

Formal Dependability Engineering with MIOA^{*}

Matthias Kuntz, Boudewijn Haverkort

University of Twente,
Faculty for Electrical Engineering, Mathematics and Computer Science

Abstract. In this paper, we introduce MIOA, a stochastic process algebra-like specification language with datatypes, as well as a logic intSPDL, and its model checking algorithms. MIOA which stands for Markovian input/output automata language, is an extension of Lynch’s input/automata with Markovian timed transitions. MIOA can serve both as a fully fledged “stand-alone” specification language and the semantic model for the architectural dependability modelling and evaluation language *Arcade*. The logic intSPDL is an extension of the stochastic logic SPDL, to deal with the specialties of MIOA. intSPDL in the context of *Arcade* can be seen as the semantic model of abstract and complex dependability measures that can be defined in the *Arcade* framework. We define syntax and semantics of both MIOA and intSPDL, and show examples of applying MIOA and intSPDL in the realm of dependability modelling with *Arcade*.

1 Introduction

Over the last decades, electronic and networked devices have become an important factor for our economy and daily live. Due to this ever increasing importance, it must be a key concern to assure that these devices are working correctly. Correct functioning encompasses besides purely functional correctness, i.e., a device does what it is expected to do, also quantitative aspects, such as performance and dependability. Where functional verification, such as model checking, has become a widely accepted and applied technique, the situation is still somewhat different for performance and dependability modelling and evaluation. Dependability evaluation must be achievable as a byproduct of the ordinary system design process, as short production cycles and tight cost objectives do not allow for additional costs (in terms of time and money) for system evaluation. Besides this requirement, a dependability evaluation formalism should have a formal semantics, high expressiveness, low modelling effort, and tool support. Over the last decades numerous dependability evaluation formalisms have been devised. All of them have shortcomings with respect to one or the other requirements mentioned here. In [4], with *Arcade*, we laid the foundations of a dependability evaluation which satisfies the mentioned requirements. In this paper, we introduce MIOA, the Markovian input/output automata language. MIOA provides the formal semantic model of *Arcade*. As special features, MIOA

^{*} This research has been partially funded by the Netherlands Organization for Scientific Research (NWO) under FOCUS/BRICKS grant number 542.000.504 (VeriGem)

possesses abstract data types and programming language constructs like loops and control structures. Abstract data types are useful in the context of *Arcade*, as they facilitate the definition of complex component repair, and replacement strategies. The new MIOA elements are also useful, when this language is used as a stand-alone specification language, as concise and more realistic system models can thus be built. We have also extended *Arcade*'s original capabilities of expressing dependability measures. Up to now, it was only possible to specify, in terms of Boolean expressions, under which circumstances a system is down. We provide a small language that allows to specify more complex dependability measures on the same abstract level as in *Arcade* dependability measures are defined. We have defined a temporal stochastic logic, on which these requirements are mapped. Thus, we can use the entire powerful model checking apparatus for dependability evaluation. Fig. 1 summarizes the application of MIOA and intSPDL in the context of the *Arcade* dependability evaluation framework. On top-level, with

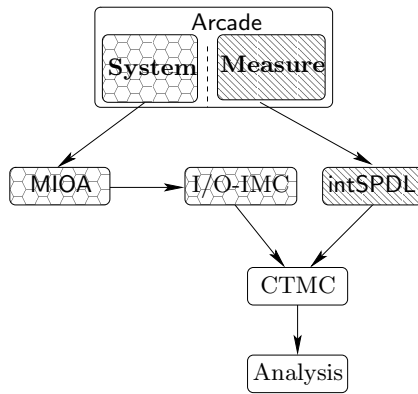


Fig. 1. Example of an I/O-IMC

Arcade, we can specify the dependability behaviour and the dependability measures of interest of the system, that is to be analysed. The system specification is mapped onto MIOA programmes, which in turn are mapped onto I/O-IMCs. The dependability measures are mapped onto intSPDL formulae. From the system's I/O-IMCs and the intSPDL representation of the current dependability measure that is to be verified, a simple continuous time Markov chain (CTMC) is derived, on which the actual dependability evaluation takes place.

Related Work We are aware of two approaches that extend process algebras, without stochastic timing, with abstract data types. LOTOS [3] provides data types for the use in process algebraic specifications. However, the synchronisation model applied in LOTOS and the lack of an appropriate stochastic extension render the application of LOTOS in our context impossible. For μ CRL resp.

mCRL2 [10, 9] data types have also been defined. But also here, no stochastic extension is available.

The paper is further organised as follows: In section 2 we introduce briefly, syntax and semantics of I/O-IMCs. Section 3 is devoted to syntax and semantics of MIOA. In section 4 we present the logic interactive SPDL (intSPDL) for MIOA. Section 5 is then devoted to the model checking algorithms for intSPDL. In section 6 we briefly present the dependability evaluation framework Arcade and show, how MIOA and intSPDL can be used in this context. Section 7 concludes this paper with a short summary and pointers to future research.

2 Input/Output Interactive Markov Chains and MIOA

In this section we introduce syntax and semantics of input/output interactive Markov chains (I/O-IMCs) as well as the MIOA language.

2.1 Input/Output Interactive Markov Chains

Input/output interactive Markov chains (I/O-IMCs) [5] are a combination of Input/Output automata (I/O-automata) [14] and interactive Markov chains (IMCs) [11]. I/O-IMCs distinguish two types of transitions: (1) *interactive transitions* labeled with actions; (2) *Markovian transitions* labeled with rates λ , indicating that the transition can only be taken after a delay that is governed by an exponential distribution with parameter λ . Inspired by I/O-automata, actions can be further partitioned into:

1. *Input actions* (denoted $a?$) are controlled by the environment. They can be *delayed*, meaning that a transition labeled with $a?$ can only be taken if another I/O-IMC performs an output action $a!$. A feature of I/O-IMCs is that they are *input-enabled*, i.e., in each state they are ready to respond to any of their inputs $a?$. Hence, each state has an outgoing transition labeled with $a?$. That means that each state is labeled with each input action the I/O-IMC has, such that it can always respond to a corresponding output message, even if this does not result in a state-change (such states have self-loops).
2. *Output actions* (denoted $a!$) are controlled by the I/O-IMC itself. In contrast to input actions, output actions cannot be delayed, i.e., transitions labeled with output actions must be taken immediately.
3. *Internal actions* (denoted $a;$) are not visible to the environment. Like output actions, internal actions cannot be delayed.

Graphically, states are depicted by circles, initial states by an incoming arrow, Markovian transitions by dashed lines, and interactive transitions by solid lines. Fig. 2 shows an I/O-IMC with two Markovian transitions: one from $S1$ to $S2$ with rate λ and another from $S3$ to $S4$ with rate μ . The I/O-IMC has one input action $a?$. To ensure input-enabling, we specify $a?$ -self-loops in states $S3$,

$S4$, and $S5$ ¹. Note that state $S1$ exhibits a race between the input and the Markovian transition: in $S1$, the I/O-IMC delays for a time that is governed by an exponential distribution with parameter λ , and moves to state $S2$. If however, before that delay ends, an input $a?$ arrives, then the I/O-IMC moves to $S3$. The only output action $b!$ leads from $S4$ to $S5$. We say that two I/O-IMCs

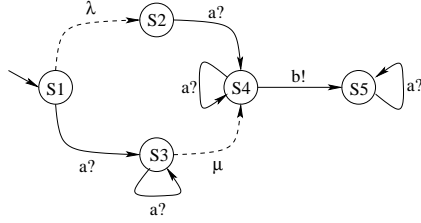


Fig. 2. Example of an I/O-IMC

synchronize if either (1) they are both ready to accept the same input action, or (2) one is ready to output an action $a!$ and the other is ready to receive that same action (i.e., has input action $a?$). I/O-IMCs can be combined with a parallel composition operator, denoted “ \parallel ”, to build larger I/O-IMCs out of smaller ones. The behavior of $P = Q \parallel R$, i.e., the parallel composition of I/O-IMCs Q and R , is the joint behavior of its constituent I/O-IMCs and can be described as follows:

1. If an action does not require synchronization then Q and R can evolve independently, i.e., if Q (R) can make any transition (interactive or Markovian) and behaves afterwards as Q' (R'), the same behavior is possible in the parallel context, i.e., $Q \parallel R$ can evolve to $Q' \parallel R$ ($Q \parallel R'$).
2. If an action of an interactive transition requires synchronization, then both I/O-IMCs Q and R must be able to perform that action at the same time, i.e., $Q \parallel R$ evolves simultaneously into $Q' \parallel R'$. Note that when an output and an input action synchronize the result is an output action.

