Reliability, Risk and Safety – Ale, Papazoglou & Zio (eds)
© 2010 Taylor & Francis Group, London, ISBN 978-0-415-60427-7

Probabilistic verification of architectural software models using SoftArc and Prism

B.R. Haverkort
University of Twente, Enschede, The Netherlands
Embedded Systems Institute, Eindhoven, The Netherlands

M. Kuntz & F. Leitner-Fischer
University of Konstanz, Konstanz, Germany

A. Remke & S. Roolvink
University of Twente, Enschede, The Netherlands

ABSTRACT: In this paper we will describe the SoftArc approach. With the SoftArc approach it is possible to model and analyse safety-critical embedded and distributed systems that consist of both hard- and software. We are going to present the SoftArc modelling language, its syntax and semantics. The semantics of the SoftArc modelling language is defined in terms of stochastic reactive modules. We will show how important measures of interest for probabilistic dependability analysis like availability, unavailability, and survivability, can be analysed. We will demonstrate the feasibility of our approach by means of two case studies, that involve hard- and software elements. First, we are presenting two industrial case studies from the automotive industry. We will analyse the non volatile random access manager (NVRAM) from the AUTOSAR open system architecture, Second, we are going to present the survivability analysis of a simplified version of the Google replicated file system.

1 INTRODUCTION

It is commonplace, that our modern society depends heavily on the correct functionality of soft- and hardware systems. Correct does not only mean, that the system does what it is expected to do, but also that the service the system has to provide is delivered timely (performance) and that the service is available most of time (dependability).

Too often, systems are designed and implemented with performance and dependability requirements only studied as an after-thought. However, it is important to already know in early phases of system design whether pre-set system criteria are met. Using performance and dependability modelling and evaluation techniques the best system design alternative can be identified.

To facilitate the analysis of such requirements throughout the system design trajectory, we advocate the use of formalisms that can be linked—unambiguously—to existing design tools. For that reason, we propose an XML-based modelling language in this paper. It has a precise underlying semantics and, at the same time, can be coupled easily to both design tools and analysis tools (such as Prism). Hence, our approach pairs rigor (details will follow) with applicability and openness with respect to true design tools.

In this paper, we propose a new XML-based modelling language for architectural models called SoftArc (for "Software-oriented Arcade" (Boudali, Crouzen, Haverkort, Kuntz, and Stoelinga 2008)). SoftArc has the following properties: (i) fully XML-based, so that interfacing to design tools and other analysis tools is easily achieved; (ii) an underlying formal semantics in terms of reactive modules (Alur and Henzinger 1999); (iii) integrated support for dependability (reliability and availability) evaluation, performability evaluation (where a notion of costs/rewards is used), and survivability evaluation (to study "performance" after the occurrence of failures); (iv) full support for the inclusion of component repair strategies.

Related work. There is a vast body on literature on approaches, that have been developed for evaluating system dependability, we refer the reader to (Boudali, Crouzen, Haverkort, Kuntz, and Stoelinga 2008) for an extensive overview. In (Boudali, Crouzen, Haverkort, Kuntz, and Stoelinga 2008), we have introduced Arcade on which SoftArc is based. The differences between Arcade and SoftArc can be summarised as follows: (i) We have extended

Arcade's syntax for SoftArc to be able to deal with software architectures, (ii) we have changed the semantic model of SoftArc to stochastic reactive modules (Kwiatkowska, Norman, and Parker 2004), mainly to be able to exploit the extended analysis techniques offered by Prism (Kwiatkowska, Norman, and Parker 2004), (iii) due to the change of the semantic model, SoftArc cannot deal with models that exhibit non-deterministic behaviour. However, the models, we have analysed here, do not exhibit non-determinism. Software reliability is an active area of research over several decades now. For an overview of existing approaches we refer to (Sharma and Trivedi 2007). However, we are not aware of any approach that allows for the combined analysis of dependability and survivability of integrated hard- and software systems on an architectural level, as we suggest in this paper.

Organisation of the paper. We introduce the SoftArc framework, its syntax and semantics in Section 2. In Section 3 the SoftArc framework is applied in the dependability analysis of the NVRAM manager and the Google replicated file system. Finally, Section 4 presents our conclusions.

