

Formal performability evaluation of architectural models of critical infrastructures

B.R. Haverkort

Embedded Systems Institute, Eindhoven, The Netherlands
University of Twente, Enschede, The Netherlands

M. Kuntz

University of Konstanz, Konstanz, Germany

A. Remke & S. Roolvink

University of Twente, Enschede, The Netherlands

ABSTRACT: In this paper, we introduce MIOA, a stochastic process algebra-like specification language with finite-domain datatypes, as well as the logic intSPDL, and its model checking algorithms. MIOA which stands for Markovian input/output automata language is an extension of Lynch’s input/output automata with Markovian timed transitions. MIOA can serve both as a fully fledged “stand-alone” specification language and as the semantic model for the architectural dependability description language Arcade. The logic intSPDL is an extension of the stochastic logic SPDL, designed to deal with the specialties of MIOA. In the context of Arcade, intSPDL 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 present application examples of 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. Besides purely functional correctness, i.e. a device does what it is expected to do, correct functioning also encompasses quantitative aspects, such as performance and dependability. While functional verification using techniques such as, for instance, 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 achieved as a byproduct of the ordinary system design process since short production cycles and tight cost objectives do not allow for additional development cycle costs for system evaluation. In addition to this requirement, a dependability evaluation formalism should have a formal semantics, high expressiveness, entail low modelling effort, and be accompanied by tool support. Over the last decades numerous dependability evaluation formalisms have been devised. All of them have shortcomings with respect to one or another

of the above mentioned requirements. By proposing Arcade in (Boudali, Crouzen, Haverkort, Kuntz and Stoelinga 2008) we laid the foundations for a dependability evaluation approach that satisfies the requirements.

In this paper, we introduce MIOA, the Markovian input/output automata language. MIOA serves both as the formal semantic model of Arcade and as a powerful stand-alone specification language for general systems that exhibit stochastic behaviour. As special features, MIOA possesses abstract data types and programming language constructs including loops and control structures. These features are also useful in the context of Arcade, as they facilitate the semantic definition of complex component repair and replacement strategies.

We have also extended Arcade’s original capabilities for expressing dependability measures. Up to now it was only possible to specify under which circumstances a system is down in terms of Boolean expressions. We provide a small language that allows us to specify more complex dependability measures at the same abstract level at which dependability measures are specified in Arcade. We have defined a temporal stochastic logic, intSPDL onto which these requirements are mapped. Thus, we can use the entire powerful apparatus

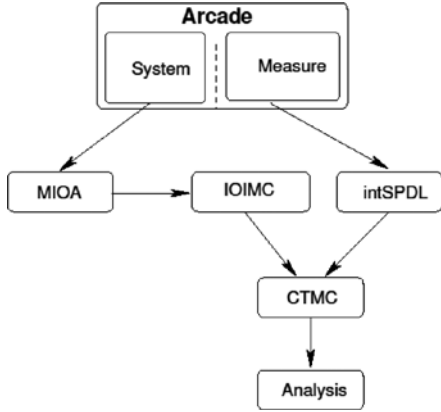


Figure 1. Overview of the Arcade approach.

of stochastic model checking for dependability evaluation. Figure 1 summarizes the application of the Arcade dependability evaluation framework. At the top-level we can specify the dependability behaviour and the dependability measures of interest of the system that is to be analysed using Arcade.

The system specification is mapped onto MIOA programmes, which in turn are mapped onto input/output interactive Markov chains (I/O-IMCs) (Boudali, Crouzen, and Stoelinga 2007b). The dependability measures are mapped onto intSPDL formulae. From the system’s I/O-IMCs and the intSPDL representation of the dependability measure that is to be verified, a simple continuous time Markov chain (CTMC) is derived, on which the actual dependability evaluation takes place.

The paper is further organised as follows. In Sec. 2 we introduce briefly, syntax and semantics of the MIOA language. In Sec. 3 we present both the logic interactive SPDL (intSPDL) for MIOA and model checking algorithms for intSPDL. In Sec. 4 we present the dependability evaluation framework Arcade and show how MIOA and intSPDL can be used in this context. We show how to apply our framework to the dependability evaluation of a simple model of a water distribution facility. Sec. 5 is devoted to the comparison with related work. Sec. 6 concludes this paper with a short summary and pointers to future research.

