Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on
Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, June 24-27, Anchorage, USA. pp. 512-521. Digital Object Identifier: 10.1109/DSN.2008.4630122

Architectural dependability evaluation with Arcade*

H. Boudali[1] P. Crouzen[2] B. R. Haverkort[1] M. Kuntz[1] M.I.A. Stoelinga[1]
[1] University of Twente, Department of Computer Science, Enschede, NL
[2] Saarland University, Department of Computer Science, Saarbrücken, D

Abstract

This paper proposes a formally well-rooted and extensible framework for dependability evaluation: Arcade *(architectural dependability evaluation). It has been designed to combine the strengths of previous approaches to the evaluation of dependability. A key feature is its formal semantics in terms of Input/Output-Interactive Markov Chains, which enables both compositional modeling and compositional state space generation and reduction. The latter enables great computational reductions for many models. The* Arcade *approach is extensible, hence adaptable to new circumstances or application areas. The paper introduces the new modeling approach, discusses its formal semantics and illustrates its use with two case studies.*

1 Introduction

Now that computers and communication systems are proliferating in all kinds of devices and home appliances, high-dependability is no longer restricted to systems that are being used in traditional safety- or mission-critical applications, such as space and aircraft or (nuclear) power control systems. An important difference with these traditional systems, however, is that although high dependability is a key concern, achieving it should be affordable in terms of costs. Hence, high dependability must be achieved as a "by product" of a sound design and implementation trajectory, at almost no additional costs. Therefore, dependability evaluation techniques are being integrated in design frameworks, to enable a cost-efficient comparison of design alternatives with respect to the dependability requirements.

Although the standard theory of reliability engineering has been around for many years now [20], the actual use of these methods during the design of computer and communication systems is far less common. Nevertheless, a wide variety of modeling approaches has been developed for evaluating system dependability. We categorize them in three classes: (1) General purpose models, such as CTMCs, stochastic Petri nets (SPNs) [3] and their extensions; stochastic process algebras (SPAs) [14, 15]; interactive Markov chains (IMCs) [13], Input/Output IMCs (I/O-IMCs) [5], and stochastic activity networks (SAN) as used in UltraSAN and Möbius [19]. These approaches 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. (2) In contrast, several dependability-specific approaches have also been developed, such as reliability block diagrams (RBDs), the System Availability Estimator (SAVE) language [12], dynamic RBDs (DRBDs) [9]; dynamic fault trees (DFTs) [10] and extended fault trees (eFTs) [7]; OpenSESAME [21], and TANGRAM [8]. (3) Finally, for some architectural (design) languages specific extensions have been developed to allow for dependability analysis, most notably, the error annex of the architectural description language AADL [2], and the UML dependability profile [17].

We have identified five criteria, a good dependability formalism in our opinion should satisfy: (1) Modeling effort: how easy is it to model a system and its dependability aspects? (2) Expressiveness: what features (repair, spare management, different failure modes, etc.) can be modeled and can new ones easily be added? (3) Formal semantics: is the meaning of the models unambiguously clear? (4) Compositionality: we distinguish between (4a) Compositional modeling, meaning that a model can be created by composing smaller submodels and (4b) Compositional state space generation and reduction. Compositional state space generation means that the state space of the entire model is constructed out of the state spaces of its constituent subcomponents. Compositional state space reduction means that the global state space of a multi-component system is obtained by repeated composition and reduction (e.g., by bisimula-

*This research has been partially funded by the Netherlands Organization for Scientific Research (NWO) under FOCUS/BRICKS grant numbers 642.000.505 (MOQS) and 542.000.504 (VeriGem); by the EU under grant numbers IST-004527 (ARTIST2); and by the DFG/NWO bilateral cooperation programme under project number DN 62-600 (VOSS2).

tion reduction). (5) Tool support: are tools available for automatic analysis?

The general-purpose formalisms, specifying system models in terms of states and transitions, have the advantage of being very flexible (hence, expressive) and precise. But, with these formalisms, it is often difficult to specify dependability models, since they do not provide any dependability-specific constructs, which in turn may lead to specifications that are hard to understand and thus are potentially subject to modelling errors. We also found that dependability specific approaches score relatively low on expressiveness; although each of them incorporates certain dependability constructs, none of them includes them all. Although we agree that it is impossible to include all possible features, we do think that a modeling approach should be extensible (cf. Section 3.6), so as to be able to accommodate any, also future, needs. Architectural languages require limited modeling effort, since they annotate architectural models (which play an important role throughout the design). However, these languages, as we know them, lack a formal and compositional semantics and tool support for automatic dependability evaluation, although recently some work in this direction has been done [18].

In this paper, we therefore propose a new, formally well-rooted and extensible framework for dependability evaluation that satisfies the five criteria we have discussed above: Arcade (for architectural dependability evaluation). In addition, we define our framework in an architectural style, i.e., we define a system model in terms of components or entities that (directly) map to actual physical/logical system components. In fact, our framework is ultimately intended to be incorporated into an architectural design language. Arcade 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. Arcade models have a semantics in terms of I/O-IMCs, thus pinning down their interpretation in an unambiguous way. Moreover, the compositional state space generation and reduction technique for I/O-IMCs also enables an efficient analysis of very large Arcade models.