Fig. 3 illustrates the parallel composition operator. Like in process algebras, the hiding operator $\text{hide } A \text{ in } P$ makes output actions in a set A internal, such that no further synchronization is possible over actions in A .

Formally, I/O-IMCs can be defined as follows:

Definition 1 (Interactive input/output Markov chains). *An interactive input/output Markov chain \mathcal{I} (I/O-IMC) is a $\mathcal{I} = (S, \text{in}(\mathcal{I}), \text{out}(\mathcal{I}), \text{int}(\mathcal{I}), \rightarrow, \text{----}, L, s)$ where:*

- S is a finite set of states.

¹ In the sequel we often omit these self-loops for the sake of clarity and simplicity of the I/O-IMC representation.

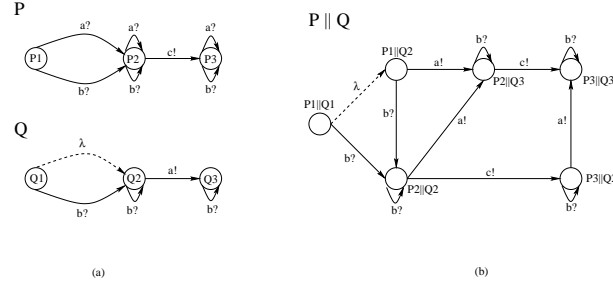


Fig. 3. Example of I/O-IMCs parallel composition

- $in(\mathcal{I})$ is the set of input actions of \mathcal{I} , an element of $in(\mathcal{I})$ is terminated by '?'.
- $out(\mathcal{I})$ is the set of output actions of \mathcal{I} , an element of $out(\mathcal{I})$ is terminated by '!'
- $int(\mathcal{I})$ is the set of internal actions of \mathcal{I} , an element of $int(\mathcal{I})$ is terminated by ';'.
- $\rightarrow \subset S \times Act(\mathcal{I}) \times S$ is the interactive transition relation.
- $\dashrightarrow \subset S \times \mathbb{R}^+ \times S$ is the Markovian transition relation.
- L is the state labelling function that associates with each state s' in \mathcal{I} the set of atomic properties that are true in s' , i.e., $L : S \mapsto 2^{AP}$. AP is the set of atomic properties defined over \mathcal{I} .
- s is the unique initial state of \mathcal{I} .

where $Act(\mathcal{I}) = in(\mathcal{I}) \cup out(\mathcal{I}) \cup int(\mathcal{I})$

In Section 4 the notion of paths in an I/O-IMC is crucial:

Definition 2 (Paths in I/O-IMCs). An infinite path σ of an I/O-IMC \mathcal{I} is a sequence of transitions of the form $s_0 \xrightarrow{a_0, t_0} s_1 \xrightarrow{a_1, t_1} \dots$, where:

- where applicable, see below, $t_i = \tau(\sigma, i)$, with $t_i \in \mathbb{R}^+$ is the positive real valued sojourn time in state s_i .
- If $s_i \xrightarrow{a_i, t_i} s_{i+1} \in \rightarrow$, then $t_i = 0$, as actions are not timed.
- If $s_i \xrightarrow{a_i, t_i} s_{i+1} \in \dashrightarrow$, then $a_i = \epsilon$, as Markovian transitions do not have an action labelling.
- $\sigma[i]$ is the $(i + 1)$ st state on path σ .
- $\sigma @ t$ is the state of path σ occupied at time point t .
- $a[i] = a_i$ is the $(i + 1)$ st action on path σ .

A finite path σ is a finite sequence of transitions of the form: $s_0 \xrightarrow{a_0, t_0} s_1 \xrightarrow{a_1, t_1} s_2 \dots s_{n-1} \xrightarrow{a_{n-1}, t_{n-1}} s_n$, where s_n is an absorbing state. For a finite path, $\tau(\sigma, i)$ is defined for $i < n$ as for infinite paths, and for $i = n$ we define $\tau(\sigma, i) = \infty$. The set $PATH^{\mathcal{M}}(s) := \{\sigma \mid \sigma[0] = s\}$ is the set of all finite or infinite paths with initial state s .

Let σ be a path in \mathcal{I} , then σ_{Act} is the path that stems from considering only transitions from \rightarrow , i.e., interactive transitions.

The semantics of I/O-IMCs is completely defined via the semantics of parallel composition and hiding.

Definition 3 (Semantics of parallel composition). *Let P and Q be any I/O-IMC specification, then the I/O-IMC parallel composition operator “ \parallel ” has the following semantics:*

1.
$$\frac{P \xrightarrow{\lambda} P'}{P \parallel Q \xrightarrow{\lambda} P' \parallel Q}$$
2.
$$\frac{Q \xrightarrow{\lambda} Q'}{P \parallel Q \xrightarrow{\lambda} P \parallel Q'}$$
3.
$$\frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q} \quad a \in \text{Act}(P) \wedge a \notin \text{act}(Q)$$
4.
$$\frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'} \quad a \notin \text{Act}(P) \wedge a \in \text{act}(Q)$$
5.
$$\frac{P \xrightarrow{a^?} P' \quad Q \xrightarrow{a^?} Q'}{P \parallel Q \xrightarrow{a^?} P' \parallel Q'} \quad a \in \text{in}(P) \wedge a \in \text{in}(P)$$
6.
$$\frac{P \xrightarrow{a^?} P' \quad Q \xrightarrow{a^!} Q'}{P \parallel Q \xrightarrow{a^!} P' \parallel Q'} \quad a \in \text{in}(P) \wedge a \in \text{out}(P)$$
7.
$$\frac{P \xrightarrow{a^!} P' \quad Q \xrightarrow{a^?} Q'}{P \parallel Q \xrightarrow{a^?} P' \parallel Q'} \quad a \in \text{out}(P) \wedge a \in \text{in}(P)$$

where:

- $\text{Act}(P)$ is the set of actions in P
- $\text{in}(P), \text{out}(P)$ are the input, resp. the output actions of P .

The hiding operator has the following semantics

Definition 4 (Semantics of hiding). *The semantics of the hiding operator can be defined as follows:*

1.
$$\frac{P \xrightarrow{a} P'}{\text{hide } b \text{ in } P \xrightarrow{a} P'} \quad a \neq b$$
2.
$$\frac{P \xrightarrow{a} P'}{\text{hide } a \text{ in } P \xrightarrow{\tau} P'}$$

3 The MIOA Language

3.1 Syntax

Each MIOA programme is in principle built as shown in Fig. 3.1. In line (1) we start the specification by assigning a name to the IO-IMC. $\langle \text{ioimc_name} \rangle$ is an arbitrary string. In lines (3) to (5) we define the actions that can occur in the IO-IMC. This section is opened by the keyword **signature**: in line (2). As usual, there are three types of actions: input, output and internal actions.

If we want to assess the reliability, dependability and availability of a system, we need means to express stochastic behaviour. To this end, we define in line (6) the rates relevant for the IO-IMC at hand. It is important to note, that rates $\langle rate_i \rangle$ can be either defined as a constant or as a real-valued variable. Defining rates as functions gives us higher modelling flexibility, as we can make, for example, the failure rate of a component dependent on the operational mode the system is in. This is highly desirable, as the failure rate of a component which serves as a spare may be smaller if it is not in its activated state. In the transitions: section of the MIOA programme we assign actual values to the rates, if rates are interpreted as variables.

In the section followed by keyword variables: (line (7)) we define the states, the IO-IMC consists of. States can be defined as being arbitrary (abstract) data types. The (abstract) data types are defined in a mathematical sound way [7, 8]. From line (10) (keyword transitions:) on, the transitions are described. In general, the transitions are given in a precondition-effect-style. That means, in order to execute a transition labelled with an action or a rate, a certain precondition must be satisfied. Preconditions are any expressions that can be evaluated either to true or false. As IO-IMCs are input-enabled, input-actions are always possible, i.e. the precondition is empty (or, equivalently, always true). To determine the effect of transitions in lines (12), (16), and (20) the result of taking a transition labelled with the action or rate associated with the particular transition can be described by employing any operation that is defined on the state's datatype that is affected by that transition. Using **if -then -else** it is possible to define different effects, depending on the value of preconditions. Repeated executions of transitions is possible by providing a **while** -loop construct. In line (26) the keyword **hiding** indicates that in the sequel (line (27)) follows the list of output actions that are to be hidden, i.e., excluded from synchronisation. Additionally, like in I/O-IMCs it is possible to combine two or more MIOA programmes in parallel, this is done by providing a MIOA programme, which consists of the name of the new programme, the signature, i.e. the set of the visible actions of the constituent MIOA programmes, and the keyword **parallel** followed by the names of MIOA programmes that are to be combined in parallel. It is also possible to hide actions from the environment. First, we will introduce the syntax of MIOA by means of a small example.