2 SYNTAX AND SEMANTICS OF MIOA

In this section we introduce syntax and semantics of the Markovian input/output automata (MIOA) language. MIOA can be seen both as an extension of IOA (Lynch and Tuttle 1989) and of I/O-IMCs.

2.1 Syntax of MIOA

Each MIOA programme is built as shown in Figure 2. In lines (2) to (5) we define the signature, i.e., the set of actions that can occur in the MIOA programme. This section is opened by the keyword signature: in line (2). In MIOA, we can distinguish between three types of actions:

1. *Input actions* (denoted $\langle \text{action} \rangle?$) are controlled by the environment. They can be *delayed*, meaning that a transition labelled with $\langle \text{action} \rangle?$ can only be taken if another MIOA programme performs an output action $\langle \text{action} \rangle!$
2. *Output actions* (denoted $\langle \text{action} \rangle!$) are controlled by the MIOA programme itself. In contrast to input actions, they cannot be delayed, i.e., transitions labelled 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.

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. $\langle \text{rate} \rangle$ is the parameter of an exponential distribution. In the section followed by keyword variables: (line (7)) we define the variables of the MIOA programme. Variables can be defined as arbitrary abstract data types. The abstract data types are defined in a mathematical sound way (Kuntz and Haverkort 2008). From line (10) (keyword transitions) on, the transitions are described. In general, the transitions are given in a precondition-effect-style. This means that in order to execute a transition labelled with an action or a rate, a certain precondition must be satisfied. Preconditions are expressions that can be evaluated either to true or false. For input-actions

```

(1) PROGRAMME: < programme_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) output: < action_i !>
(14) precondition: < side_cond >
(15) effect: . . .
(16) internal: < action_i >
(17) precondition: < side_cond >
(18) effect: . . .
(19) markovian: < rate_i >
(20) precondition: < pre_cond >
(21) effect: . . .
(22) hiding < action_1 !> . . . < action_n !>
  
```

Figure 2. Structure of MIOA programmes.

no precondition is required, since input-actions can always synchronise with a corresponding output action. To determine the effect of transitions in lines (12), (18), and (21) 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 variable'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 are possible by providing a *while*-loop construct. In line (22) the keyword *hiding* indicates that the sequel gives the list of output actions that are to be hidden, i.e., excluded from synchronisation. Two or more MIOA programmes can be composed in parallel. To this end, we simply need line (1) of Fig. 2, and add a second line PARALLEL: $\langle \text{programme_name} \rangle_1, \dots, \langle \text{programme_name} \rangle_n$. This indicates that the programmes $\langle \text{programme_name} \rangle_1$ to $\langle \text{programme_name} \rangle_n$ have to be composed in parallel.

Example: Programme in \MIOA-Syntax Let the MIOA-specification of Figure 3 be given, which is the textual representation of a simple failure model. In this model, some component can be either operational or failed. We assume, the component is repairable, i.e., if failed it can eventually become operational again.

Intuitively, the semantics of this failure model is as follows: The failure behaviour of the component is given by the Markovian transition with rate λ . When failed, the IO/IMC 'sends' the output signal *failed!*, the repair unit 'senses' this failed signal and moves to variable *busy*. This transition is labelled with input action *failed?* which means that the transition is only possible if in some other IO/IMC a corresponding output signal was sent.

```
(1) PROGRAMME: Failure
(2) signature:
(3) input: repaired?
(4) output: up!, failed!
(5) markovian:  $\lambda$ 
(6) variables:
(7) UP: Bool := true
(8) DOWN: Bool := false
(9) i1 : Bool := false
(10) i2 : Bool := false
(11) transitions:
(12) input: repaired?
(13) effect: if DOWN = true
(14) DOWN := false ; i2 := true
(15) else
(16) DOWN := DOWN ; UP := UP ; i1 := i1 ; i2 := i2;
(17) output: failed!
(18) precondition: i1 = true
(19) effect: i1 := false ; DOWN := true
(20) output: up!
(21) precondition: i2 = true
(22) effect: i2 := false ; UP := true
(23) markovian:  $\lambda$ 
(24) precondition: UP = true
(25) effect: UP := false ; i1 := true
```

Figure 3. MIOA specification.