The paper is further structured as follows. In Section 2 we provide background on IMCs and I/O-IMCs, the underlying semantical models used in the remainder of this paper. Section 3 introduces the Arcade modeling approach. Section 4 describes the currently employed tool-chain to evaluate Arcade models, whereas Section 5 reports on two cases studies. Section 6 concludes the paper.

2 Input/Output Interactive Markov Chains

Input/Output interactive Markov chains (I/O-IMCs) [5] are a combination of Input/Output automata (I/O-automata) [16] and interactive Markov chains (IMCs) [13]. I/O-IMCs distinguish two types of transitions: (1) *Interactive transitions* labeled with actions (also called signals); (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 $a?$.

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.

States are depicted by circles, initial states by an incoming arrow, Markovian transitions by dashed lines, and interactive transitions by solid lines. Fig. 1 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

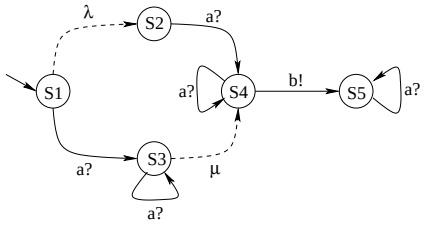


Figure 1. Example of an I/O-IMC

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

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

input action $a?$). I/O-IMCs can be combined with a parallel composition operator "||", to build larger I/O-IMCs out of smaller ones. The behavior of $P = Q||R$, 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., $Q||R$ can evolve to $Q'||R$ ($Q||R'$).

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., $Q||R$ evolves simultaneously into $Q'||R'$. Note that when an output and an input action synchronize the result is an output action.

Like in process algebras, the hiding operator hide *A* in P makes output actions in a set *A* internal, such that no further synchronization is possible over actions in *A*. More details on the I/O-IMC formalism can be found in [5].

3 Arcade: Semantics and Syntax

This section describes the semantics and syntax of Arcade. We have identified three main building blocks 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). These building blocks interact with each other by sending and receiving input/output actions. The semantics of these building blocks and their interactions is based on the I/O-IMC framework. In the following, we describe each of these building blocks.

3.1 Basic component

The basic component building block represents a physical/logical system component that has a distinct operational and failure behavior. There are two steps involved in defining a BC: (1) defining the BC's operational modes, (2) defining the BC's failure model. In theory, there could be a different failure model (failure behavior) for each of the BC's operational modes; however, for simplicity we will restrict these differences.

3.1.1 Operational modes

A basic component can be in various operational modes (OM). Examples of operational modes include *active* versus *inactive*, which are two typical modes of operation when a component is used as a primary or as a spare. We define operational modes in terms of groups of operational modes. A group of operational modes defines a set of *mutually exclusive* operational modes, e.g. active mode versus inactive mode. At the I/O-IMC level, a mode corresponds to an operational state. Thus, each OM group defines a set of operational states. If a BC has multiple OM groups, then the BC operational states consist of the *cross-product* of the operational states of the different OM groups. For example, let's assume a BC has two OM groups: inactive/active and on/off. In this case, the BC has four operational states, namely: (active,on), (active,off), (inactive,on), and (inactive,off). Switching from one operational mode to another needs to be defined as an *input action* at the I/O-IMC level. The mode switching or transition is thus triggered by some external event. Fig. 2 shows, for the example, the two OM groups (along with the mode switches) of the BC and the resulting operational states[2].

A user can essentially specify any number of OM groups as long as for each group the mode switches are clearly defined. At this point, we have identified a predefined set of OM groups from which a user can chose:

1. *active/inactive*: As explained earlier, this OM group allows the modeling of a component acting as a spare (and thus typically having a reduced failure rate while in inactive mode). The activation and deactivation signals (causing the mode switching) are managed by a spare management unit (cf. Section 3.3).

2. *on/off*: This group allows to model, for instance, the fact that if the power fails, then the BC is shut down and can no longer fail (i.e., its failure rate equals zero).

3. *accessible/inaccessible*: This group is used to model a *non-destructive functional dependency* (as in, e.g., [10, 21]); for example a database becomes inaccessible if the bus linking to it fails. Switching from accessible to inaccessible does not mean that the component

[2]For readability, all input self loops have been omitted.

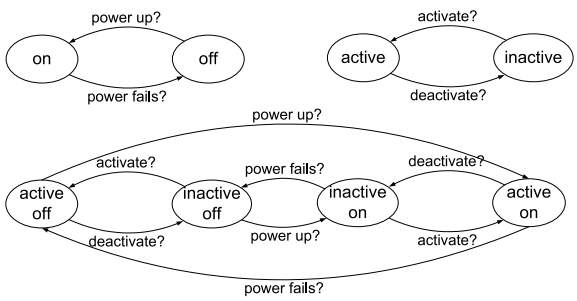


Figure 2. Defining operational modes of a BC.

has failed (hence, no repair is initiated). However, to the environment, i.e., the rest of the system, an inaccessible component might or might not be, depending on the system at hand, viewed as a failed component. While defining a BC, the user has to specify if inaccessibility is seen by the environment as a failure or not (cf. Section 3.5).

