reader can refer to Section 2 to understand limitations regarding our goals.

A compromise can be met by choosing to model outputs in a generative way, while inputs are modelled reactively [3], [33]. This approach however, has several limitations. First, it does not allow non-determinism on output actions, as they must be strictly generative. Second, it also requires input-enabledness in a way similar to Input/Output automata. In [33] it is noted that input-enabledness can be skirted, but doing so imposes the restriction that models may not have any input-input or input-output races. On the one hand this limits the expressibility of models that can be represented, and on the other hand such an approach ends up modelling the scheduling of actions in an explicit way.

It must be noted that an important precedent to this work is that of probabilistic Input/Output automata [3]. This model enriches classic Input/Output automata [12] with probabilities, establishing a hybrid between the generative and reactive models, since output actions are modelled in a generative way while input actions are modelled reactively. The approach in itself is interesting, but the probabilistic Input/Output automata model has some characteristics we consider problematic. In the first place, it inherits from Input/Output automata the notion of input enabledness, that is, every component automaton, at every state, must allow every possible input as a transition. As we previously argued, this is not a realistic restriction in most cases, since systems are usually designed with some concept of the environment in mind, and thus it is reasonable that they restrict some inputs at certain points of execution. Another characteristic aspect of probabilistic Input/Output automata is that they introduce a real-valued parameter to each state in each component automaton. This parameter, an additional random variable as it happens, models a *delay* on each automaton state. The rationale for this delay is the need to somehow resolve conflicting races, since at some points when composing it would be feasible for more than one component to synchronise its actions. This delay then establishes an order in which the automata advance, that is, the automata in which the state delay is the least will advance first. This notion of resolving races between competing transitions is also present in our model, as in other proposed

models [7]. However, this choice is represented by an external entity, the scheduler. The scheduler has no dependence on the model itself, and the model behaves independently of the scheduler. Additionally, the notion of a scheduler models an unknown within the system under analysis. That is, it models a behaviour that cannot be explicitly quantified; the Input/Output automata notion of delay defeats this modelling objective. In that sense, we argue that the idea of a built-in scheduler as a composite aspect of the system model-be it probabilistic or not-is undesirable, as we aim to a separation of concerns. Finally, a behaviour composability result is presented for probabilistic Input/Output automata, though it is different to the one we present in this paper. Probabilistic Input/Output automata behaviour preservation stems from that of the original non-probabilistic Input/Output automata. This result states that every execution trace in the composite automata, when restricted to the actions of each component automaton, is an execution trace of said component automaton. However, this result leverages heavily on the embedded scheduler concept depicted above. Our result does not establish such a stringent relation, since we establish that system-environment composition does refine the specified behaviour, but observed probabilistic behaviour in the environment is still preserved, thus allowing for early elicitation of interesting properties.

Apart from modelling system behaviour by means of synchronising automata, there have also been advances in quantitative contract-based modelling or, in a similar fashion, quantitative assume-guarantee reasoning. The work by Delahaye et al. [34] presents a contract-based approach that shares many similarities with the work we present in this paper. In particular, both works aim at presenting a formalism that can reason about isolated components in the context of a composite systems. There exist two key differences between the approach presented here and that of [34]. First and foremost, the object of study in [34] are contracts which are represented by sets of traces while our work deals with automata-like description of behaviour. This makes the approach in [34] unable to reason about branching non-deterministic behaviour. The use of traces allows them to define composition and conjunction between systems (by composing or conjuncting their contracts), while also allowing for a notion of refinement between systems (that is, contracts that refine other contracts that otherwise allow less or require more). In turn, we do provide the notion of composition, but where conjunction does not have a direct analogue. Our choice of automata as models allows for explicit representation of non-deterministic choices and permits a larger degree of expressibility than that of the contracts of Delahaye et al. In that sense, our approach is closer to modelling formalisms such as Segala's Probabilistic Automata and Markov Decision Processes than those of contracts. As an additional difference, the work in [34] analyses contracts in isolation, and results in a lower bound for the probability of satisfying the contract that results of the composition of these contracts. Our approach is also intended for the isolated analysis of components; however we introduce a notion of preservation of WpCTL* properties rather than bounds.