Afterwards, the system can be repaired. It stays in its *down* state, until it can synchronise with a *repaired!* output signal from its corresponding repair unit. This forced waiting via synchronisation can be reached by adding the corresponding input signal *repaired?* to the outgoing transition of the *down* state.

Finally, the component sends an *up!* output signal.

As synchronisation between programmes is a multiway-synchronisation, besides the repair unit also a spare management unit could be synchronised over the component's *failed!* signal. In this case, the spare management unit would also be synchronised with the *up!* output signal via an *up?* input signal, to indicate, that the primary component can take over again. In line (1) the keyword PROGRAMME 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. Two types of actions are used here: 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 variables of the MIOA programme are defined. All variables 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 variables obtain an initial value, which is true in case of "UP" and false in all remaining cases. In the remaining lines, the transitions of the MIOA programme are described. In the case of input actions the preconditions are empty, which means that they are equals to true (cf. line (12), input action "*repaired?*"). Here, the variable values are changed only, if *DOWN* is true, otherwise, the variable values remain unchanged (lines (15) to (16)). In lines (17) to (22) the transitions labelled with output actions are described. The transition labelled with *failed!* can only be taken if the precondition (line (18)) evaluates to true, i.e., if the variable *i1* is true. Likewise, the *up!*-transition can only be executed, if the variable *i2* is true. From line (23) on, the Markovian behaviour is described. The component fails with rate λ , if the *UP*-variable has value true.

2.2 Semantics of MIOA

The semantics of $\$MIOA\$$ consists of two parts:

(1) The semantics of the used data types and operations, (2) the semantics of the MIOA programmes. The data type semantics can be defined in the usual mathematical axiomatic style. For a more thorough treatment of this, cf. (Kuntz and Haverkort 2008).

Due to space restrictions, we cannot give the full formal semantics of MIOA, but describe

```

(1) stateset() := ∅
(2) states to process := ∅
(3) transitionset := ∅
(4) s := initialize MIOA.states
(5) stateset := stateset ∪ {s}

(6) states to process := states to process ∪ {s}

(7) while states to process != ∅
(8) s := choose(states to process)
(9) forall MIOA.transition t do
(10) if precondition(s) = true then
(11) snew := s.apply(t.effect)
(12) if snew ∈ stateset then
(13) stateset := stateset ∪ {snew}

(14) states to process := states to process ∪ snew
(15) endif
(16) transitionset := transitionset ∪ {s.id, new.id,
t.signal}
(17) else
(18) if t.signal = input then
(19) transitionset := transitionset ∪ {s.id, s.id,
t.signal}
(20) endif
(21) endif
(22) endfor
(23) endwhile

```

Figure 4. Generation of IO/IMCs from MIOA programmes.

instead how to derive an IO/IMC from the MIOA programme in the style of a depth-first search (cf. Figure 4).

A similar algorithm was presented in (Sözer 2009), where MIOA was used for the analysis of fault-tolerant software architectures. For a formal treatment of the semantics we again refer to (Kuntz and Haverkort 2008). The algorithm in Figure 4 is a simple DFS-algorithm that generates the state-space, i.e., the I/O-IMIC from the given MIOA-programme. The only specialty is in lines (18) to (22): As an IO/IMC must be able to synchronise in any state with a corresponding output signal from another ioimc, we have to add a self loop with the input action label. This states, that any output action, can be “processed”, but that this does not have an effect on the system state, i.e., on the system behaviour.

3 A LOGIC FOR MIOA

In this section we introduce the logic intSPDL (interactive SPDL), a stochastic logic that like SPDL (Kuntz and Siegle 2006) allows to reason about the behaviour of MIOA specifications at the level of action sequences. To an external observer, the behaviour of a MIOA specification can be seen as a stream of actions, which are the basic elements of the behavioural description. In contrast, using logics like CTL, or in the stochastic world CSL (Aziz, Sanwal, Singhal, and Brayton 1996; Baier, Haverkort, Hermanns, and Katoen 2003), the desired behaviour can only be described at the level of the state space of the underlying semantic model.