Example 1. Let the MIOA-specification of Fig. 1 be given. In line (1) the keyword IOIMC: is followed by a unique identifier (name) for the MIOA programme. In lines (3) to (5) the set of actions and rates of the programme are defined, the keyword signature: (line (2)) opens this section. Here, two types of actions are used: input, and output actions. In line (5), we do not assign a value to rate λ , thus it is interpreted as a variable.

In lines (6) to (10) the states of the MIOA programme are defined. All states are of data type **Bool**, thus they can take either the value **true** or **false**. All Boolean operators are defined on the abstract data type **Bool**. All states obtain an initial value, which is **true** in the case of “UP” and **false** in all remaining cases.

```

(1) IOIMC: < ioimc_name >
(2)   signature:
(3)     input: < action_1 >? ... < action_m >?
(4)     output: < action_1 >! ... < action_n >!
(5)     internal: < action_1 > ... < action_l >
(6)     markovian: < rate_1 > ... < rate_k >
(7)   variables:
(8)     < state_def >
(9)     ...
(10)  transitions:
(11)    input: < action_i >?
(12)    effect:
(13)      ...
(14)    output: < action_i >!
(15)    precondition: < side_cond >
(16)    effect:
(17)      ...
(18)    internal: < action_i >
(19)    precondition: < side_cond >
(20)    effect:
(21)      ...
(22)    markovian: < rate_i >
(23)    precondition: < pre_cond >
(24)    effect:
(25)      ...
(26)  hiding
(27)    < action_1 >! ... < action_n >!

```

Fig. 4. Principle building blocks of MIOA programmes

In the remaining lines, the transitions of the MIOA programme and thus the behaviour of its underlying IO-IMC is described. The transitions are given as pairs of action-precondition-effect. In case of input actions the preconditions are empty, i.e., simply equals to `true` (cf. lines (12) and (17)). In lines (12) to (16) the transitions that bear the action labelling `DF?` are described. Depending on the actual state the state changes that can occur are described: If the system is in state `UP` (`UP = true`), then the state values are changed as follows:

```
UP := false ; i1 := true
```

In case the system is in state `i2`, then the the state value of `i2` is set to `false` and that of `i1` is set to `true` respectively. In lines (21) to (24) the transitions labelled with output action `failed!` are described. This transition can only be taken if the precondition (line (22)) evaluates to `true`, i.e., if the state `i1` is `true`. From line (29) on, the Markovian behaviour is described. Also here, a precondition must be satisfied, before this transition can actually be taken (line (30)). In line (33) the output action `up!` is hidden from the environment.

Additionally, not shown in this example, like in I/O-IMCs, it is possible to combine two or more MIOA programmes in parallel, this is done by providing a MIOA programme, which consists of the name of the new programme, the signature, i.e., the set of the visible actions of the constituent MIOA programmes, and the keyword `parallel` followed by the names of MIOA programmes that are to be combined.

```

(1) IOIMC: IOIMC2MIOA
(2)   signature:
(3)     input: DF?, repaired?
(4)     output: up!, failed!
(5)     markovian: λ
(6)   variables:
(7)     UP: Bool := true
(8)     DOWN: Bool := false
(9)     i1 : Bool := false
(10)    i2 : Bool := false
(11)  transitions:
(12)    input: DF?
(13)    effect:
(14)      if UP = true
(15)        UP := false ; i1 := true
(16)      else if i2 = true
(17)        i2 := false ; i1 := true
(17)    input: repaired?
(18)    effect:
(19)      if DOWN = true
(20)        DOWN := false ; i2 := true
(21)    output: failed!
(22)    precondition: i1 = true
(23)    effect:
(24)      i1 := false ; DOWN := true
(25)    output: up!
(26)    precondition: i2 = true
(27)    effect:
(28)      i2 := false ; UP := true
(29)    markovian: λ
(30)    precondition: UP = true
(31)    effect:
(32)      UP := false ; i1 := true
(33)    hide up!

```

Fig. 5. MIOA specification

3.2 Formal Syntax of MIOA

On top-level, a MIOA-specification has the following grammatical definition:

Definition 5 (Fundamental Grammar of a MIOA-specification). *A MIOA-specification consists of two fundamental parts: First, a datatype-specification $Datatype_Spec$, and second an ioimc-specification $IOIMC_Spec$. In the datatype specification, all datatypes that are used in the I/O-IMC-specification. The I/O-IMC-specification contains the behavioural description of the modelled system.*

$$MIOA_Spec := Datatype_Spec \quad IOIMC_Spec$$

In turn, the I/O-IMC-specification has the following grammatical definition:

$$IOIMC_Spec := 'ioimc' \quad IOIMC_Name \quad IOIMC_Behaviour \quad Composition$$

for $Datatype_Spec$ we have:

$$Datatype_Spec := 'datatype' \quad Datatype_Name \quad axioms$$

$IOIMC_Name$, $Datatype_Name$ are simple strings that can contain any character or numeral. We do not define this further.

We will continue with the syntactical definition of the MIOA composition operator. MIOA provides the same composition operator as I/O-IMCs do, i.e., parallel composition. Further, it is possible to hide actions.

Definition 6 (Syntax of MIOA parallel/hiding operator). *In MIOA more complex I/O-IMCs can be composed by using the parallel composition operator “||”; actions can be hidden from the outside by the hiding operator hide. These two operators have the following syntactical definition:*

$$\text{Composition} := \text{'parallel'} \text{ IOIMC_Name } (\text{IOIMC_Name})^+ \mid \\ \text{'hide'} \text{ actionName_List 'in'} \text{ IOIMC_Name}$$

where *actionName_List* is defined as follows:

$$\text{actionName_List} := (\text{actionName})^+ \\ \text{actionName} := \text{internal_action} \mid \text{output_action} \mid \text{input_action} \mid \text{markovian_action} \\ \text{internal_action} := \text{action_Name}' \\ \text{output_action} := \text{action_Name}'! \\ \text{input_action} := \text{action_Name}'? \\ \text{markovian_action} := \text{action_Name}$$

where *action_Name* is again an arbitrary string.

IOIMC_Behaviour defines (1) the variables that occur in the MIOA programme at hand, (2) the actions defined in it, i.e., its signature, and finally, (3) the behavioural body of the MIOA-specification. This behavioural description describes for every action of the signature section the effect on the MIOA variables the execution of this action has on the value of the variables, depending on the actual value of variables before that action is executed.

Definition 7 (Behavioural body of MIOA specifications). *The body of a MIOA specification has the following grammar:*

$$\text{IOIMC_Behaviour} := \text{'signature'} \text{ actionName_List} \\ \text{'variables'} \text{ variable_List} \text{'transitions'} \text{ transition_List}$$

where *actionName_List* is defined as before, and *variable_List* is defined as follows:

$$\text{variable_List} := \text{variable_Name}' :! \text{ dataType}' :=! \text{ init_Value} \\ \text{dataType} := \text{dataType_Name} \mid \text{dataType_Name}' [' \text{parameter}' :! \text{ dataType} ']$$

dataType_Name and *variable_Name* are again arbitrary strings.

transition_List consists of at least one transition, that describes the effect of executing an action on the current variable values. Depending on the type of action, the transitions can be taken unconditionally (input-actions, as I/O-IMCs are input-enabled) or a precondition has to be satisfied to take the transition (all other action-types).

Definition 8 (Syntax of MIOA transitions). MIOA transitions can syntactically be defined by the following grammar:

$$\begin{aligned}
\text{transition_List} &:= \text{markovian_Transition} \mid \text{input_Transition} \mid \text{output_Transition} \mid \\
&\quad \text{internal_Transition} \\
\text{markovian_Transition} &:= \text{'Markovian' action_Name 'precondition' condition} \\
&\quad \text{'effect' transition_Effect} \\
\text{output_Transition} &:= \text{'Output' action_Name 'precondition' condition} \\
&\quad \text{'effect' transition_Effect} \\
\text{internal_Transition} &:= \text{'Internal' action_Name 'precondition' condition} \\
&\quad \text{transition_Effect} \\
\text{input_Transition} &:= \text{'Input' action_Name 'effect' transition_Effect}
\end{aligned}$$

Condition defines the conditions under which a transition can be taken. These conditions are arbitrary Boolean (propositional logic) formulae.