2 SYNTAX AND SEMANTICS OF SOFTARC

SoftArc is an extension resp. modification of Arcade (Boudali, Crouzen, Haverkort, Kuntz, and Stoelinga 2008; Haverkort, Kuntz, Remke, Roolvink, and Stoelinga 2010), to deal with the evaluation of systems that consist of both soft- and hardware. In (Boudali, Crouzen, Haverkort, Kuntz, and Stoelinga 2008) we defined the formalism Arcade in (Haverkort, Kuntz, Remke, Roolvink, and Stoelinga 2010) we extended Arcade to ExtArc. ExtArc extends Arcade on both the theoretical and the practical side: On the practical side we provide a fully automated tool chain, on the theoretical side, we extended the possibilities of measure specification. To support the practical applicability, we gave to ExtArc a novel semantic model based on probabilistic reactive modules (Alur and Henzinger 1999; Kwiatkowska, Norman, and Parker 2004).

2.1 *Syntax of SoftArc*

An SoftArc consist of two parts:

1. Behavioural description: Here, the basic behaviour of the system is described. SoftArc provides the following ingredients for behavioural descriptions: basic components (BCs), repair units (RUs), spare management units (SMUs), schedulers (SCHED).
2. Measures of interest: Here, we can define some basic measures that can be analysed for the system. We provide means to define a fault tree (FT), i.e., to define under which conditions the system is not operational, we can define service levels, and disasters for checking the system's survivability (Cloth and Haverkort 2005).

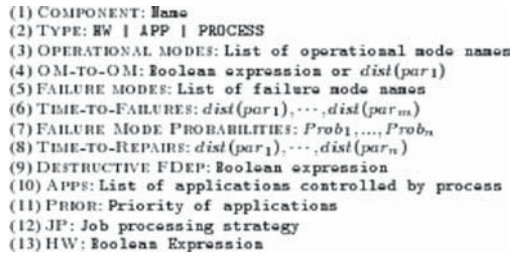
In the sequel, we will, due to space restrictions, describe the syntax of the elements that are relevant for the evaluation of mixed soft- and hardware systems. For the remaining elements we refer to (Boudali, Crouzen, Haverkort, Kuntz, and Stoelinga 2008).

Basic Components. Basic Components (BC) describe the basic behaviour of the components a system consists of. Each component can be in different operational modes (OMs), it can be either *on* or *off*; it can be *accessible* or *inaccessible*; it can be in *normal* or *degraded* OM.[1] If no special OM is given, then we implicitly assume two OMs, namely *up* denoting that the BC is operational and *down* denoting that the system has failed.

A component can have several down modes, which reflect the system failing in different ways. We give names to the different failure modes and can attribute them with a probability and a dedicated failure rate. If a component can only fail in one way, we assign to the failure mode probability 1.

Failure times can be either exponentially or Erlang distributed. Using Erlang distribution, it is possible to approximate more general distributions. In practice, it is also possible that a component fails due to the failure of another component (destructive failure). This is reflected by *FDEP*, where the components whose failures affect the functionality of the component are listed. Finally, we can also indicate the costs that are inflicted when the component is operational (or not). This is useful for the assessment of different repair strategies.

Software runs on hardware, therefore we provide means to specify the hardware components that have to be available (up, on, accessible, etc.) for the software to be executable.



[1] These are predefined operational modes. The user can define more, but would have to provide the semantics in terms of ExtArc to Prism translation procedures.