3.1 Syntax of intSPDL

The logic intSPDL is a stochastic extension of the logic PDL (propositional dynamic logic) (Fischer and Ladner 1979), 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. 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 changed or added the following operators to obtain intSPDL: The original \diamond -operator has been extended to the path-operator $\Phi[\rho]^{[t,t']}\Psi$ (cf. Sec. 3.2): by specifying lower time bound t and upper time bound t' within which the Ψ state has to be reached, visiting only Φ -states before, thereby executing programme ρ , a probabilistic path quantifier $\mathcal{P}_{>p}(\Phi[\rho]^{[t,t']}\Psi)$ to reason about the transient probabilistic behaviour of a system, and a steady-state $\mathcal{S}_{>p}(\Phi)$ operator to reason about the behaviour of the system once stationarity of the underlying Markov chain is reached. Programmes ρ are composed of the usual operators known from the theory of regular expressions (Hopcroft and Ullman 1979): (sequential composition), \cup (choice), and $*$ (Kleene star) that have their usual meaning. Programmes extend regular expressions by the operator $\Phi?$; ρ (resp. $\Phi?$; a), 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. From language 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.

3.2 Semantics and model checking of intSPDL

Due to lack of space, we will describe the semantics and model checking of intSPDL only in a very informal way. For details, we refer to (Kuntz and Haverkort 2008).

Semantics: 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.
- $\mathcal{S}_{>p}(\Phi)$ asserts that the steady-state probability to reside in a Φ -state, once the system has reached stationarity satisfies the bounds as given by $>p$.
- $\mathcal{P}_{>p}(\Phi[\rho]^{[t,t']}\Psi)$ asserts that the probability measure of the paths that satisfy $\Phi[\rho]^{[t,t']}\Psi$ is within the bounds as given by $>p$.
- The 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 to the Ψ state must correspond to the action sequence of 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 corresponding states on the path.

Model Checking: The basic approach of model checking intSPDL formulae 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 verified. Model checking of intSPDL steady state formulae $S \models \langle p(\Phi) \rangle$ requires the solution of a usually large system of linear equations. Thus, the model checking problem of intSPDL amounts to a problem of the numerical analysis of large CTMCs (Baier, Haverkort, Hermanns, and Katoen 2003; Kuntz and Haverkort 2008). Model checking of intSPDL probabilistic path formulae requires the construction of a product transition system between the semantic model of the original MIOA-specification and a finite automaton-like transition system that can be derived from the formula at hand that is to be verified. Based on this product transition system, a system of linear differential equations has to be solved in order to compute the actual satisfaction probability (Baier, Haverkort, Hermanns, and Katoen 2003; Kuntz and Haverkort 2008).

4 MIOA AND ARCADE

We will now show, how we can use the MIOA language as a semantic model for Arcade (Boudali, Crouzen, Haverkort, Kuntz, and Stoelinga 2008). The key 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. In (Boudali, Crouzen, Haverkort, Kuntz, and Stoelinga 2008) a predefined set of components along with an extensible set of features (such as interactions, dependencies, operational/failure modes, etc) was proposed. There, three main components with which a system model can be constructed in a modular fashion have been identified: (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 MIOA-programmes. A basic component represents a physical/logical system component that has a distinct operational and failure behaviour. 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 (realised 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., firstcome-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.

4.1 Dependability analysis of a Water Distribution acility

Structure of Water Distribution Facility: In Figure 5, we can find the global structure of a Water storage and Distribution Facility (WDF); the WDF consists of two water tanks, a pumping and a distribution station. The WDF has to provide drinking water for two districts. In this paper, we want to analyse the availability, reliability, as well as the survivability of the Distribution Station (DS) of the WDF.

Structure of Distribution Station: In Figure 6, we find the architecture of the DS. The DS consists of two input lines, six valves and a water tank.

Arcade Model of the Distribution Station: In Figure 7 the Arcade model of its dependability-relevant behaviour. We need BCs for each of the DS's components and define their respective failure and repair behaviour. A valve can, in principle, fail in two ways, it can be either stuck open or stuck closed. In the sequel, we assume that only stuck closed is a failure, as a stuck open valve would not interrupt water distribution. A tank can fail in different ways, it can leak, rupture, or become contaminated. For simplicity, we do not consider the different failures for the tank separately. Pipes are not considered to be subject to failures. If failed, valves and tank can be repaired. We assume that all the valves share a common repair unit, and that they are repaired according to a First-Come-First-Served (FCFS) repair strategy. The tank has its own RU.