Definition 9 (Syntax of MIOA transition preconditions). The preconditions under which a MIOA transition can be taken are defined by the following grammar:

$$\begin{aligned}
\text{Condition} &:= \text{Boolean} \mid \text{'!' Condition} \mid \text{Condition '!' Condition} \mid \\
&\quad \text{Condition '&' Condition} \mid \text{'(' Condition ')'} \mid \text{variable_Instance ' = ' value} \\
\text{variable_Instance} &:= \text{variable_Name} \mid \\
&\quad \text{variable_Name ' (' (parameter_Instance)^+ ')'} \\
\text{parameter_Instance} &:= \text{value}
\end{aligned}$$

where *value* is the actual value of the variable instance (of course, *value* has to match the variable's data type).

transition_Effect describes the change of the values of variables that are effected by taking a specific transition.

Definition 10 (Syntax of effect of MIOA transitions).

$$\begin{aligned}
\text{Effect} &:= \text{variable_Instance ' := ' value ';' } \mid \mid \text{variable_Instance ' := ' op ' (' (variable_instance)^+ ')'} \\
&\quad \text{'if' Condition 'then' Effect 'else' Effect} \\
&\quad \text{'for' Condition 'do' Effect}
\end{aligned}$$

where *op* is any function/operation defined on the datatype of the variable at hand.

Now, we will briefly define the syntax of abstract datatype (ADT) definitions.

Definition 11.

$$\begin{aligned}
\text{datatype_Definition} &:= \text{'datatype' datatype_Name datatype_Body} \\
\text{datatype_Body} &:= \text{Functions Axioms Side_conditions} \\
\text{Functions} &:= \text{'functions' operator_Name ' : ' datatype_Name ' \to ' datatype_Name |} \\
&\quad \text{datatype_Name ' \times ' datatypes} \\
\text{datatypes} &:= \text{datatype_Name | datatype_Name ' \times ' datatypes} \\
\text{Axioms} &:= \text{'Axioms' (Axiom)^+} \\
\text{Axiom} &:= \text{operator_Name ' ((Operand)^+) ' = ' Result} \\
\text{Side_conditions} &:= \text{'sideconditions' (Axiom)^+}
\end{aligned}$$

Operand and Result can be any arbitrary string.

3.3 Semantics of MIOA

The semantics of MIOA consists of two parts:

1. The semantics of the used data types and their operations.
2. The semantics of the MIOA programmes.

The data type semantics can be defined in the usual mathematical axiomatic style. We will give two examples for this.

Abstract Data Types We restrict ourselves to give the semantics of a few regularly used abstract data types. For natural numbers and reals we can assume the usual axioms of Peano arithmetic.

Definition 12 (ADT Queue). *The data type “Queue” and its corresponding operators can be defined in the mathematic-axiomatic style as follows:*

$$\begin{aligned}
\underline{\text{Type}} &: \text{Queue}[IN] \\
\underline{\text{use}} &: \mathbb{B}, IN \\
\underline{\text{Functions}} &: \\
&\quad \text{mt_queue} \rightarrow \text{Queue}[IN] \\
&\quad \text{insert} : IN \times \text{Queue}[IN] \rightarrow \text{Queue}[IN] \\
&\quad \text{head} : \text{Queue}[IN] \rightarrow IN \\
&\quad \text{remove} : \text{Queue}[IN] \rightarrow \text{Queue}[IN] \\
&\quad \text{is_empty} : \text{Queue}[IN] \rightarrow \mathbb{B} \\
\underline{\text{Axioms}} &: \forall x, y \in IN, s \in \text{Queue}[IN] \\
&\quad \text{is_empty}(\text{mt_queue}) = \text{true} \\
&\quad \text{is_empty}(\text{insert}(s, x)) = \text{false} \\
&\quad \text{head}(\text{insert}(\text{mt_queue}, x)) = x \\
&\quad \text{head}(\text{insert}(\text{insert}(s, x), y)) = \text{head}(\text{insert}(s, x)) \\
&\quad \text{remove}(\text{insert}(\text{mt_queue}, x)) = \text{mt_queue} \\
&\quad \text{remove}(\text{insert}(\text{insert}(s, x), y)) = \text{insert}(\text{remove}(\text{insert}(s, x)), y) \\
\underline{\text{Side conditions}} &:
\end{aligned}$$

$$\begin{aligned} \text{head}(\text{mt_queue}) &= \perp \\ \text{remove}(\text{mt_queue}) &= \perp \end{aligned}$$

where mt_queue denotes the empty queue.

Definition 13 (ADT Array). *The data type “Array” and its corresponding operators can be defined in the mathematic-axiomatic style as follows:*

$$\begin{aligned} \underline{\text{Type}} &: \text{Array}[\text{Type}] \\ \underline{\text{use}} &: \mathbb{B}, \mathbb{N}, \text{Type} \\ \underline{\text{Functions}} &: \\ &\text{create} : \mathbb{N} \times \mathbb{N} \rightarrow \text{Array}[\text{Type}] \\ &\text{put} : \text{Array}[\text{Type}] \times \mathbb{N} \times \text{Type} \mapsto \text{Array}[\text{Type}] \\ &\text{lower} : \text{Array}[\text{Type}] \rightarrow \mathbb{N} \\ &\text{upper} : \text{Array}[\text{Type}] \rightarrow \mathbb{N} \\ &\text{get} : \text{Array}[\text{Type}] \times \mathbb{N} \mapsto \text{Type} \\ &\text{is_empty} : \text{Array}[\text{Type}] \rightarrow \mathbb{B} \\ \underline{\text{Axioms}} &: \forall i, j, k \in \mathbb{N}, a \in \text{Array}[\text{Type}], x \in \text{Type} \\ &\text{lower}(\text{create}(i, j)) = i \\ &\text{lower}(\text{put}(a, i, x)) = \text{lower}(a) \\ &\text{upper}(\text{create}(i, j)) = j \\ &\text{upper}(\text{put}(a, i, x)) = \text{upper}(a) \\ &k = i \rightarrow \text{get}(\text{put}(a, i, x), k) = x \\ &k \neq i \rightarrow \text{get}(\text{put}(a, i, x), k) = \text{get}(a, k) \\ &\text{is_empty}(\text{create}(i, j)) = \text{true} \\ &\text{is_empty}(\text{put}(a, i, x)) = \text{false} \\ \underline{\text{Side conditions}} &: \\ &\forall i \in \mathbb{N}, a \in \text{Array}[\text{Type}], x \in \text{Type} \\ &\text{put}(a, i, x) : \text{lower}(a) \leq i \leq \text{upper}(a) \\ &\text{get}(a, i, x) : \text{lower}(a) \leq i \leq \text{upper}(a) \end{aligned}$$

In MIOA, a (programme) state is interpreted as an ordered tuple ν of arity n , n is the number of variables of the MIOA programme. The value of the entries represent the current values of the variables. We can assume an ordering of variables of the programme. Variable i has position i in ν . A state change is then a change in at least one of the variables contained in the state descriptor. A similar concept of state is known in Petri net theory, where states are represented as tuples, where each entry represents a place and the current number of tokens, it contains.

Definition 14 (SO Semantics of MIOA). *The semantics of MIOA can formally be defined in the SO style of [15].*

1. $\frac{}{\langle \text{skip}, \nu \rangle \rightarrow \langle E, \nu \rangle}$:
If nothing (*skip*) is to be done, and ν is a state of the programme, we execute the empty programme E and leave the state ν unaltered.
2. $\frac{}{\langle \text{skip}; S, \nu \rangle \rightarrow \langle S, \nu \rangle}$:

- skip is the neutral element of programme composition.*
3.
$$\frac{}{\langle u := t, \nu \rangle \xrightarrow{a!} \langle E, \nu[u := t] \rangle} :$$

In state ν , the variable u is assigned the value t , and input action $a!$ can be consumed.
 4.
$$\frac{\nu \models \Phi}{\langle u := t, \nu \rangle \xrightarrow{a?} \langle E, \nu[u := t] \rangle} :$$

If in state ν the precondition Φ is satisfied, the variable u is assigned the value t , and output action $a?$ can be consumed.
 5.
$$\frac{\nu \models \Phi}{\langle u := t, \nu \rangle \xrightarrow{\lambda} \langle E, \nu[u := t] \rangle} :$$

If in state ν the precondition Φ is satisfied, the variable u is assigned the value t , and after an exponentially distributed delay, governed by rate λ , the state is changed to $\nu[u := t]$ and the empty programme E remains to be executed.
 6.
$$\frac{\langle S_1, \nu \rangle \xrightarrow{a} \langle S_2, \tau \rangle}{\langle S_1; S, \nu \rangle \xrightarrow{a} \langle S_2; S, \tau \rangle} :$$