In line (1) the BC's name is defined, line (3) specifies the OMs defined for a particular BC. In line (2) it is defined whether the BC represents hardware behaviour (HW), applications (APP) or processes (PROCESS). We make this distinction between application and processes, as applications should not access hardware, shared resources etc., directly, but only via a process that controls this access. If the BC can only be *up* or *down*, line (3) can be omitted. Lines (3) and (4) define the operational modes (OM) the BC can be in, and the conditions under which OM switches can occur. In line (4) also define how a mode switch can occur. The mode can switch either if some conditions are satisfied, in this case we use Boolean expression to describe these conditions; alternatively the mode switch can occur after a probabilistically distributed interval of time. Up to now, we support exponential distribution, where *dist* = *exp* and $par_i = \lambda_i$ and Erlang distributions, with *dist* = *erl* and *par* = (λ_i, *n*), where n is the number of exponentially distributed phases with rate λ_i. This line has to be repeated for any operational mode specified in line (3). Line (5) specifies the names of the failure modes. Lines (6) and (7) specify the modes' resp. rates and probabilities. Here, we match the failure modes and their resp. rates and probabilities by the position in the list of line (6). In line (8) the repair times are specified. Each failure mode can be associated with its own specific repair rate. Line (9) lists the conditions under which the BC fails due to a destructive functional dependency. $dist(par_i)$ specifies the probability distributions that are applicable to define the failure and repair times. Lines (10) to (13) are required if the BC is software, i.e., either APP or PROCESS. If it is of type PROCESS, we need a list of application software that is controlled by the current process (line (10)). In line (11) we define the priority of applications controlled by a process. If no specific priority is defined, this line can be omitted. Line (12) defines the job processing strategy. Currently we support first-come-first-served (FCFS) and priority queues. To be executable, software needs hardware, therefore in line (13) we define which hardware components have to be available for software to be executed, this can be done via a Boolean expression.

Repair Units (RUs). RUs repair BCs if they have failed. Each RU is responsible for the repair of a group of BCs, and each BC is repaired by at most one RU. Repair times use a phase-type distribution. Repair takes place according to a predefined strategy; currently, we support: dedicated repair (DED), i.e., the RU is responsible for one component; first-come-first-served (FCFS); and non preemptive priority queuing (NPP).

The syntax of RUs is as follows:

```
(1) RU: Name
(2) COMPONENTS: comp1, comp2,..., compn
(3) STRATEGY: DED | FCFS | NPP | PP
```

Line (2) lists all the BCs the repair unit is responsible for. In Line (3) the repair strategy is defined. We allow for dedicated repair units (DED), simple first-come-first-served (FCFS), non-interrupting priorities (NPP) and interrupting priorities (PP).

Scheduler. In systems with shared resources, but several competing components that require exclusive access to this resource, like multi-process-software that accesses shared memory, we need a scheduler (SCHED), that defines the strategy that is applied to grant exclusive access.

```
(1) SCHED: Name
(2) PROCESSES: List of Processes
(3) POLICY: RR | SJF | FCFS
(4) TIME-SLOT: dist(par_1),...,dist(par_n)
```

In line (1) the name of the scheduler is given. Line (2) specifies the name of the processes that are controlled by that specific scheduler. In line (3) we define the scheduling policy. Currently, we support round robin (RR), shortest job first (SJF), and first-come-first-served (FCFS). In line (4) the time-slots that every job resp. process is granted is specified. Unfortunately, currently, we cannot incorporate hard real time, therefore the time slots have to be either exponentially or Erlang distributed. Where with the latter distribution we can try to approximate hard real time.

Spare Management Unit. For fault-tolerant systems, spares are an important feature. The SMU manages the activation and deactivation of spare components.

The primary component provides the required service, and if it fails, the spare component takes over; there can be one or more spares per primary. If the primary fails, then the (first) spare is deactivated. If the spare also fails, and the next spare is activated; if there is no spare available, then the whole subsystem fails. Spares are deactivated once their primary has been repaired.

```
(1) SMU: Name
(2) COMPONENTS: primary, sp 1,..., sp n
(3) STRATEGY: COLD | WARM | HOT
(4) ACTIVATION-TIME: dist(rate_sp1),...,dist(rate_spn)
```

Here, line (2) defines the primary component and n spare components for that primary. Line (3) specifies the standby strategy which is either cold warm or hot for all spares of the primary. In line (4) the activation times are specified, where *dist* is an exponential or Erlang distribution.