There is also work on assume-guarantee verification of safety properties, which have some similarity to our own. The work of Kwiatkowska et al. [35] is noteworthy. In that work the authors model probabilistic systems through automata much like those presented here, and perform an assume-guarantee analysis on properties. This approach, however, limits itself to safety properties which are represented via deterministic automata. Other work [36] also presents an assume-guarantee approach where the object of study are Interactive Markov Chains [37]. These models, however, establish a segregation between probabilistic and labelled actions that does not allow for modelling synchronisation on actions triggered by different probability distributions.

The notion of refinement in automata-based formalisms is related to that of simulation (and bisimulation). Being that our PIAs are a restricted case of Segala's Simple Probabilistic Automata [20], the notion of (bi)simulation is well-defined. However, bisimulation can be too strict, and not an effective notion, in the presence of components with internal computation that needs to be abstracted away. In regards to this question, the notion of *weak bisimulation* [24] has been employed effectively in the context of non-probabilistic systems. Such a notion of weak bisimulation has been recognised, although it is problematic for probabilistic systems [38], [39]. We do not go into detail in these aspects, however some interesting work includes [40] where the authors present a weak bisimulation notion along with a decision procedure, albeit focused on fully probabilistic systems alone. Also, [10] introduces a notion of weak bisimulation for systems exhibiting non-determinism, where the bisimulation proposed includes the potential generation of infinite probabilistic distributions representing all possible intermediate internal steps. Philippou et al. [41] and Cattani and Segala [42] attack this problem by restricting distributions to a certain class.

The second main result presented in this paper regards the synchronising conditions for Interface Automata. We found the synchronising conditions to be too strict regarding the immediate necessity for synchronisation. However, software systems that need to perform several internal actions before allowing inputs from its environment are commonplace. Such systems cannot be easily modelled with Interface Automata without abstracting away such internal behaviour, eliminating the possibility to observe this potentially interesting behaviour. In this paper, we have relaxed the need for immediate synchronisation in these cases, while requiring a notion of fairness on the schedulers allowed for the composite system. This decision allows for further analysis. Although the fairness conditions imposed are not esoteric or overly restrictive, it may be the case that they can be refined and further relaxed. Preliminary analysis has shown that the fairness requirement over some states may be relaxed in some cases-for example, loops made up purely of internal actions, that can be ignored if not allowed to happen-but a generalisation and proper characterisation remain as future work.

It must be noted that our approach calls for a composition operator that performs extensive checks to allow for this delayed synchronisation. If these checks took an inordinate amount of time or memory to be performed, the approach would suffer from applicability. Fortunately this is not the case. Automatically checking for Probabilistic Interface Automata illegal states during composition is

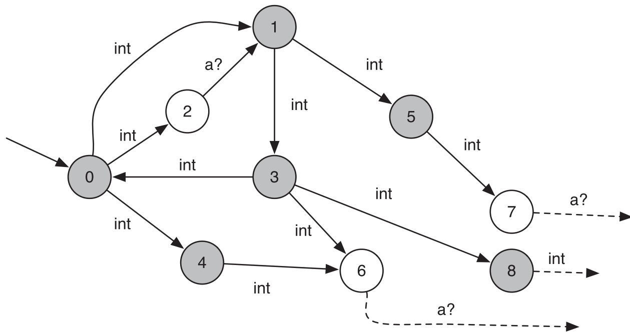


Fig. 10. Deferred blocking of an input action a.

similar to checking for illegal states in classic Interface Automata composition. Namely, at every point in the composition process it must be checked that the conditions for illegal states do not hold. For the case of Interface Automata, this is a completely local verification step, as we only need to validate that input actions are not blocked at any step. The complexity is thus polynomial in the size of the composite system, and it can be performed on-the-fly as the procedure for building the composite system is executed.