4. *normal/degraded*: This group is useful to model degraded modes of operation. A prime example is *load sharing*, where a component switches to a degraded operational mode (and consequently exhibits an increased failure rate) in case the component with which it is sharing the load fails. It is of course possible to model a group with more than two modes, e.g., normal/degraded$_1$/degraded$_2$/... /degraded$_n$.

Whenever a mode switch (except for the *active/inactive* OM group) occurs, this is due to failure or repair events of other components (this is further explained in Section 3.5).

3.1.2 Failure model

We attach a failure model to each BC operational state. For simplicity and to keep the framework well-structured, the failure model of each operational state is essentially the same except for possibly different values of Markovian rates[3].

The failure model describes how a BC fails, i.e., how it moves from an operational (or up) state to a failed (or down) state and visa versa. We distinguish two ways in which a component can fail: (1) an *inherent failure* specified as a Markovian transition, and (2) a failure due to a *destructive functional dependency* (as in [21, 10]) specified as a transition to a failed state upon the receipt of a signal (denoted DF). The destructive functional dependency is due to the failure of another (or many other) component; e.g., a processor depends upon the functioning of a fan and if the fan fails, the processor overheats and fails as well. The two types of failures lead to slightly different behavior, in particular when dealing with repair. The failure model consists of an UP, $DOWN_M$, $DOWN_{DF}$, and some (numbered) intermediate states. Fig. 3 shows the I/O-IMC representing a BC failure model. The upper portion (states 1, 2, and $DOWN_M$) represents an inherent failure. The lower portion (states 3, 4, 5, and $DOWN_{DF}$) represents a destructive functional dependency failure. UP is the starting operational state, and state 6 is added to properly handle the transitions between the two portions of the I/O-IMC.

From the UP state, the BC can fail in two ways by moving to state 2 or state 3 respectively, and from these states the BC immediately moves to the down states by outputting the corresponding failure signal ($failed_{df}$! and $failed_m$!). Any other BC, RU, or SMU can now respond to these output failure signals. Once a BC has failed, it waits for a $repaired?$ input signal (cf. Section 3.2 for details on the interaction with an RU) and then immediately outputs an up! signal and moves to its operational state. The rate λ in Fig. 3 is only a placeholder which will be instantiated once the failure model is combined with the operational modes. Note that if the BC fails due to a destructive functional dependency and is repaired without the component upon which it depends being operational, then the repair does not lead to an operational state (transition from $DOWN_{DF}$ to state 3). Signal $\neg DF$ is the complement of DF. Typically DF corresponds to the $failed$ ($failed_{df}$ or $failed_m$) signal of another component and $\neg DF$ corresponds to its up signal. If a BC has no destructive functional dependency, we can discard states 3, 4, 5, 6, and $DOWN_{DF}$.

In the failure model, we also allow multiple *failure modes* with regard to the inherent failure. In this case, if the BC has n failure modes, the user needs to specify n probabilities[4], and the Markovian rates are adjusted accordingly. Fig. 4 shows the I/O-IMC of a BC with two failure modes (with probabilities $1-p$ and p).

3.1.3 Combining operational modes and failure model

To obtain the final I/O-IMC model of a BC, we simply superimpose the failure model on each operational state, with the UP state of the failure model corresponding to an operational state. Note that here, different Markovian rates can be specified according to the operational state the BC is in. Fig. 5 shows the I/O-IMC model of a BC having operational modes of Fig. 2 and a failure model with one failure mode and no destructive functional dependency[5]. Note the different rates used in the model. There is, of course, a syntactic way for specifying these rates (cf. Section 3.5).

[3] In the future, one might allow for more flexibility in the failure model used for each operational state.

[4] With their sum being 1.

[5] For readability, we omitted the transitions from the four unnumbered states which are similar to the ones for states 1 through 4.

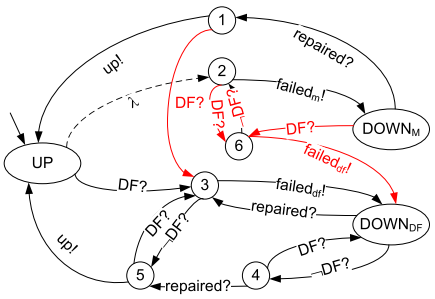


Figure 3. The failure model of a BC.

3.2 Repair unit

Repairing a component or a set of components is handled in a separate entity called the repair unit (RU). In fact, this allows handling complex *repair policies* and *repair dependencies* between various components. The RU semantics is again described in terms of an I/O-IMC with failure signals as inputs and repair signals as outputs. The RU is also aware

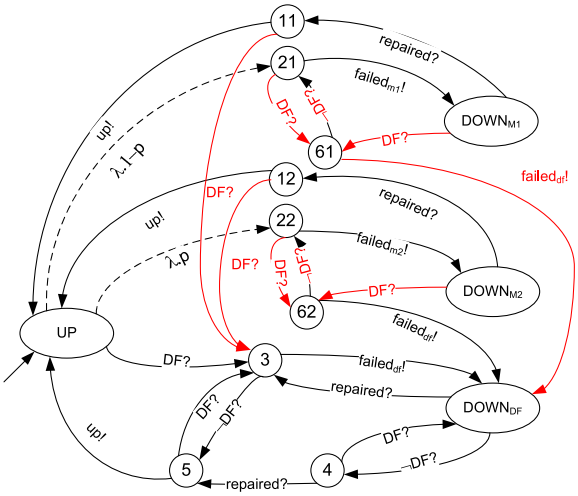


Figure 4. The failure model of a BC having two failure modes.