Measure Definition. In SoftArc we can analyse system dependability w.r.t. three basic measure types: reliability, availability and survivability:

1. Reliability: is defined as the probability of having no system failure within a certain mission time, assuming that no component is repaired (Sanders and Malhis 1992).
2. Availability: is defined as the probability of the system being in an operational state within a mission time assuming that components are being repaired (Avizienis, Laprie, Randell, and Landwehr 2004).
3. Survivability: is defined as the ability of a system to recover to predefined service levels in a timely manner after the occurrence of disasters (Cloth and Haverkort 2005).

Fault Trees (FTs) define when the system is down, and are thus used to express a systems availability and reliability. Formally, a FT is a Boolean expression consisting of AND and OR gates; when it evaluates to 1, the system is down. For survivability we need Disaster Trees (DTs) and Service Levels (SL). DTs define when a disaster has occurred SLs define the desired service level of a system. DTs and SLs are also a Boolean expression consisting of AND and OR gates. When they evaluates to 1, the system is in a disaster state resp. at the desired service level.

The conditions are expressed in terms of the possible failure or operational modes a BC can be in. For instance, if a component X has the failure modes with IDs *m1* and *m2*, and *m2* is relevant for the evaluation, then the user writes: *X.down.m2*.

2.2 *Semantics of SoftArc*

Each SoftArc element is translated into a Prism element. The behavioural components (BCs, SMUs, and RUs) are translated into Prism modules (Kwiatkowska, Norman, and Parker 2004). This translation process is totally transparent for the user of SoftArc, who only provides the ingredients discussed in the previous section, then Prism does the analysis automatically.

Prism and SoftArc. A Prism specification consists in general of several *modules*. A module is a program in a guarded command language that consists of *state variables* and *guarded commands* or *transitions*. Each guarded command has the following structure:

$$[\text{action}]\text{guard} \; -> \; rate_1 : variable\_update_1 \; + \; \ldots \; + \; rate_n : variable\_update_n$$

Here action is an action label, guard is a Boolean expression over values of the state variables, which has to be satisfied for the command to be executable, $rate_i$ is the parameter of an exponential distribution that indicates the average delay before the variable update is done. "+" denotes the choice operator and indicates that any of the commands $variable\_update_i$ can be executed. action realizes the synchronization of commands, i.e., their simultaneous execution, over different commands with the same action-label.

The overall system's behaviour is given as the parallel composition of the behaviour of its constituent modules. The semantics of this can be defined in the traditional SOS style (Plotkin 1981; Parker 2002).

Thus, Prism fits very well as semantic foundation for SoftArc: (1) The overall behavioural model of an SoftArc specification can be considered as the parallel composition of its constituent behavioural components, (2) each behavioural component can be translated into a Prism module, (3) components may require synchronization of parts of their behaviour, e.g., a repair unit remains idle, until one of its attached BCs require repair. This can be modelled via synchronizing actions in their resp. Prism module, (4) the failure modes, operational modes etc., of SoftArc behavioural components can be modelled as Prism state variables, the mode changes can be matched by guarded commands. We will show Prism modules in a pictorial way, state variables are attached to the states, guards and action names are attached to transition arrows. The labellings obey the scheme: guard – > action, transition rate.

General Semantics of Basic Components. Each OM in a BC can be in is translated into a single Prism module. For a BC without any special operational mode, i.e., it can be either *up* or *down*, we get the Prism module shown in Fig. 1. To the states, we attached the variables. The BC can fail in m different ways, therefore we have *m* transitions from state, in which the state is up (state variables up = 1, down = 0. Each transition is labelled with an action name fail_mode_*i*, and a failure rate fail_rate_*i*. Taking a transition leads to an update of the state variables, in this case, the up variable is set to 0, the down variable is set to the corresponding mode.

Semantics of application software and processes. For the application software, we have to generate a semantic model also for the operational modes the software can be in. As an example, assume the application requires read and write access to memory. Then, we would obtain the semantic model shown in Fig. 2 As SW can be subject to failures, we implicitly still have the up and down mode. Thus, to correctly reflect the behaviour, we have to combine the two operational modes. This can be obtained by composing the BCs for every operational mode in parallel (cf. Fig. 3, where parallel composition is represented by two strokes "—"). For processes, we would have to add a description

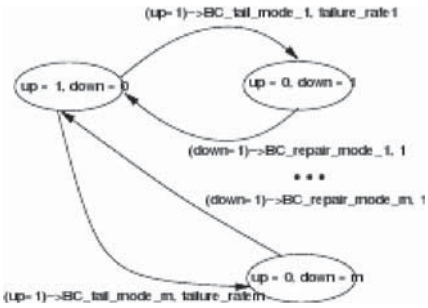


Figure 1. Prism module of a BC.