If a programme S_1 in state ν (input/output or Markovian transition) can evolve into programme S_2 (state τ), then it is also possible to compose S_1 sequentially with programme S and still execute the same transitions evolving into programme $S_2; S$ in state τ .
 7.
$$\frac{\nu \models \Phi}{\langle \mathbf{if} \Phi \mathbf{then} S_1 \mathbf{else} S_2, \nu \rangle \rightarrow \langle S_1, \nu \rangle} :$$

*If in the current state ν the guard of the **if**-clause is satisfied, we execute programme S_1 . The choice of the **then**-branch does not alter the current state ν of the MIOA programme.*
 8.
$$\frac{\nu \not\models \Phi}{\langle \mathbf{if} \Phi \mathbf{then} S_1 \mathbf{else} S_2, \nu \rangle \rightarrow \langle S_2, \nu \rangle} :$$

*If in the current state ν the guard of the **if**-clause is not satisfied, we execute programme S_2 . The choice of the **else**-branch does not alter the current state ν of the MIOA programme.*
 9.
$$\frac{\nu \models \Phi}{\langle \mathbf{while} \Phi \mathbf{do} S, \nu \rangle \rightarrow \langle S; \mathbf{while} \Phi \mathbf{do} S, \nu \rangle} :$$

*If in the current state ν the guard of the **while**-loop is satisfied, we execute programme S , after executing S , we can execute **while** Φ **do** S again.*
 10.
$$\frac{\nu \not\models \Phi}{\langle \mathbf{while} \Phi \mathbf{do} S, \nu \rangle \rightarrow \langle E, \nu \rangle} :$$

*If in the current state ν the guard of the **while**-loop is not satisfied, we can not execute programme S , we end up with the empty programme E in state ν .*

4 A Logic for MIOA

In this section we introduce the logic intSPDL (interactive SPDL), a stochastic logic that like SPDL [13] or asCSL [1] allows to reason about the behaviour of MIOA specifications on the level of action sequences. That means the desired system behaviour is mainly described by means of regular programmes.

For intSPDL we have to take into consideration that the Markovian behaviour is separated from the interactive behaviour. That means, actions are always untimed. This fact has to be reflected in the semantic definition of path formulae. Finally, this leads to a semantics for intSPDL, which is similar to that of IMSPDL [13].

The semantic model of intSPDL are I/O-IMCs, as defined in Section 2.1.

4.1 Syntax of intSPDL

The logic intSPDL is a stochastic extension of the logic PDL [6], a multi-modal programme logic. Beside the standard ingredients such as propositional logic and the modal \diamond -operator (“possibly”), PDL enriches the \diamond -operator with so-called regular programmes which are regular expressions of actions and tests (cf. Def. 16 below). If Φ and Ψ are PDL formulae and ρ is a programme, then $\Phi \vee \Psi$, $\neg\Phi$ and $\langle \rho \rangle \Psi$ are formulae. $\langle \rho \rangle \Psi$ means that it is possible to execute programme ρ , thereby ending up in a state that satisfies Ψ .

With respect to PDL we have added the following operators to obtain intSPDL: A path operator that extends the original PDL $\langle \cdot \rangle$ -operator by specifying time bounds within which the Ψ state has to be reached, a probabilistic path quantifier $\mathcal{P}_{\bowtie p}$ to reason about the transient probabilistic behaviour of a system, and a steady-state operator $\mathcal{S}_{\bowtie p}$ to reason about the behaviour of the system once stationarity of the underlying Markov chain is reached. The formulae of intSPDL are formally defined as follows:

Definition 15. (Syntax of intSPDL) Let $p \in [0, 1]$ be a probability and $q \in \text{AP}$ an atomic proposition and $\bowtie \in \{\leq, <, \geq, >\}$ a comparison operator. The state formulae Φ of SPDL are defined as:

$$\Phi := q \mid \Phi \vee \Phi \mid \neg\Phi \mid \mathcal{S}_{\bowtie p}(\Phi) \mid \mathcal{P}_{\bowtie p}(\phi) \mid (\Phi)$$

Path formulae ϕ are defined by

$$\phi := \Phi[\rho]^I \Phi,$$

where I is the closed time interval $[t, t']$ of the real axis. The symbol ρ represents a programme as defined by Def. 16.

Definition 16. (Programmes) Let \mathcal{I} be an I/O-IMC, $\text{Act}(\mathcal{I})$ be the set of actions (also called atomic programmes) defined over \mathcal{I} and TEST be a set of intSPDL state formulae. A programme ρ is defined by the following grammar:

$$\rho := \epsilon \mid \Phi?; a \mid \rho; \rho \mid \rho \cup \rho \mid \rho^* \mid \Phi?; \rho \mid (\rho)$$

where $\epsilon \notin \text{Act}$ is the empty programme, $a \in \text{Act}$ and $\Phi \in \text{TEST}$.

The operators $;$ (sequential composition), \cup (choice), and $*$ (Kleene star) have their usual meaning. The operator $\Phi?; \rho$ (resp. $\Phi?; a$) is the so-called test operator (also called guard operator). Its informal semantics is as follows: test whether

Φ holds in the current state of the model. If this is the case, then execute programme ρ , otherwise ρ is not executable. Def. 16 requires that every atomic programme is preceded by a test formula Φ , but this can be the trivial test (i.e., $\Phi = \text{true}$). From automata theory it is known that regular expressions coincide with regular languages, i.e., sets of words that are generated according to the rules of regular expressions. Programmes as defined in Def. 16 can be seen as regular expressions over the alphabet $\Sigma = \text{TEST} \times (\text{Act} \cup \epsilon)$. Words that are generated from programmes in intSPDL will be referred to as *programme instances*. The set of these programme instances is called, as before, a language. For a given programme ρ , $\mathcal{L}(\rho)$ is the language, induced by ρ . The length of a programme instance r , denoted by $|r|$, is the number of elements from Σ occurring in it. For $0 \leq i < |r|$, $r[i]$ is the $(i + 1)$ st element of r . $TF(r[i])$ denotes the test formula part of $r[i]$, and $\text{Act}(r[i])$ denotes the action part of $r[i]$.

4.2 Semantics of intSPDL

Informally, the semantics of intSPDL formulae can be described as follows:

- The meaning of negation ($\neg\Phi$) and disjunction ($\Phi \vee \Psi$) is as usual.
- $\mathcal{S}_{\bowtie p}(\Phi)$ asserts that the steady-state probability of the set of Φ -states, i.e. the probability to reside in a Φ -state once the system has reached stationarity, satisfies the bound as given by $\bowtie p$.
- $\mathcal{P}_{\bowtie p}(\phi)$ asserts that the probability measure of the paths that satisfy ϕ is within the bound as given by $\bowtie p$.
- Path formula $\Phi[\rho]^{[t,t']}\Psi$ means that a state that satisfies Ψ is reached within at least t but at most t' time units, and that all preceding states must satisfy Φ . Additionally, the action sequence of the path to the Ψ state must correspond to the action sequence of a word from the language \mathcal{L}_ρ (the language induced by programme ρ) and all test formulae that are part of programme ρ must be satisfied by the corresponding states on the path.

Definition 17. (State probabilities) *The probability to be in state s' at time point t , provided that the system is in state s at time 0, is given by*

$$\pi^{\mathcal{I}}(s, s', t) = Pr(\sigma \in \text{PATH}^{\mathcal{I}}(s) \mid \sigma @ t = s')$$

The definition for steady-state probabilities is similar, taking into account that steady-state means 'on the long run':

$$\pi^{\mathcal{I}}(s, s') = \lim_{t \rightarrow \infty} \pi^{\mathcal{I}}(s, s', t)$$

These definitions can be extended to sets of states: For $S' \subseteq S$:

$$\pi^{\mathcal{I}}(s, S', t) := \sum_{s' \in S'} \pi^{\mathcal{I}}(s, s', t) \quad \text{and} \quad \pi^{\mathcal{I}}(s, S') := \sum_{s' \in S'} \pi^{\mathcal{I}}(s, s').$$

We are now ready to give the formal semantics of intSPDL:

Definition 18 (Semantics of intSPDL). *The semantics of state formulae is defined as follows:*

$$\begin{aligned}
\mathcal{I}, s \models q &\iff q \in L(s) \\
\mathcal{I}, s \models \neg\Phi &\iff \mathcal{I}, s \not\models \Phi \\
\mathcal{I}, s \models (\Phi \vee \Psi) &\iff \mathcal{I}, s \models \Phi \text{ or } \mathcal{I}, s \models \Psi \\
\mathcal{I}, s \models \mathcal{S}_{\bowtie p}(\Phi) &\iff \pi^{\mathcal{I}}(s, \text{Sat}(\Phi)) \bowtie p \\
\mathcal{I}, s \models \mathcal{P}_{\bowtie p}(\phi) &\iff \text{Prob}^{\mathcal{I}}(s, \phi) \bowtie p
\end{aligned}$$

$\text{Sat}(\Phi)$ is the set of states that satisfy Φ , and $\text{Prob}^{\mathcal{I}}(s, \phi)$ is the probability measure of all paths $\sigma \in \text{PATH}(s)$ that satisfy ϕ :

$$\text{Prob}^{\mathcal{I}}(s, \phi) := \text{Pr}(\sigma \in \text{PATH}^{\mathcal{I}}(s) \mid \mathcal{I}, \sigma \models \phi)$$

For the semantics of path formulae we have to relate the instances of the programme ρ to words on paths in the ESLTS \mathcal{I} .