4.2 Dependability analysis

In order to carry out dependability analysis, we have to specify the measures of interest, and we have to map both the Arcade model and the

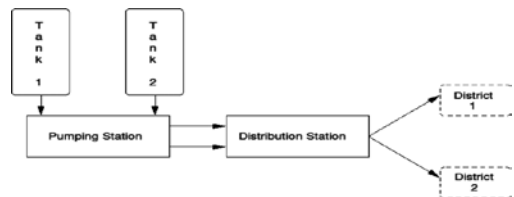


Figure 5. Water Distribution Facility.

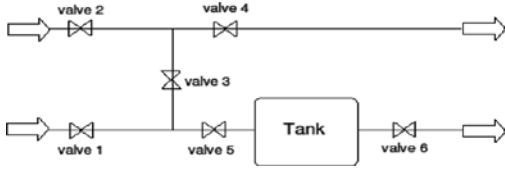


Figure 6. Water Distribution Station.

```

BC: tank                BC: valve j
OM: (UP, DOWN)         OM: (UP, DOWN)
TTF: exp( 1/2000 )     TTF: exp( 1 /8000 )
TTR: exp(1)            TTR: exp(1)
RU: RU.valve           RU: RU.tank
Components: valve1,...,valve6  Components: tank

```

Figure 7. MIOA specification of DS.

measures of interest onto their semantic representations, to finally carry out the analysis.

Semantic Representation of the Arcade Model: The semantic model of the valves and the tank is essentially the MIOA programme in Figure 3. This programme only has to be instantiated with the correct action names and failure rates. In Figure 8 we can find the transition part of the MIOA specification for the FCFS repair strategy.

The MIOA-programme in Figure 3 corresponds to the failure behaviour of the valves and the tank. The final semantic model, i.e., the model on which the dependability evaluation can be performed, is obtained by composing the semantic models of valves, tank and their respective repair units in parallel, using the composition operator defined for I/O-IMCs resp. MIOA (Kuntz and Haverkort 2008).

Measures of Interest: We are interested in the DS's steady-state availability, reliability over varying time bounds t and its survivability. First, we define the conditions, under which the DS is down, i.e., under which conditions at least one district does not receive water:

$$\Phi_{FT} := \text{valve 4.down} \vee \text{valve 5.down} \vee \text{valve 6.down} \vee \text{tank.down} \vee (\text{valve 1.down} \wedge \text{valve 2.down}) \vee ((\text{valve 1.down} \vee \text{valve 2.down}) \wedge \text{valve 3.down})$$

1. Availability is defined as the probability of the system being in an operational state during a mission time assuming that components are repaired. Here, we assume steady state availability, i.e., the mission time is infinite. The availability of the DS in terms of Arcade can be seen as the negation of the Boolean formula that can be derived from the fault tree:

Availability: $!\Phi_{FT}$

This formula can be translated automatically into an intSPDL formula of the following kind: $S_{>p}(!\Phi_{FT} FT)$ If not steady state, but timepoint

```

(1) PROGRAMME: Repair FCFS Valves
(2) signature:
(3) input: failed(6 : Int)?
(4) output: repaired(6 : Int)!
(5) markovian: (6 : Int)
(6) variables:
(7) available: Bool := true
(8) busy: Array[6 : Bool] := false
(9) internal: Array[6 : Bool] := false
(10) queue: Queue[6 : 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  $\in$  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

```

Figure 8. MIOA specification of FCFS repair strategy.

availability is considered, we would add “AT t ” in the above definition.

2. Reliability is defined as the probability of having no system failure within a certain mission time, assuming that no component is repaired. In terms of Arcade, we can express this as follows:

Reliability : Φ_{FT} within t

This will be translated into intSPDL, yielding the following formula:

$$P_{>p}(\text{true} [\text{Act}^* \setminus \{\text{Rep_Act}\}]^{0..t} \Phi_{FT}),$$

where Act is the set of actions defined in the MIOA programme, and Rep Act is the set of actions representing repair. That means, this formula excludes repairs from the set of satisfying paths, as required by the definition of reliability.

3. According to (Cloth 2006), survivability is defined as the ability to recover predefined service levels in a timely manner after the occurrence of disasters. Using MIOA, the class of such properties can be expressed as follows:

Survivability: After disaster reach
service_level within t

Both, disaster and service_level are Boolean formulae that define the relevant disaster and the service level of interest. This Arcade measure can be translated into intSPDL formulae obeying the following pattern:

Table 1. Dependability analysis results for DS.