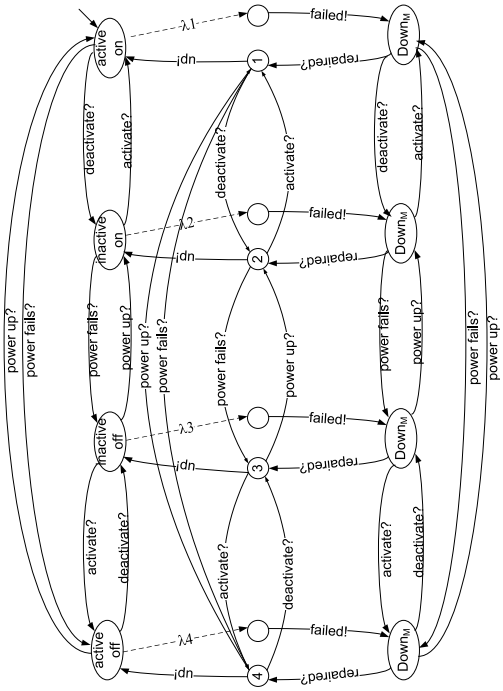
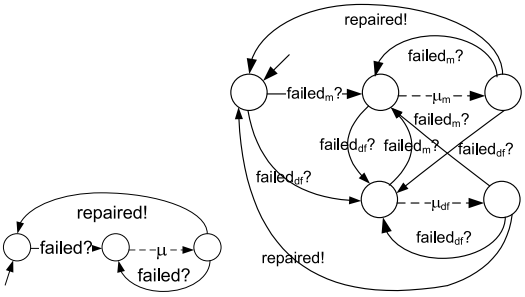


Figure 5. The I/O-IMC model of a BC.



(a) Single failure mode. (b) Multiple failure modes.

Figure 6. Dedicated repair strategy.

of all rates related to repair times. In short, the RU listens to failure signals output by one or many components, picks a component (given some policy) and initiates a repair operation according to a specific repair rate, and finally outputs the appropriate $repaired$! signal when the repair is finished. This procedure is then repeated. We allow at most one RU per component.

So far we have considered the following repair configurations/strategies: (1) *dedicated* repair, where each component has its own RU, (2) *first come first served (FCFS)*, (3) FCFS with *non-preemptive priorities (PNP)*, and (4) FCFS with *preemptive priorities (PP)*.

Fig. 6(a) shows the I/O-IMC of the dedicated repair policy and Fig. 7 shows the I/O-IMC of the FCFS repair strategy with two components A and B. Note that the I/O-IMC models of the FCFS, PP, and PNP can get quite large with increasing number of components. This is essentially due to the fact that the RU needs to keep track of the failing components and the order in which the failures occurred.

Often a component has multiple failure modes, i.e., different failure signals, and different repair rates for each mode. Fig. 6(b) shows a dedicated RU for a component having two different failure signals, i.e., two failure modes,

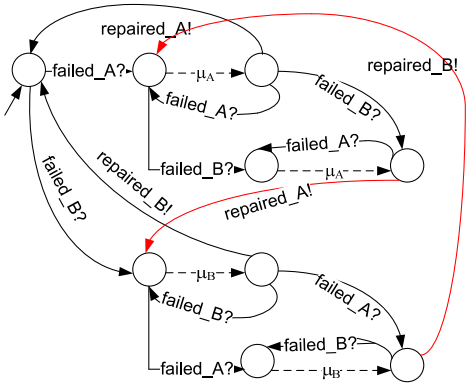


Figure 7. FCFS repair strategy.

with two repair rates, μ_m and μ_{df} respectively.

3.3 Spare management unit

The spare management unit (SMU) handles the activation and deactivation of spare components. Two configurations are possible at this point:

1. **One primary and one spare:** In this configuration, the assumption[6] is that the primary component is always in active mode, and thus always providing the service whenever it is operational. In fact, the primary component does not have an inactive mode per se and is therefore never activated or deactivated by the SMU. When the primary fails, the SMU activates the spare component which takes over the primary. As soon as the primary is up again, the spare is deactivated and the primary resumes operation. The I/O-IMC model of the SMU is shown in Fig. 8.

2. **One primary and two or more spares:** This configuration can be modeled based on the previous configuration; however, due to lack of space it will not be further discussed here.

3.4 System failure evaluation

Once all the basic components and units have been defined along with their interactions and dependencies, we need to specify the condition under which the whole system is failed or operational. We chose a fault tree representation (i.e., an AND/OR expression whose literals are failure modes of the BCs)[7] as the system evaluation criterion[8]. A fault tree also has a corresponding I/O-IMC model [6]. Thus, the entire system failure/operation is represented as an I/O-IMC. A simple example would be a system comprised of two redundant processors; the system fails if both processors fail. In this case, the whole system failure/operation would be modeled by a fault tree consisting of a repairable AND gate with the two processors as inputs. The repairable AND gate represents the overall system failure/operation and has a corresponding I/O-IMC model [6].