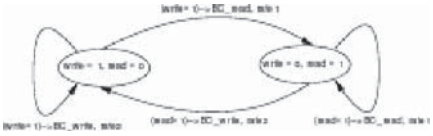


Figure 2. Prism module of an application BC.

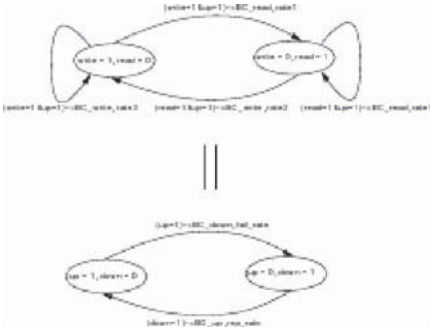


Figure 3. Combined Prism module of application BC.

of the semantics of the job processing strategies. For simplicity, assume, we have a process that is responsible for one application, that sends one type of request to the process, that are processed according to FCFS policy, and the queue has capacity one. The simplified Prism module of the can be found in Fig. 4.

Semantics of schedulers. The Prism semantics of a scheduler following the round robin scheduling strategy is shown in Fig. 5. The scheduler polls all the processes and attributes them the time slot specified with *rate_i*.

3 CASE STUDIES

Here, we will show the feasibility of our approach by applying it to two software-related case studies, the NVRAM manager (AUTOSAR a) and a simplified version of the Google replicated file system

Figure 4. FCFS processing semantics of process BC.

Figure 5. Prism module for RR scheduling strategy.

(Ghemawat, Gobioff, and Leung 2003; Cloth and Haverkort 2005).

3.1 *The NVRAM manager*

System Description. For this case study, we analyse the *non volatile random access manager* (NVRAM) from the AUTOSAR automotive open system architecture (AUTOSAR a). AUTOSAR has been defined by leading companies from the automotive industry to increase the modularity and reusability of software modules. AUTOSAR promotes the usage of ready-to-use, well tested commercial and off-the-shelf software, to reduce the costs of developing novel software systems in the realm of automotive applications.

The NVRAM module (AUTOSAR b) provides the central memory management of AUTOSAR control units. The application has only access to the random access memory (RAM). Upon request from the application the NVRAM Manager copies data from read only memory (ROM) to RAM, RAM to non volatile memory (NVM) or vice versa. The NVRAM Manager provides services to ensure the data storage and maintenance of non volatile data according to the individual requirements in an automotive environment. An example for such a requirement is to handle concurrent access to the memory. The main reason why we choose the NVRAM Manager was its independence from the application. Each automotive application has to control memory, that is why the NVRAM module will be found in almost every AUTOSAR control unit regardless whether this is an airbag control unit or an electronic stability program control unit.

The interface to the NVRAM Manager is provided through the API and callback functions. Neither the NVRAM module nor the layers above the NVRAM Manager have direct access to the non volatile memory. All access from application to the memory is routed through the AUTOSAR runtime environment (RTE) to the NVRAM module. NVRAM calls an API function of the memory hardware abstraction, where the call is forwarded either to the COM driver (external memory) or to the memory drivers (internal memory). The job processing is controlled by a cyclic operating system task, which can be concurrent to the API calls of the application. The NVRAM module consists of two parts, one for the job scheduling and one for the data handling. The central function of the NVRAM Manager is the job scheduling, the data management part is just needed to store information about the configuration of the several memory blocks. In Fig. 6 we find the schematic architecture of the NVRAM module. Thus, the NVRAM consists of the following modules:

- Job Manager: The job manager handles the job requests sent from the application. There are two types of requests: read requests and write requests. The requests are stored in first-come-first-served (FCFS) queues, one for each type of requests. Write requests have priority over read requests.
- Memory Modules: Each memory module consists of one non-volatile memory (NVM) block, one RAM block and one ROM block. Each of these memory blocks is subject to failures.
- Operating System: The operating system initiates the processing of requests stored by the job manager. We can assume that this process is activated in a cyclic fashion, i.e., we assume round robbin (RR) process scheduling policy.
- Application: The application sends read and write requests to the job manager. These requests can also be sent, if the job manager is not activated by the operating system.