Definition 19 (Words on paths). *The word \mathcal{W}^k of length $k \geq 0$ on a path $\sigma_{\text{Act}} \in \text{PATH}^{\mathcal{I}}$ is defined as follows:*

$$\begin{aligned}
\mathcal{W}^0(\sigma_{\text{Act}}) &:= \epsilon, \\
\mathcal{W}^k(\sigma_{\text{Act}}) &:= \mathcal{W}^{k-1}(\sigma_{\text{Act}}) \circ a[k-1], \\
\text{where } a[k-1] &\in \text{Act} \wedge \sigma[k-1] \xrightarrow{a[k-1]} \sigma[k].
\end{aligned}$$

For $i = 0, 1, \dots, k-1$, $\mathcal{W}^k(\sigma_{\text{Act}})[i]$ denotes the $(i+1)$ st action on path σ_{Act} .

Definition 20 (Semantics of path formulae). *The semantics of path formulae is defined as follows:*

$$\begin{aligned}
\mathcal{I}, \sigma \models \Phi[\rho]^{[t, t']}\Psi &\iff \exists k (\mathcal{I}, \sigma[k] \models \Psi \wedge \forall 0 \leq i < k (\mathcal{I}, \sigma[i] \models \Phi) \\
&\quad \wedge \text{time_restriction} \\
&\quad \wedge \text{programme_matching})
\end{aligned}$$

The first line states that there must be a state $\sigma[k]$ that satisfies Ψ and that all preceding states must satisfy Φ . The formula “time_restriction” is defined as follows:

$$\begin{aligned}
(1) \quad &((t = 0 \wedge \sum_{i=0}^{k-1} t_i \leq t') \vee \\
(2) \quad &(t \neq 0 \wedge ((t \leq \sum_{i=0}^{k-1} t_i \leq t') \vee (\sum_{i=0}^{k-1} t_i < t \wedge \sum_{i=0}^k t_i > t \wedge \sigma[k] \models \Phi)))
\end{aligned}$$

It expresses the restrictions stemming from the time bounds that are imposed on paths. In line (1), if the lower time bound is zero, then the only requirement is to reach a Ψ -state before more than t' time units have passed. Line (2) covers the case where the lower time bound is greater than zero. In this case, either

the entry time into state $\sigma[k]$ must lie within the interval $[t, t']$, or if the entry time is less than t , then the sojourn time in $\sigma[k]$ plus the sojourn times in the previous states must be greater than t .

The formula “programme_matching” is defined as follows:

- (1) $(\exists r \in \mathcal{L}(\rho) \wedge |r| = l \wedge \text{Act}(r[l-1]) \neq \epsilon \wedge \forall 0 \leq i \leq l-1 (\text{Act}(r[i]) = \mathcal{W}^l(\sigma_{\text{Act}})[i] \wedge \mathcal{I}, \sigma_{\text{Act}}[i] \models \text{TF}(r[i])))$
- (2) $(\exists r \in \mathcal{L}(\rho) \wedge |r| = l \wedge \text{Act}(r[l-1]) = \epsilon \wedge \mathcal{I}, \sigma_{\text{Act}}[l] \models \text{TF}(r[l]) \wedge \forall 0 \leq i \leq l-1 (\text{Act}(r[i]) = \mathcal{W}^l(\sigma_{\text{Act}})[i] \wedge \mathcal{I}, \sigma_{\text{Act}}[i] \models \text{TF}(r[i])))$

where σ_{Act} is the portion of the original path σ , restricted to the transitions that are labelled with actions, i.e. transition from “ \rightarrow ” and $l = |\sigma_{\text{Act}}|$ is the length of σ_{Act} . This formula expresses that the word induced on path σ must be matched by the corresponding action parts of a program instance r and that the tests appearing in the program must be satisfied by the appropriate states on the path. There are two possibilities, as indicated in the formula: (1) If the last element of r is of the form $\Phi?; a$, where $a \neq \epsilon$, the corresponding state must satisfy the test formula and the last transition on the path must have a label identical to the action part of $r[k-1]$. (2) If the last element of r is of the form $\Phi?; \epsilon$, i.e. has an empty action part, then it only has to be checked whether the corresponding state on the path satisfies the test formula.

5 Model Checking intSPDL

In this section, we describe the model checking algorithm for the logic intSPDL. Central for this are the notions of programme automata and product I/O-IMCs, which we introduce in the sequel.

5.1 Basic Model Checking Algorithm for intSPDL

The basic idea of model checking intSPDL is borrowed from CTL, in the sense that the model checking starts with atomic properties, and then proceeds to ever more complex subformulae until the entire formula has been checked. This core algorithm can be found in Fig. 6. We will present the basic ideas of model checking formulae of the type $\mathcal{S}_{\triangleright p}(\Phi)$ and $\mathcal{P}_{\triangleright p}(\phi)$, as for the rest, the CTL approach can be applied.

5.2 Model Checking Steady State Formulae $\mathcal{S}_{\triangleright p}(\Phi)$:

In order to compute the satisfiability set of steady state formulae $\mathcal{S}_{\triangleright p}(\Phi)$, we need the following definition.

Definition 21 (State-labelled CTMC). A state-labelled CTMC (SMC) is a quadruple $\mathcal{M} := (S, L, R, s)$, where:

```

(0)  $Sat(\Phi)$  {
(1)   switch( $\Phi$ )
(2)     case:  $\Phi = true$ : return  $S$ 
(3)     case:  $\Phi = q$ : return  $\{s \in S \mid q \in L(s)\}$ 
(4)     case:  $\Phi = \neg\Psi$ : return  $S \setminus Sat(\Psi)$ 
(5)     case:  $\Phi = \Psi \vee \Xi$ : return  $Sat(\Psi) \cup Sat(\Xi)$ 
(6)     case:  $\Phi = \mathcal{S}_{\bowtie p}(\Psi)$ : return  $Sat(\mathcal{S}_{\bowtie p}(\Psi))$ 
(7)     case:  $\Phi = \mathcal{P}_{\bowtie p}(X^I\Psi)$ : return  $Sat(\mathcal{P}_{\bowtie p}(X^I\Psi))$ 
(8)     case:  $\Phi = \mathcal{P}_{\bowtie p}(\Psi U^I \Xi)$ : return  $Sat(\mathcal{P}_{\bowtie p}(\Psi U^I \Xi))$ 
(9) }

```

Fig. 6. Model Checking Algorithm for intSPDL

- S is a finite set of states.
- $L : S \mapsto 2^{AP}$ is the state labelling function that associates with every state $s \in S$ the set of atomic propositions which hold in that state. AP is the set of atomic propositions.
- $R : S \times \mathbb{R} \times S$ is the Markovian transition relation. Act_M is a finite set of Markovian action labels, i.e. actions, that are associated with Markovian transitions.
- $s \in S$ is the unique initial state of \mathcal{M} .

Having this, we can proceed as follows to actually compute $Sat(\mathcal{S}_{\bowtie p}(\Psi))$

1. Compute the satisfiability set of Ψ .
2. The I/O-IMC \mathcal{I} is transformed into a state-labelled Markov chains (SMC) \mathcal{M} .
3. On \mathcal{M} model checking $\mathcal{S}_{\bowtie p}(\Psi)$ is identical to the corresponding CSL case [2].
4. A state with only outgoing interactive transitions satisfies $\mathcal{S}_{\bowtie p}(\Psi)$ if a state with only outgoing Markovian transitions that satisfies $\mathcal{S}_{\bowtie p}(\Psi)$ is reachable from it.