[6]Other assumptions, e.g., treating symmetrically both components, are possible at the cost of complicating the SMU I/O-IMC model.

[7]We can also use the K/M gate as a shorthand notation.

[8]We can also consider adding the Priority-AND gate [10].

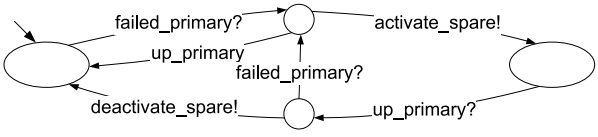


Figure 8. The SMU I/O-IMC model.

3.5 Syntax

This section describes the (textual) syntax of Arcade, i.e., the syntax for the BC, the RU, and the SMU.

3.5.1 BC syntax

(1) **COMPONENT:** `Name`
(2) **OPERATIONAL MODES:** `List of OM groups`
(3) **ACCESSIBLE-TO-INACCESSIBLE:** `AND/OR expr.`
(4) **INACCESSIBLE MEANS DOWN:** `YES or NO`
(5) **ON-TO-OFF:** `AND/OR expression`
(6) **NORMAL-TO-DEGRADED:** `AND/OR expr.`
(7) **TIME-TO-FAILURES:** $exp(\lambda_1), exp(\lambda_2), \cdots, exp(\lambda_m)$
(8) **FAILURE MODE PROBABILITIES:** $Prob_1, ..., Prob_n$
(9) **TIME-TO-REPAIRS:** $exp(\mu_1), \cdots, exp(\mu_n), exp(\mu_{df})$
(10) **DESTRUCTIVE FDEP:** `AND/OR expression`

Line (1) defines the unique name of the BC. Line (2) defines the OM groups of the component. At this point, the BC syntax is limited with respect to the restriction we have on the OM groups that are available to the user. Each of line (3), (5), and (6) defines an expression which tells us when exactly a mode switch occurs, e.g., if the AND/OR expression for an on/off switch evaluates to true, then the BC switches from 'on' to 'off'. If later the evaluation of the expression changes to false, the BC switches back to 'on'.

The mode switches of a component Y are expressed in terms of the failure modes of other components. A simple example would be if we have for Y **ON-TO-OFF:** $X.down$, then this means that Y switches from mode 'on' to mode 'off' upon the failure of X. Moreover, it is also implicit that Y switches back to its 'on' mode upon the repair of X. The active/inactive mode transitions are handled by an SMU through the $activate$ and $deactivate$ signals.

Line (4), specifies if the inaccessibility of the BC is seen as a failure by the environment (c.f. Section 3.1.1). Line (7) defines the time-to-failure distribution for each operational state[9] (e.g., in Fig. 5, the BC has four operational states, therefore the user needs to provide four distributions). Line (8) defines the n probabilities corresponding to n failure modes. Line (9) defines the time-to-repair distributions for each of the n failure modes and the distribution associated to the destructive functional dependency. Finally, line (10) specifies the condition under which the BC fails due to a destructive functional dependency.

All the distributions defined in lines (7) and (9) can, in general, be any phase-type distribution (see an example in Section 5).

[9]The order in which the OM groups are listed determines which distribution matches which operational state. The same goes for the repair distributions w.r.t. failure modes.

3.5.2 RU syntax

(1) **RU:** `Name`
(2) **COMPONENTS:** `comp1, comp2,..., comp`n
(3) **STRATEGY:** `Dedicated | FCFS | PP | PNP`
(4) **PRIORITIES:** `pr1, pr2,..., pr`n

Line (2) lists all the n component names that are repairable by the unit. The RU I/O-IMC model varies depending on the number of failure modes of the components (cf. Section 3.2). Line (3) specifies the repair policy. Line (4) defines the priority values (i.e., non-zero integer) of the various components in case of a PP or a PNP repair strategy.

3.5.3 SMU syntax

(1) **SMU:** `Name`
(2) **COMPONENTS:** `primary, sp 1,..., sp` n

Line (2) defines a primary component and n possible spare components for that primary.

3.5.4 System failure evaluation syntax

(1) **SYSTEM DOWN:** `AND/OR expression`

Line (1) defines the condition under which the system is failed (cf. Section 3.4 for more details). The elementary conditions under which the system fails, are expressed in terms of the failure modes that are defined for the component. If for a component more than one failure mode is defined, then the user has to specify the failure mode that is relevant for the system failure evaluation. For example, component X has two failure modes, and mode 2 is relevant for the evaluation, then the user writes $X.down.m2$ to state that mode 2 is the relevant failure mode. If there is only one failure mode, we can simply write $X.down$.

3.6 Extensibility

Arcade is extensible in the sense that it is easy to incorporate new or additional dependability constructs the user may think are important for his/her needs. All that has to be done is to provide the syntax, i.e., the Arcade specification of that additional construct, and its semantics in terms of an I/O-IMC model. State space generation, reduction and analysis do not have to be changed at all.