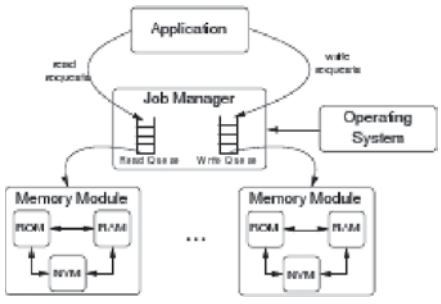


Figure 6. The NVRAM module architecture.

SoftArc model of the NVRAM module. In Fig. 6 the basic architecture of the NVRAM module is shown. For the SoftArc model we need BCs for: (1) application, (2) job manager, (3) operating system, (4) the RAM, ROM, and NVM modules. Additionally, we need repair units for the RAM and the NVM modules. According to the description of the NVRAM, the application requires read or write access to the NVM modules or read access to the ROM modules. We assume, that the mode switch from *read* to *write* occurs after an exponentially distributed amount of time. This basic software model yields the following BC:

(1) COMPONENT: Application
(2) TYPE: APP
(3) OPERATIONAL MODES: read,write
(4) READ-TO-WRITE: $exp(write\_rate)$
(5) READ-TO-READ: $exp(read\_rate)$
(6) WRITE-TO-READ: $exp(read\_rate)$
(7) WRITE-TO-WRITE: $exp(write\_rate)$

The job manager will be modelled as a process. It controls only the application Application, the write jobs have priority over read jobs. The job manager needs access to the RAM to be executable.

(1) COMPONENT: Job_Manager
(2) TYPE: PROCESS
(3) APPS: Application
(4) PRIOR: Application.read, Application.write
(5) JP: FCFS-PRIO
(6) HW: RAM

In this case, the scheduler is the operating system, as the only interesting task of the operating system is to schedule the process Job Manager. We assume an RR strategy and Job Manager is the only process the scheduler is responsible for:

(1) SCHED: Operating_System
(2) PROCESSES: Job_Manager
(3) POLICY: RR
(4) TIME-SLOT: $exp(\gamma)$

For the NVRAM module we have an arbitrary number of memory modules, where each module consists of a RAM, a ROM and an NVM block. The contents of RAM and NVM can be valid or invalid. This can be considered as the two operational modes of the RAM and the NVM. Switching between the two modes obeys the following rules: From valid to invalid a switch occurs with an exponentially distributed rate. From invalid to valid the switch occurs also with an exponentially distributed rate, but additionally the contents of the NVM must be valid for the RAM to switch back, resp. the ROM must not be failed if the NVM requires refreshing its contents. Both RAM and NVM can also fail with an exponentially distributed rate. All three blocks can also fail totally, and

can then not be repaired. The contents of the ROM are considered to be always valid, but the ROM is also subject to failures. For RAM resp. NVM we obtain the following SoftArc specification:

(1) COMPONENT: RAM
(2) TYPE: HW
(3) OPERATIONAL MODES: valid, invalid
(4) VALID-TO-INVALID: $exp(inval\_rate\_RAM)$
(5) INVALID-TO-VALID: $exp(val\_rate\_RAM)$
(6) INVALID-TO-VALID: NVM.valid
(7) TIME TO FAILURES: $exp(failure\_RAM)$

(1) COMPONENT: NVM
(2) TYPE: HW
(3) OPERATIONAL MODES: valid, invalid
(4) VALID-TO-INVALID: $exp(inval\_rate\_NVM)$
(5) INVALID-TO-VALID: $exp(val\_rate\_NVM)$
(6) INVALID-TO-VALID: ROM.up
(7) TIME TO FAILURES: $exp(failure\_NVM)$

Measures of interest. We want to analyse the NVRAM manager with respect to its reliability, time point availability and survivability. To this end, we have to define, under which conditions the NVRAM manager can be considered to be failed (reliability, availability) resp. what we consider to be disasters and service levels (survivability).