Reliability	t = 170 0.7557	t = 350 0.5701	t = 500 0.4308	t = 650 0.3256
Survivability	t = 5 0.3404	t = 20 0.811	t = 30 0.9179	t = 40 0.9643

disaster $\Rightarrow P_{>cp}(\text{true } [\text{Act}]^{[0..1]} \text{ service_level})$

In this paper we define the disaster, to be the failure of the tank: disaster := tank.down and the service level that is to be reached, is full service:

service level := valve 1.up \wedge ... \wedge valve 6.up \wedge tank.up.

In Table 1 we find the results of dependability analysis for reliability and survivability. For the actual computation of values, we used the MRMC model checking tool (Katoen, Khattri, and Zapreev 2005). The steady state-availability computed to 0.84. Line immediately below the heading. Explanations should be given at the foot of the table, not within the table itself. Use only horizontal rules: One above and one below the column headings and one at the foot of the table (Table rule tag: Use the Shift-minus key to actually type the rule exactly where you want it). For simple tables use the tab key and not the table option. Type all text in tables in small type: 10 on 11 points (Table text tag). Align all headings to the left of their column and start these headings with an initial capital. Type the caption above the table to the same width as the table (Table caption tag). See for example Table 1.

5 RELATED WORK

Here, we distinguish between related work in the field of process algebras with data types and dependability formalisms:

1. We are aware of two approaches that extend process algebras, without stochastic timing, with abstract data types. LOTOS (Bolognesi and Brinksma 1989) 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 μCRL resp. mCRL2 (Groote and Ponse 1995; Groote, Mathijssen, Reniers, Usenko, and van Weerdenburg 2007) data types have also been defined. But also here, no stochastic extension is available.
2. Over decades, a wide variety of modeling approaches has been developed for evaluating system dependability. General purpose models,

such as CTMCs, stochastic Petri nets (SPNs) (Ajmone Marsan, Balbo, Conte, Donatelli, and Franceschinis 1995) and their extensions; stochastic process algebras (SPAs) (Hermanns, Herzog, Klehmet, Mertsiotakis, and Siegle 2000; Hillston 1996); interactive Markov chains (IMCs) (Hermanns 2002), and stochastic activity networks (SAN). These models are general-purpose, serving the specification and validation of a wide variety of quantitative properties of computer and communication systems, and certainly not of dependability properties only. In contrast, several dependability-specific approaches have also been developed, such as the System Availability Estimator (SAVE) language (Goyal, Carter, de Souza e Silva, Lavenberg, and Trivedi 1986), OpenSESAME (Walter, Siegle, and Bode 2007) and many others. Finally, for some architectural (design) languages specific “error annexes” have been developed to allow for dependability analysis, most notably, the architectural description language AADL (SAE 2006), and the UML dependability profile (OMG Group 2006).

6 CONCLUSIONS

In this work, we have introduced the Markovian I/OIMC language MIOA. MIOA is inspired by I/OIMCs (Boudali, Crouzen, and Stoelinga 2007a) and the input/output automata language (IOA) (Garland, Lynch, Tauber, and Vaziri 2004). We have defined the syntax and semantics of MIOA, a logic for MIOA, intSPDL, and their corresponding model checking algorithms. Using MIOA, we gave Arcade a formal semantics, that allows for the fully automatic generation of CTMCs from a given Arcade specification. This extends the semantics given in (Boudali, Crouzen, Haverkort, Kuntz, and Stoelinga 2008), where the semantic models for, e.g., repair strategies and sparse management had to be manually designed, according to the current Arcade-model. A few application examples of MIOA were given. In the future, we plan to extend MIOA to support data types with a potentially infinite carrier set. To this end, we have to introduce an extended semantic model, based on symbolic state graphs. We want to fully integrate the approach presented here, into an automated tool chain, up to now, the translation of the MIOA programmes and the measures of interest needs some manual interaction. 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. To this end, we plan to extend the approach of (Grunske 2008) and adapt

it accordingly to the needs of an action-oriented stochastic logic.