As an example, a simple *failover time* (i.e. the time it takes for an SMU to detect the primary failure and activate the spare component is exponentially distributed rather than instantaneous) can be added to the framework in the following way: First, Arcade's syntax is extended (here for an SMU with one primary and one spare):

(1) **SMU:** `Name`
(2) **COMPONENTS:** `primary, sp 1`
(3) **FAILOVER-TIME:** $exp(\delta)$

Secondly, the I/O-IMC model has to be defined (Fig. 9), which is an extension of the semantic model of an SMU (Fig. 8).

4 System dependability evaluation

To evaluate Arcade models, we use a three step approach, similar to the one in [5], using the CADP toolset [11].

First, we translate (according to the models defined in Section 3) all basic components, spare management units, repair units, and system failure evaluation models into their underlying I/O-IMCs. This translation step has not been automated yet.

The second step is to combine these models to obtain the overall system model. To this end, we use the Composer tool [5], which incrementally composes (using the well-defined parallel composition operator) the I/O-IMC models. Each composition step is followed by an aggregation (i.e., state minimization or reduction) step. The order in which the I/O-IMC models are composed is given by the user. This compositional aggregation approach has proved to be crucial in alleviating the state-space explosion problem. The output of the Composer tool is a single I/O-IMC, modeling the entire system. This I/O-IMC has two output signals: $failed!$ to denote the failure and $up!$ for the restoration of the system. Our Composer tool, which uses the CADP toolset, fully automates the composition and aggregation steps.

In a third step, we convert this system I/O-IMC into a labelled CTMC on which standard CTMC solution techniques to compute availability and reliability can be performed. This step has been automated, using the CADP toolset.

5 Case studies

To demonstrate the feasibility and usability of Arcade, we address two case studies from the literature. In Section 5.1 we analyze a distributed database system (DDS),

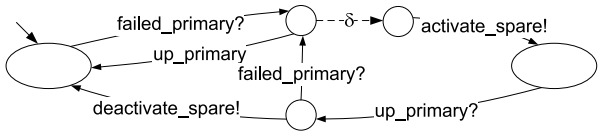


Figure 9. The SMU I/O-IMC model with failover time.

which was evaluated in [19] using SANs. In Section 5.2, we analyze a cooling system of a nuclear reactor, which was evaluated in [7] using eFTs.

5.1 Distributed database architecture

This system consists of two processors, one of which is a spare; four disk controllers, divided into two sets; and 24 hard disks, divided in 6 clusters, i.e., each cluster consisting of four disks. Each controller is responsible for three disk clusters, and each of the twelve disks, which the controller set is responsible for, is accessible by any of the two controllers in the respective set. Furthermore, each processor can access each of the four disk controllers. The processors are administrated by a spare management unit and share one repair unit. For each disk controller set and disk cluster there is a separate repair unit responsible. All repair units choose the next item to be repaired according to a FCFS repair strategy.

The system is down, if (at least) one of the following conditions is met: (1) all processors are down, or (2) in at least one controller set, no controller is operational, or (3) more than one disk in a cluster is down.

5.1.1 Arcade model

The Arcade models for the components of the DDS system are fairly simple. Most components have no distinguished OMs, except the spare processor which has OM group *(inactive, active)*. If there are no special OMs to be considered, the line **OPERATIONAL MODES** can be omitted.

1. Arcade model of processors:

   (a) Primary processor

   COMPONENT: pp
   TIME-TO-FAILURE: $exp(\frac{1}{2000})$
   TIME-TO-REPAIR: $exp(1)$

   The disk controllers ($dc\_i, i = 1, ..., 4$) and the disks ($d\_j, j = 1, ..., 24$) have the same Arcade model, except for a different time-to-failure in case of the disks, which is $exp(\frac{1}{6000})$.

   (b) Spare processor:

   COMPONENT: ps
   OPERATIONAL MODES: $(inactive, active)$
   TIME-TO-FAILURE: $exp(\frac{1}{2000})$, $exp(\frac{1}{2000})$
   TIME-TO-REPAIR: $exp(1)$

2. Arcade model of processors' repair unit:

   REPAIR UNIT: $p.rep$
   COMPONENTS: pp, ps
   REPAIR STRATEGY: $FCFS$

3. The Arcade model for the system evaluation criteria is given by the fault tree description of the system failure conditions:

   SYSTEM DOWN:
   $(pp.down \wedge ps.down)$
   $\vee (dc\_1.down \wedge dc\_2.down)$
   $\vee (dc\_3.down \wedge dc\_4.down)$
   $\vee (2of4 \;\; d\_1.down, ..., d\_4.down)$
   $\vee ... \vee ((2of4 \;\; d\_21.down, ..., d\_24.down)$

   $((2of4 \;\; d\_1.down, ..., d\_4.down)$ is a 2-out-of-4 failure among d_1, d_2, d_3, and d_4[10].

5.1.2 Analysis

Using the methodology described in Section 4 we generated the CTMC representing the behavior of the DDS. This CTMC has 2,100 states and 15,120 transitions. During the generation of this model, the largest I/O-IMC encountered had 6,522 states and 33,486 transitions. For comparison, the final model generated in [19] had 16,695 states.