For the case of Probabilistic Interface Automata, the verification step is more involved, due to the fact that deferred synchronisation is allowed. At some points, it may be possible to check legality locally. For example, picture the case of a component that does not allow a certain input action a to be taken, nor it allows any internal actions of its own. This means that this action a is not enabled at this point, and will never be enabled through internal actions. Therefore, composing this component with another that intends to take its own output action a would be illegal. In this case, it would be easy to check illegality locally, on-the-fly.

However, in general, that may not be the case. Consider the example depicted in Fig. 10. Here we see a partial model of a component (dotted lines denote continuing executions). In this case, the situation is similar to that discussed before, as this component is still blocking input a at state 0 but, opposed to the situation before, it allows several of its internal actions (int) to be executed. In this case, it is not possible to check illegality locally; executions continuing from the grey states may *potentially* block action a at some point in the future, depending on the non-deterministic choices taken. It is necessary to verify the ACTL conditions of illegal states for all paths that sprout from taking one of these internal actions. In the worst case, this may require an ACTL check for every single state of the composition as it is being constructed. However, ACTL checking is still polynomial on the size of the system where it is being checked (which of course is smaller than the whole composite system), and the ACTL formula to check is of a constant size. Therefore, although illegal state checking for Probabilistic Interface Automata is harder than checking for Interface Automata, the complexity is still polynomial on the size of the composite system.

7 Conclusions

Quantitative model checking and analysis are promising techniques that complement Yes/No automatic analyses of behaviour. This approach naturally raises several formal and practical challenges. As a first challenge, it is important that these probabilities be introduced in a component-wise fashion. This is desirable because of two reasons. The first one is that it is often difficult to establish the quantitative behaviour of the system at large. In trying to do so, there is a risk that the engineer cannot properly validate that the introduced probabilities are result of a single component behaviour, and therefore these probabilities would lose their meaning. The second reason for component-wise modelling is that it is much easier for an engineer to reason individually about one component and then integrate the resulting model with other component models that were developed independently.

A second challenge is that introducing probabilities should not interfere with the behaviour that was described previously, that is, it should not preclude behaviour that was modelled and validated previously in an independent manner. In other words, the introduction of probabilities should be performed in such a way that already validated component-wise properties are preserved over the composite model.

The key to these challenges is a careful treatment of controllability of actions, non-determinism, and fairness assumptions over the behaviour of composite systems. We present Probabilistic Interface Automata as a suitable formalism satisfying these requirements and show that they are compositional, that is, there is a notion of property preservation between the components and the composite system. As a way to validate our claims, we present a case study along with our results obtained by the use of the technique. Although the case study presented leverages on manually generated environments, research on the generation of useful and sound environments is the focus of future and ongoing work.

Deeper understanding of fairness assumptions also merits further work. In the particular case of this paper, we have shown that a notion of strong fairness, relaxed for probabilistic behaviour, is sufficient to ensure compositionality of Probabilistic Interface Automata. However, it remains to be seen if such assumptions are completely necessary, or if they could be weakened. If so, further analysis is necessary for understanding under which conditions these assumptions may be weakened and what their impact is on modelling different environmental domains.

Acknowledgments

This work was partially supported by grants ANPCyT PICT 2011-1774, ANPCyT PICT 2012-0724, ANPCyT PICT 2013-2341, CONICET PIP 11220110100596CO, CONICET PIP 11220130100688CO, UBACYT 036, UBACYT 0384 and MEALS 295261. The authors would also like to thank Pedro D'Argenio and Holger Hermanns for their input and discussions on this paper.

References

[1] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[2] J. Musa, "Operational profiles in software reliability engineering," *IEEE Softw.*, vol. 10, no. 2, pp. 14–32, Mar. 1993.

[3] S.-H. Wu, S. Smolka, and E. Stark, "Composition and behaviors of probabilistic I/O automata," *Theoretical Comput. Sci.*, vol. 176, no. 1, pp. 1–38, 1997.

[4] I. Christoff, "Testing equivalences and fully abstract models for probabilistic processes," in *Proc. Theories Concurrency : Unification Extension*, 1990, pp. 126–138.