REFERENCES

- Ajmone Marsan, M., G. Balbo, G. Conte, S. Donatelli and G. Franceschinis (1995). *Modelling with generalized stochastic Petri nets*. Wiley.
- Aziz, A., K. Sanwal, V. Singhal and R. Brayton (1996). Verifying continuous time Markov chains. In *Computer-Aided Verification*, Volume LNCS 1102, pp. 146–162. Springer.
- Baier, C., B. Haverkort, H. Hermanns and J. Katoen (2003, July). Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Trans. Software Eng.* 29(7), 1–18. Bognesi, T. and E. Brinksma (1989). Introduction to the ISO Specification Language LOTOS. In *The Formal Description Technique LOTOS*, pp. 23–73. North-Holland, Amsterdam.
- Boudali, H., P. Crouzen and M. Stoelinga (2007a). A compositional semantics for Dynamic Fault Trees in terms of Interactive Markov Chains. In *Proceedings of ATVA 2007*, Volume LNCS 4762, pp. 441–456.
- Boudali, H., P. Crouzen B. Haverkort, M. Kuntz and M. Stoelinga (2008). Architectural Dependability Modelling with Arcade. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 512–521.
- Boudali, H., P. Crouzen, and M. Stoelinga (2007b). Dynamic fault tree analysis using input/output interactive markov chains. In *Proceedings of DSN 2007*, pp. 708–717. IEEE.
- Cloth, L. (2006). *Model Checking Algorithms for Markov Reward Models*. Ph. D. thesis, University of Twente, Enschede, Netherlands.
- Fischer, M. and R. Ladner (1979). Propositional dynamic logic of regular programs. *J. Comput. System Sci.* 18, 194–211.
- Garland, S., N. Lynch, J. Tauber and M. Vaziri (2004). IOA User Guide and Reference Manual. Technical Report MITLCS-TR-96, Massachusetts Institute Technology, Cambridge, MA.
- Goyal, A., W.C. Carter, E. de Souza e Silva, S.S. Lavenberg and K.S. Trivedi (1986, July). The system availability estimator. In *Proceedings of the 16th Int. Symp. on Fault-Tolerant Computing*, pp. 84–89.
- Groote, J. and A. Ponse (1995). The syntax and semantics of μ CRL. In *Algebra of Communicating Processes '94*, Workshops in Computing Series, pp. 26–62. SV.
- Groote, J.F., A. Mathijssen, M. Reniers, Y. Usenko and M. van Weerdenburg (2007). The Formal Specification Language mCRL2. In *Methods for Modelling Software Systems (MMOSS)*, Number 06351 in Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl, Germany.
- Grunske, L. (2008). Specification Patterns for Probabilistic Quality Properties. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, New York, NY, USA, pp. 31–40. ACM.
- Hermanns, H. (2002). *Interactive Markov Chains*, Volume 2428 of *Lecture Notes in Computer Science*. Springer.
- Hermanns, H., U. Herzog, U. Klehmet, V. Mertsiotakis and M. Siegle (2000, January). Compositional performance modelling with the TIPPTool.
- Performance Evaluation* 39(1–4), 5–35. Hillston, J. (1996). *A Compositional Approach to Performance Modelling*. Cambridge University Press.
- Hopcroft, J.E. and J.D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company.
- Katoen, J.-P., M. Khattri and I.S. Zapreev (2005). A Markov Reward Model Checker. In *QEST '05*, pp. 243–244. IEEE Computer Society.
- Kuntz, M. and B. Haverkort (2008). Formally Founding Dependability Engineering with MIOA. Technical report, Universiteit Twente, TR-CTIT-08-39.
- Kuntz, M. and M. Siegle (2006). Symbolic Model Checking of Stochastic Systems: Theory and Implementation. In *13th International SPIN Workshop*, pp. 89–107. Springer, LNCS 3925.
- Lynch, N. and M. Tuttle (1989). An Introduction to Input/output Automata. *CWI Quarterly* 2(3), 219–246.
- OMG Group (2006, june). UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. Technical report. SAE (2006, June). SAE Architecture Analysis and Design Language (AADL) Annex Volume 1. SAE standards AS5506/1. Available at <http://www.sae.org/technical/standards/AS5506/1>.
- Sözer, H. (2009). *Architecting Fault-Tolerant Software Systems*. Ph. D. thesis, University of Twente.
- Walter, M., M. Siegle and A. Bode (2007). OpenSESAME: The Simple but Extensive, Structured Availability Modeling Environment. *Reliability Engineering and System Safety* 93(6), 857–873.