Using the overall CTMC we can analyze the steady-state availability (A) and reliability ($R(t)$) of the distributed database system. Table 1 shows the results of this analysis compared to the SAN-based results in [19]. Note that the reliability results in this table are based on the definition of reliability used in [19], i.e., the probability of having no system failures within a certain mission time assuming that no component is ever repaired. Because of the discrepancy in reliability results we have also verified our results for the DDS system with the DFT tool Galileo [1].[11]

Measure	Arcade	SAN	Galileo
A	0.999997	0.999997	-
R(5 weeks)	0.402018	0.425082	0.402018

Table 1. Dependability analysis for DDS

5.2 Reactor Cooling System

This case study was described in [22, 7]. In [7], the system was modeled using the eFT approach.

The reactor cooling system (RCS) consists of a reactor, two parallel pump lines, a heat exchanger and a bypass system for the heat exchanger. Each of the two pump lines consists of a single pump, a single filter and a number of control valves. The heat exchanging unit consists of the heat exchanger itself, a number of valves and one filter. The

[10] i.e., (*2of4* is shorthand for $(d\_1.down \wedge d\_2.down) \vee (d\_1.down \wedge d\_3.down) \vee ... \vee (d\_3.down \wedge d\_4.down)$

[11] It is possible to use DFTs here because we do not consider repair.

bypass system can be opened and closed by means of two motor driven valves.

All components, except the reactor itself whose failure behavior is not considered here, are subject to failures and are repairable. The filters and the heat exchanger are either operational or failed. The valves can fail in two different modes, either *stuck-open* or *stuck-closed*. The pumps have two different operational modes and one failure mode. They are either fully operational, or in a degraded operational mode, which is reached if one of the two pumps fails. In degraded mode, the remaining pump will fail with a higher failure rate. We refer to this operational mode as *degraded* mode. This is indeed a typical *load sharing* situation.

Except for the two pumps, which share a single repair unit with an FCFS repair strategy, each component has its own dedicated repair unit.

The system is down, if either none of the two pump lines is operational, or both the heat exchanger and the bypass system are not operational. A pump line is defective, if one of its components is defective, where for the valves, only the stuck-closed case is considered to be a relevant failure. The heat exchanging unit is defective if the heat exchanger itself fails or one of its accompanying filters or valves fails. Finally, the bypass line fails if one of the motor driven valves is stuck-closed.

5.2.1 Arcade **model**

Here, we will give the Arcade models for a few of the components of the RCS.

1. The pumps have two operational modes, normal and degraded. For P1, the pump goes to its degraded mode, if P2 fails. The time-to-failure is distributed according to an Erlang-2 distribution, where each phase has rate $5.44 \cdot 10^{-6}$, i.e., we have $erlang(2, 5.44 \cdot 10^{-6})$. If the pump is in degraded mode, the rate doubles, i.e., becomes $10.88 \cdot 10^{-6}$ The time-to-repair is distributed according to an Erlang-2 distribution with rate 0.1.

   COMPONENT: $P1$
   OPERATIONAL MODES: $(normal, degraded)$
   NORMAL-TO-DEGRADED: $P2.down$
   TIME-TO-FAILURE: $erlang(2, 5.44 \cdot 10^{-6})$, $erlang(2, 10.88 \cdot 10^{-6})$
   TIME-TO-REPAIR: $erlang(2, 0.1)$

   The same Arcade model applies for P2.

2. The valves can fail in two different ways: either they are stuck-open or stuck-closed. That means, we have two failure modes, which are equally probable. The overall failure rate is $2 \cdot 4.2 \cdot 10^{-8}$, as each of the failure modes has failure rate $4.2 \cdot 10^{-8}$ according to the eFT specification in [7][12].

   COMPONENT: $VIP1$
   TIME-TO-FAILURE: $exp(8.4 \cdot 10^{-8})$
   FAILURE MODE PROBABILITIES: 0.5, 0.5
   TIME-TO-REPAIR: $exp(0.1), exp(0.1)$

3. The filters can be either free or blocked, where the latter state is the failure case. In Arcade terminology this means to be either "up" or "down"[13].

   COMPONENT: $FP1$
   TIME-TO-FAILURE: $exp(2.19 \cdot 10^{-6})$
   TIME-TO-REPAIR: $exp(0.1)$

4. The heat exchanger can be either up or down, it fails with rate $1.14 \cdot 10^{-6}$. The repair rate is again 0.1.

   COMPONENT: HX
   TIME-TO-FAILURE: $exp(1.14 \cdot 10^{-6})$
   TIME-TO-REPAIR: $exp(0.1)$

In [7] the repair policies are not clearly specified. From the remarks w.r.t. the repair of the pumps, we conclude that there are dedicated repair units for all elements, thus, we will assign to each component its own repair unit, except for the pumps, which have a common repair unit.

1. We only show the RU for valve VIP1, all remaining valves are handled similarly. The RU can handle both failure modes of the valve.

   REPAIR UNIT: $VIP1.rep$
   COMPONENTS: $VIP1$
   REPAIR STRATEGY: $DEDICATED$

2. Both pumps are repaired by a single repair unit.

   REPAIR UNIT: $P.rep$
   COMPONENTS: $P1$, $P2$
   REPAIR STRATEGY: $FCFS$

5.2.2 Analysis

After generating the CTMC models for the pump and the heat exchanger subsystem, we could apply the technique of modularization [7] to compute the reliability and availability of the RCS.