5.3 Model Checking Probabilistic Path Formulae $\mathcal{P}_{\bowtie p}(\phi)$:

The procedure for the computation of the satisfiability set of probabilistic path formulae $\mathcal{P}_{\bowtie p}(\phi)$, where $\phi = \Phi[\rho]^I\Psi$ is much more involved and proceeds along the following lines:

- We assume, we want to check whether in an I/O-IMC \mathcal{I} a state s satisfies $\mathcal{P}_{\bowtie p}(\phi)$, with $\phi = \Phi[\rho]^I\Psi$. The basic idea is to reduce the model checking problem of intSPDL to one of CSL, which consists of deciding whether a continuous time Markov chain (CTMC) \mathcal{M}^\times (to be constructed) and a state s^\times in \mathcal{M}^\times satisfies the CSL formula $\mathcal{P}_{\bowtie p}(F^I succ)$. A path satisfies $F^I succ$, if within time interval I a state is reached that satisfies the atomic property $succ$. To reach this goal, we proceed as follows:

1. From the programme ρ we derive a deterministic programme automaton A_ρ , which is a variant of deterministic finite automata.²
2. Using the given I/O-IMC \mathcal{I} and the programme automaton A_ρ we build a product I/O-IMC (PIOIMC) \mathcal{I}^\times . The state space of \mathcal{I}^\times is the product of \mathcal{I} and A_ρ , i.e., its states are of the form (s_i, z_i) , where s_i is a state of \mathcal{I} and z_i a state of A_ρ . Additionally, \mathcal{I}^\times possesses two new, absorbing states: the error state *FAIL*, and the success state *SUCC*.

In \mathcal{I}^\times a transition $(s_i, z_i) \xrightarrow{\lambda} (s_j, z_j)$ is kept, where λ is the rate of the transition from s_i to s_j , iff the following two constraints are satisfied:

- (s_i, z_i) must satisfy Φ , this is the case iff s_i satisfies Φ .
- Both s_i and z_i must be capable to perform the same action, and if the current action is associated with a test, then s_i must also satisfy this test.

If one of these two constraints is violated, we have to introduce a transition $(s_i, z_i) \xrightarrow{\lambda} \text{FAIL}$ and delete transition $(s_i, z_i) \xrightarrow{\lambda} (s_j, z_j)$.

3. Finally, to compute the probability measure of the paths that satisfy ϕ we proceed as follows. All states (s_j, z_j) of \mathcal{I}^\times for which s_j is a Ψ -state and z_j is an accepting state of A_ρ are replaced by the newly introduced absorbing success state *SUCC*, labelled with the special, newly introduced atomic state formula *succ*, thereby redirecting all incoming transitions from the old states to the new *SUCC* state.
4. At this point, it is possible to check, whether $\mathcal{P}_{\triangleright\triangleright p}(\Phi[\rho]^{[t,t']}\Psi)$ is functionally satisfiable: If in \mathcal{I}^\times a path to a *succ* state exists, then $\mathcal{P}_{\triangleright\triangleright p}(\Phi[\rho]^{[t,t']}\Psi)$ can be satisfied at least on the functional level.
5. If the original state s_i is a Markovian state, we can compute \mathcal{I}^\times (which was transformed as described in step 3) we can compute the probability measure of all paths satisfying the CSL formula $\mathcal{P}_{\triangleright\triangleright p}(\mathbf{F}^{[t,t']}\text{succ})$, which is equal to the probability measure of the paths satisfying the original formula $\mathcal{P}_{\triangleright\triangleright p}(\Phi[\rho]^{[t,t']}\Psi)$ in the original model \mathcal{I} .
6. If the state, for which the path formula is checked, is an interactive state, we have the probability to reach the *SUCC* state is the probability of paths that emanate from Markovian states that are reachable from this interactive state.

Definition 22 (Product I/O-IMC (PIOIMC)). *Let an I/O-IMC \mathcal{I} and a DPA A_ρ be given. The PIOIMC $\mathcal{I}^\times = (S^\times, \rightarrow, \dashrightarrow, L^\times, s^\times)$ is defined as follows:*

- *initial states:* $S_{Start}^\times := \{(s_i, z_\rho^{Start}) \mid s_i \in S\}$
- *accepting states:* $S_{Acc}^\times := \{(s_i, z_\rho^j) \in S^\times \mid s_i \in \text{Sat}(\Psi) \wedge z_\rho^j \in E_\rho\}$
- *labelling:*
 1. $\forall (s_i, z_\rho^j) \in S^\times \setminus S_{Acc}^\times (L^\times(s_i, z_\rho^j) = L(s_i))$
 2. $\forall (s_i, z_\rho^j) \in S_{Acc}^\times (L^\times(s_i, z_\rho^j) = \{\text{succ}\})$

² For the derivation of A_ρ from programme ρ we refer to [12] for a thorough discussion of this issue. As such this issue does not play a crucial role in understanding this paper.

3. $L^\times(\text{FAIL}) = \{\text{fail}\}$
- S^\times is the product state space between the original I/O-IMC and the programme automaton. Additionally, S^\times contains two new states **FAIL** and **SUCC**. **FAIL** is an absorbing state, to which all transitions are directed that make a path formula functionally non-satisfiable. **SUCC** is an absorbing state, to which all transitions are directed that lead to a state that functionally satisfies the path formula that is currently to be verified. That means, all states bearing the label **succ** can be deleted and need to be included in the product state space S^\times . Formally, this can be defined as follows:

$$S^\times := \{(s_i, z_j^i) \mid s_i \in S \setminus S_{Acc}^\times \wedge z_j^i \in Z_\rho\} \cup \{\text{FAIL}\} \cup \{\text{SUCC}\}$$

- transition relation: $R^\times \subseteq (S \times Z) \times \mathbb{R}_{>0} \times (S \times Z)$ as defined in definition 23

Definition 23 (Transition relation R^\times for PIOIMC). Assume, the path formula to be checked is, $\Phi[\rho]^I \Xi$, then the transition relation R^\times for a PIOIMC \mathcal{T}^\times can inductively be defined as follows:

0. Initially, $R^\times = \emptyset$
1. $R^\times := R^\times \cup \{((s_i, z_i), \lambda, (s_j, z_i)) \mid s_i \xrightarrow{\lambda} s_j \wedge s_i \notin \text{Sat}(\Psi) \wedge s_j \in \text{Sat}(\Phi)\}$
2. $R^\times := R^\times \cup \{((s_i, z_i), a[!|?|;], (s_j, z_j)) \mid s_i \xrightarrow{a[!|?|;]} s_j \wedge s_i \in \text{Sat}(\Psi)\}$
3. $R^\times := R^\times \cup \{((s_i, z_i), a[!|?|;], \text{SUCC}) \mid s_i \xrightarrow{\Psi?; a[!|?|;]} s_j \wedge z_i \xrightarrow{\Psi?; a[!|?|;]} z_j \wedge s_j \in \text{Sat}(\Xi) \wedge s_i \in \text{Sat}(\Psi) \wedge z_j \in E(A_\rho)\}$
4. $R^\times := R^\times \cup \{((s_i, z_i), \lambda, \text{SUCC}) \mid s_i \xrightarrow{\lambda} s_j \wedge s_i \in \text{Sat}(\Psi) \wedge z_i \in E(A_\rho)\}$
5. $R^\times := R^\times \cup \{((s_i, z_i), \lambda, \text{SUCC}) \mid s_i \xrightarrow{\lambda} s_j \wedge z_i \xrightarrow{\Psi?; \epsilon z_j} \wedge s_i \in \text{Sat}(\Phi) \wedge s_j \in \text{Sat}(\Psi) \wedge z_j \in E(A_\rho)\}$
6. $R^\times := R^\times \cup \{((s_i, z_i), a[!|?|;], \text{FAIL}) \mid s_i \xrightarrow{a[!|?|;]} s_j \wedge z_i \xrightarrow{\Psi?; a[!|?|;]} z_j\}$
7. $R^\times := R^\times \cup \{((s_i, z_i), a[!|?|;], \text{FAIL}) \mid s_i \xrightarrow{a[!|?|;]} s_j \wedge z_i \xrightarrow{\Psi?; a[!|?|;]} z_j \wedge s_j \notin \text{Sat}(\Phi)\}$
8. $R^\times := R^\times \cup \{((s_i, z_i), a[!|?|;], \text{FAIL}) \mid s_i \xrightarrow{a[!|?|;]} s_j \wedge z_i \xrightarrow{\Psi?; a[!|?|;]} z_j \wedge s_j \notin \text{Sat}(\Psi)\}$
9. $R^\times := R^\times \cup \{((s_i, z_i), \lambda, \text{FAIL}) \mid s_i \xrightarrow{\lambda} s_j \wedge z_i \xrightarrow{\Psi?; \epsilon z_j} \wedge s_i \in \text{Sat}(\Phi) \wedge s_j \notin \text{Sat}(\Psi) \wedge z_j \in E(A_\rho)\}$

6 MIOA and Arcade

6.1 Arcade Modelling Approach and MIOA

The basic idea behind **Arcade** is that it defines a system as a set of interacting components, where each component is provided with a set of operational/failure modes, time-to-failure/repair distributions, and failure/repair dependencies. We propose a predefined set of components along with an extensible set of features (such as interactions, dependencies, operational/failure modes, etc).