[5] R. van Glabbeek, S. Smolka, and B. Steffen, "Reactive, generative, and stratified models of probabilistic processes," *Inf. Comput.*, vol. 121, no. 1, pp. 59–80, 1995.

[6] P. D'Argenio, H. Hermanns, and J.-P. Katoen, "On generative parallel composition," *Electron. Notes Theoretical Comput. Sci.*, vol. 22, pp. 30–54, 1999.

[7] A. Sokolova and E. de Vink, "Probabilistic automata: System types, parallel composition and comparison," in *Validation of Stochastic Systems*. Berlin, Germany: Springer, 2004, pp. 1–43.

[8] E. Pavese, V. Braberman, and S. Uchitel, "Probabilistic environments in the quantitative analysis of (non-probabilistic) behaviour models," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 335–344.

[9] T. Henzinger and L. de Alfaro, "Interface automata," *ACM SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 109–120, 2001.

[10] R. Segala and N. Lynch, "Probabilistic simulations for probabilistic processes," *Nordic J. Comput.*, vol. 2, no. 2, pp. 250–273, 1995.

[11] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A tool for automatic verification of probabilistic systems," in *Proc. 6th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2006, pp. 441–444.

[12] N. Lynch and M. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *Proc. 6th ACM Symp. Principles Distrib. Comput.*, 1987, pp. 137–151.

[13] P. Halmos, *Measure Theory* (Graduate Texts in Mathematics Book vol. 18). New York, New York, USA: Springer, 1976.

[14] R. Keller, "Formal verification of parallel programs," *Commun.*, vol. 19, no. 7, pp. 371–384, 1976.

[15] C. Baier, M. Groesser, and F. Ciesinski, "Quantitative analysis under fairness constraints," in *Proc. 7th Int. Symp. Autom. Technol. Verif. Anal.*, 2009, pp. 135–150.

[16] M. Vardi, "Automatic verification of probabilistic concurrent finite state programs," in *Proc. 26th Annu. Symp. Found. Comput. Sci.*, Oct. 1985, pp. 327–338.

[17] C. Baier and M. Kwiatkowska, "Model checking for a probabilistic branching time logic with fairness," *Distrib. Comput.*, vol. 11, no. 3, pp. 125–155, Aug. 1998.

[18] E. A. Emerson and E. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons," *Sci. Comput. Programm.*, vol. 2, no. 3, pp. 241–266, Dec. 1982.

[19] R. De Nicola and F. Vaandrager, "Action versus state based logics for transition systems," *Semantics Syst. Concurrent Processes*, vol. 469, pp. 407–419, 1990.

[20] R. Segala, "Modeling and verification of randomized distributed real-time systems," Ph.D. dissertation, Massachusetts Instit. Technol., Cambridge, MA, USA, 1995.

[21] L. de Alfaro, "Formal verification of probabilistic systems," Ph.D. dissertation, Stanford Univ., Stanford, CA, USA, 1997.

[22] W. Feller, *An Introduction to Probability Theory and Its Applications*. New York, NY, USA: Wiley, 1968.

[23] V. Kulkarni, *Modeling and Analysis of Stochastic Systems*. Boca Raton, FL, USA: CRC Press, 2009.

[24] R. Milner, *Communication and Concurrency*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1989.

[25] A. Aziz, V. Singhal, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli, "It usually works: The temporal logic of stochastic systems," in *Proc. 26th Int. Conf. Comput. Aided Verif.*, 1995, pp. 155–165.

[26] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model evolution by run-time parameter adaptation," in *Proc. Int. Conf. Softw. Eng.*, 2009, pp. 111–121.

[27] R. Cheung, "A user-oriented software reliability model," *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 2, pp. 118–125, Mar. 1980.

[28] G. N. Rodrigues, D. S. Rosenblum, and S. Uchitel, "Sensitivity analysis for a scenario-based reliability prediction model," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005.

[29] K. Goševa-Popstojanova and K. S. Trivedi, "Architecture-based approach to reliability assessment of software systems," *Perform. Eval.*, vol. 45, no. 2-3, pp. 179–204, 2001.