The CTMC for the pump subsystem has 10,404 states and 109,662 transitions. The CTMC for the heat exchanger subsystem (including the bypass) has 240 states and 1,668

[12] The Arcade models for the remaining valvesare similar.
[13] Arcade models for other filters are similar.

transitions. The largest model encountered during generation had 98,056 states and 411,688 transitions. Unfortunately, in [7] no state space size was given, thus no comparison is possible in this case.

For a mission time of for example 50 hours, the system unavailability and unreliability are $6.52100 \cdot 10^{-10}$ and $52.9242 \cdot 10^{-10}$ respectively. Our unavailability results coincide with the results in [7].

6 Summary and conclusions

In this paper we have proposed a new framework for dependability evaluation named Arcade. The framework is based on the formal and compositional I/O-IMC semantics. Moreover, its compositional aggregation technique has shown to be very effective in combating the state space explosion problem during analysis. The Arcade approach is extensible. Furthermore, we envision Arcade as a step towards a design language for large and complex systems. Indeed, the ultimate goal is to integrate Arcade in a design environment based on e.g. AADL or UML. It is important to note that although the syntax of the Arcade language bears resemblance to SAVE, the approaches are truly different. Where in SAVE the actual semantics of the models was hidden in a software program that coded the translation from that syntax to a large (flat) Markov chain, Arcade has a formal semantical model that allows for compositional modeling and state space generation and reduction, as well as facilitates the extension of the modeling language.

As for the future, we plan to work on a further automation of the tool chain, as well as to connect to design approaches based on AADL and UML. Furthermore, where we now use relatively simple fault-tree like expressions to specify system failure (cf. Section 3.5.4), we plan to allow for CSL-type expressions [4], thus querying more complex measures than system reliability or availability.

Acknowledgment: The authors thank Holger Hermanns for his valuable comments on an earlier draft version of this paper. The reviewers are also thanked for their constructive comments that helped to improve the quality of the paper.

References

[1] Galileo tool. http://www.cs.virginia.edu/~ftree.

[2] Architecture Analysis and Design Language (AADL). SAE standards AS5506, Nov 2004.

[3] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets*. Wiley, 1995.

[4] C. Baier, B. Haverkort, H. Hermanns, and J. Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Trans. Software Eng.*, 29(7):1–18, July 2003.

[5] H. Boudali, P. Crouzen, and M. Stoelinga. A compositional semantics for Dynamic Fault Trees in terms of Interactive Markov Chains. *LNCS*, 4762:441–456, 2007.

[6] H. Boudali, P. Crouzen, and M. Stoelinga. Dynamic fault tree analysis using input/output interactive markov chains. In *Proc. of the 37th Annual IEEE/IFIP International Conference on DSN*, pages 708–717. IEEE, 2007.

[7] K. Buchacker. Modeling with extended fault trees. In *5th IEEE Int. Symposium on High Assurance Systems Engineering*, pages 238–246, Nov 2000.

[8] E. de Souza e Silva and R. M. M. Leao. The "TANGRAM-II" environment. In *Computer Performance Evaluation. Modelling Techniques and Tools: 11th Int. Conference, TOOLS 2000*, volume 1786, pages 366–369. LNCS, 2000.

[9] S. Distefano and L. Xing. A new approach to modeling the system reliability: dynamic reliability block diagrams. In *RAMS'06 proceedings*, pages 189–195, 2006.

[10] J. B. Dugan, S. J. Bavuso, and M. A. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Trans. on Reliability*, 41(3):363–377, September 1992.

[11] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: a toolbox for the construction and analysis of ditributed processes. In *Proc. of the 19th International Conference on Computer Aided Verification (CAV)*, 2007.

[12] A. Goyal, W. C. Carter, E. de Souza e Silva, S. S. Lavenberg, and K. S. Trivedi. The system availability estimator. In *Proceedings of the 16th Int. Symp. on Fault-Tolerant Computing*, pages 84–89, July 1986.

[13] H. Hermanns. *Interactive Markov Chains*, volume 2428 of *Lecture Notes in Computer Science*. Springer, 2002.

[14] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPtool. *LNCS*, 1469:51–62, 1998.

[15] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[16] N. Lynch and M. Tuttle. An Introduction to Input/output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[17] OMG Group. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. Technical report, june 2006.

[18] A.-E. Rugina, K. Kanoun, and M. Kaniche. A System Dependability Modeling Framework Using AADL and GSPNs. In R. de Lemos, C. Gacek, and A. B. Romanovsky, editors, *WADS*, volume 4615 of *LNCS*, pages 14–38. Springer, 2006.

[19] W. H. Sanders and L. M. Malhis. Dependability evaluation using composed SAN-based reward models. *Journal of Parallel and Distributed Computing*, 15(3):238–254, 1992.

[20] M. L. Shooman. *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*. John Wiley & Sons, 2002.

[21] M. Walter, M. Siegle, and A. Bode. OpenSESAME: the simple but extensive, structured availability modeling environment. *RESS*, In Press, corrected proof, April 2007.

[22] L.-M. Xing, K. Fleming, and W.-T. Loh. Comparison of Markov model and fault tree approach in determining initiating event frequency for systems with two train configurations. *RESS*, 53(1):17–29, 1996.