We have identified three main components with which we can, in a modular fashion, construct a system model: (1) a Basic Component (BC), (2) a Repair Unit (RU), and (3) a Spare Management Unit (SMU). The underlying semantics of each of these components are I/O-IMCs resp. MIOA programmes.

A basic component represents a physical/logical system component that has a distinct operational and failure behavior. A BC can have any number of operational modes (e.g., *active vs. inactive*, *normal vs. degraded*) and can fail either due to an inherent failure (realized as a Markovian transition) or due to a *destructive functional dependency*.

The RU component handles the repair of one or many BCs. Various *repair policies* (e.g., first-come-first-served, priority) and repair dependencies between BCs can be implemented. Finally, the SMU handles the activation and deactivation of BCs used as spare components.

Example 2. Assume, we have some RAID system, consisting of 10 hard disks, which are controlled by a single disk controller. The disks and the disk controller are subject to failures. The ten disks share a single repair unit, and are repaired according to a first-come-first-served (FCFS) policy. The disk controller has its own repair unit. The basic component *Arcade* models of the disk controller and the disks as well as the *Arcade* model of the disks' repair unit can be found in Fig. 7. The semantic model of the disks and the disk controller is essentially

| | |
|--|---|
| <p>COMPONENT: <i>disk_controller</i> TIME-TO-FAILURE: $\exp(\frac{1}{2000})$ TIME-TO-REPAIR: $\exp(1)$</p> | <p>COMPONENT: <i>disk_1</i> TIME-TO-FAILURE: $\exp(\frac{1}{8000})$ TIME-TO-REPAIR: $\exp(1)$</p> |
| <p>REPAIR UNIT: <i>disk.rep</i> COMPONENTS: <i>disk_1, \dots, disk_10</i> REPAIR STRATEGY: FCFS</p> | |

Fig. 7. Arcade models of disks and disk controller and repair unit for disks

the MIOA programme in Fig. 1. This programme only has to be instantiated (automatically!) with the correct action names and failure rates. In Fig. 8 we can find the MIOA specification for the FCFS repair strategy. The final semantic model, i.e., the model on which the dependability evaluation can be done, is obtained by composing the semantic models of disks, disk controllers and their respective repair units in parallel, using the composition operator defined for I/O-IMCs resp. MIOA.

6.2 The Arcade Approach and intSPDL

Arcade only provides fairly limited means to specify dependability measures. Typically, it is only possible to define under which conditions and with which probability the system under analysis is down or operational.

```

(1) IOIMC: Repair_FCFS
(2) signature:
(3)   input: failed(10 : Int)?
(4)   output: repaired(10 : Int)!
(5)   markovian:  $\lambda(10 : \text{Int})$ 
(6) variables:
(7)   available: Bool := true
(8)   busy: Array[10 : Bool] := false
(9)   internal: Array[10 : Bool] := false
(10)  queue: Queue[10 : Int] := empty
(11) transitions:
(12) input: failed(i)?
(13) effect:
(14)   if available = true
(15)     busy(i) := true ; available := false
(16)   else if busy(j) (or internal(j)) = true
(17)     insert(i, queue) ; busy(j) (internal(j)) := true
(18)   else if internal(j) = true
(19)     insert(i, queue) ; internal(j) := true
(20)   else if busy(i) = true
(21)     busy(i) := true
(22)   else if internal(i) = true
(23)     insert(i, queue) ; internal(i) := false ; j := head(queue) ; busy(j) := true
(24) output: repaired(i)!
(25) precondition: internal(i) = true  $\wedge$  i  $\notin$  queue
(26) effect:
(27)   if queue = empty
(28)     available := true ; internal(i) := false
(29)   else
(30)     j := head(queue) ; busy(j) := true
(31) markovian:  $\lambda(i)$ 
(32) precondition: busy(i) = true
(33) effect:
(34)   busy(i) := false ; internal(i) := true

```

Fig. 8. MIOA specification of FCFS repair strategy

By providing some predefined patterns, that can be mapped to intSPDL formulae, it is possible to express and analyse more complex quantitative measures. Up to now, we have defined the some patterns, we found useful in the context of dependability engineering, that can be mapped on intSPDL probabilistic path formulae, e.g.:

Component i fails before/after component j : Such a measure can be useful in the context of spare management, to judge the probability that a spare component (in warm or hot standby) fails after/before the primary component:

comp_ifails before comp_j

This translates to

$$\mathcal{P}_{\triangleright sp}(\text{true}[(\text{Act} \setminus \{ \text{failed}_i, \text{failed}_j \})^* ; \text{failed}_i! ; (\text{Act} \setminus \{ \text{failed}_i, \text{failed}_j \})^* \text{failed}_j!]^{[0, \infty]} \text{true})$$

If a time bound other than ∞ is required, the keyword *within* followed by $[t, t']$ can be added.

7 Conclusion

In this report, we have introduced the Markovian I/O-IMC language MIOA. MIOA is inspired by I/O-IMCs and the input/output automata language (IOA). We have defined the syntax and semantics of MIOA, a logic for MIOA, intSPDL, and their corresponding model checking algorithms. A few application examples of MIOA were given.

As a next step, we will implement a parser and state space generator, i.e. an I/O-IMC generator for MIOA specifications.

It is also planned to further develop the possibilities of mapping high-level, abstract dependability measures onto intSPDL to further increase the usability of MIOA in the context of Arcade.

References

1. Ch. Baier, L. Cloth, B. R. Haverkort, M. Kuntz, and M. Siegle. Model Checking Markov Chains with Actions and State Labels. *IEEE Transactions on Software Engineering*, 33(4):209–224, 2007.
2. Ch. Baier, B. Haverkort, H. Hermanns, and J.P. Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Trans. Software Eng.*, 29(7):1–18, July 2003.
3. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. North-Holland, Amsterdam, 1989.
4. H. Boudali, P. Crouzen, B.R. Haverkort, M. Kuntz, and M.I.A. Stoelinga. Architectural Dependability Modelling with Arcade. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, to appear*, 2008.
5. H. Boudali, P. Crouzen, and M.I.A. Stoelinga. A compositional semantics for Dynamic Fault Trees in terms of Interactive Markov Chains. In *Proc. of the 5th International Symposium on Automated Technology for Verification and Analysis*, pages 441–456. LNCS, 2007.
6. M. Fischer and R. Ladner. Propositional dynamic logic of regular programs. *J. Comput. System Sci.*, 18:194–211, 1979.
7. S. Garland, N. Lynch, J. Tauber, and M. Vaziri. IOA User Guide and Reference Manual. Technical Report Technical Report MIT-LCS-TR-961, Massachusetts Institute Technology, Cambridge, MA, 2004.
8. S. J. Garland, J. V. Guttag, and J. J. Horning. An Overview of Larch. In *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992*, pages 329–348. Springer-Verlag, 1993.
9. J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The Formal Specification Language mCRL2. In *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl, Germany, 2007.
10. J.F. Groote and A. Ponse. The syntax and semantics of μ CRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. SV, 1995.

11. H. Hermanns. *Interactive Markov Chains*, volume 2428 of *Lecture Notes in Computer Science*. Springer, 2002.
12. M. Kuntz. *Symbolic Semantics and Verification of Stochastic Process Algebras*. PhD thesis, Universität Erlangen-Nürnberg, Institut für Informatik 7, 2006.
13. M. Kuntz and M. Siegle. Symbolic Model Checking of Stochastic Systems: Theory and Implementation. In *13th International SPIN Workshop*, pages 89–107. Springer, LNCS 3925, 2006.
14. N. Lynch and M. Tuttle. An Introduction to Input/output Automata. *CWI Quarterly*, 2(3):219–246, 1989.
15. G. Plotkin. A structural approach to operational semantics. technical report, Computer Science Department FN-19, DAIMI, Aarhus University, September 1981.