[30] S. Yacoub, B. Cukic, and H. Ammar, "A scenario-based reliability analysis approach for component-based software," *IEEE Trans. Rel.*, vol. 53, no. 4, pp. 465–480, Dec. 2004.

[31] R. Roshandel and N. Medvidovic, "Toward architecture-based reliability estimation," in *Proc. ICSE Workshop Architecting Dependable Syst.*, 2004, pp. 2–6.

[32] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[33] R. M. Hierons and M. Núnez, "Testing probabilistic distributed systems," in *Proc. Formal Techn. Distrib. Syst.*, 2010, pp. 63–77.

[34] B. Delahaye, B. Caillaud, and A. Legay, "Probabilistic contracts: A compositional reasoning methodology for the design of systems with stochastic and/or non-deterministic aspects," *Formal Methods Syst. Des.*, vol. 38, pp. 1–32, 2011.

[35] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu, "Assume-guarantee verification for probabilistic systems," in *Proc. 16th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2010, pp. 23–37.

[36] H. Hermanns, J. Krčál, and J. Křetínský, "Compositional verification and optimization of interactive Markov chains," in *Proc. CONCUR - Concurrency Theory*, 2013, pp. 364–379.

[37] H. Hermanns and J.-P. Katoen, "The how and why of interactive Markov chains," in *Proc. 8th Int. Symp. Formal Methods for Components Objects*, 2009, pp. 311–337.

[38] H. Hansson and B. Jonsson, "A calculus for communicating systems with time and probabilities," in *Proc. 11th Real-Time Syst. Symp.*, 1990, pp. 278–287.

[39] S. Smolka and C.-C. Jou, "Equivalences, congruences, and complete axiomatizations for probabilistic processes," in *Proc. CONCUR Theories Concurrency: Unification Extension*, 1990, pp. 367–383.

[40] C. Baier and H. Hermanns, "Weak bisimulation for fully probabilistic processes," in *Proc. 9th Int. Conf. Comput. Aided Verification*, 1997, pp. 119–130.

[41] A. Philippou, I. Lee, and O. Sokolsky, "Weak bisimulation for probabilistic systems," in *Proc. 11th Int. Conf. Concurrency Theory*, 2000, vol. 1877, pp. 334–349.

[42] S. Cattani and R. Segala, "Decision algorithms for probabilistic bisimulation," in *Proc. 13th Int. Conf. Concurrency Theory*, 2002, pp. 371–386.

Esteban Pavese received both the undergraduate computer science degree and the PhD degree in computer science from the University of Buenos Aires, where he has also taught. He is an assistant researcher at Humboldt Universität zu Berlin. His research interests include modelling and analysis of uncertain and probabilistic behaviour, automated model learning, validation and verification, and adaptive systems.

Víctor Braberman holds a professorship at the Department of Computing, FCEN, University of Buenos Aires, and is a CONICET researcher working in the area of software engineering. He has headed the development of tools for the modelling and analysis of software intensive systems. His publication track record includes regular publications in the top software engineering conferences and journals. He has served as a PC member of several flagship conferences during the past 10 years. He acted as a coordinator for the CS grant proposal evaluation process at the Argentinean research funding agency. He has a significant industrial experience as a consultant and leads R&D projects with the software industry.

Sebastian Uchitel received the undergraduate computer science degree from the University of Buenos Aires and the PhD degree in computing from Imperial College London. He is a professor at the University of Buenos Aires, a CONICET researcher and holds a readership at Imperial College London. His research interests are in behaviour modelling, analysis and synthesis applied to requirements engineering, software architecture and design, validation and verification, and adaptive systems. He was an associate editor of the *IEEE Transactions on Software Engineering* and is currently an associate editor of the *Requirements Engineering Journal* and the *Science of Computer Programming Journal*. He was a program cochair of ASE '06 and ICSE '10, and will be a general chair of ICSE '17 to be held in Buenos Aires. He has been distinguished with the Philip Leverhulme Prize, an ERC StG Award, the Konex Foundation Prize, and the Houssay Prize.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.