We consider the NVRAM module to be failed, if it shows any kind of erroneous behaviour. Here, we restrict ourselves to the following cases: (i) the RAM or the NVM memory module failed, or (ii) in case of a ROM failure, (iii) occurrence of buffer overflow in the job manager. In term of SoftArc the first scenario can be expressed as follows:

For the first scenario, we analyse both reliability over varying time bounds and steady-state availability, for the case of the ROM failure, we only have to consider reliability. The ROM is not considered to be repairable, therefore its steady-state availability is zero. In Table 1, we find the results of reliability analysis. The steady-state availability is 0.99999.

For the third scenario, we compute the reliability for varying time bounds and varying buffer sizes N. In Table 2 we can find the results of the analysis.

For a buffer size of $N = 50$, the reliability approximates 1.

For survivability, the results are shown in Table 3.

We restrict ourselves to the disaster scenario, where the NVM or the RAM failed and the desired service level is defined as the state, where all memory modules contain valid data. In terms of SoftArc this reads as follows:

(1) DT: RAM.invalid | NVM.invalid
(2) SL: RAM.valid & NVM.valid & ROM.valid

Table 1. Reliability analysis for memory failures.

Reliability	t = 10	t = 20	t = 50	t = 100
NVM/RAM failure	0.9992	0.9982	0.9955	0.9911
ROM failure	0.00027	0.00059	0.00149	0.00299

Table 2. Reliability analysis for buffer overflows.

Reliability	t = 10	t = 20	t = 50	t = 100
N = 10	0.9978	0.9725	0.8755	0.7342
N = 20	≈ 1	0.9999	0.9965	0.9872
N = 50	≈ 1	≈ 1	≈ 1	≈ 1

Table 3. Survivability analysis for NVRAM manager.

Survivability	t = 10	t = 20	t = 50	t = 100
	0.1527	0.4008	0.8557	0.9911

3.2 *Replicated file system*

System description. In this case study, originally published in (Cloth and Haverkort 2005), we will analyse the reliability, availability, and survivability of a simplified version of a replicated file system, inspired by the Google file system (GFS) (Ghemawat, Gobioff, and Leung 2003).

In the GFS, files are divided into *chunks*, evenly sized portions of files. A chunk is usually 64 MB large. The GFS consists of one master server, and a number M of chunk servers. Chunk servers store three replica of each of these chunks.

A read-request from a client is handled in the following way: The client accesses the master server, the master-server maintains a hash table of all replicas of a chunk. Then, the client is referred to the closest chunk server and either reads the desired replica or initiates an update of all replicas. The data transfer does not involve the master.

Both the master server and the chunk servers are subject to failures, here, we distinguish between hardware and software failures. In the case of a chunk server failure, we additionally distinguish between destructive and nondestructive failures. In the case of a destructive failure a file chunk is destroyed and has to be replicated by the master server, in the case of a nondestructive failure, the replicas remain intact.

SoftArc specification. The SoftArc specification of the GFS consists of BCs for the master server, the chunk servers, and the replicas.

For the master server and the chunk servers, we have to define RUs, as both can be repaired.

For chunk servers, we assume a single RU, that is operated according to a simple first-come-first-served (FCFS) repair strategy, the maser server has its own repair unit.

The master server can fail in two ways: either due to a software or due to a hardware failure, i.e., there are two failure modes in its BC description.

(1) COMPONENT: `Master`
(2) TYPE: `HW`
(3) FAILURE MODES: `down_soft, down_hard`
(4) FAILURE MODE PROBABILITIES: `0.95, 0.05`
(5) TIME-TO-FAILURES: $exp(f\_M\_soft), exp(f\_M\_hard)$
(6) TIME-TO-REPAIRS: $exp(r\_M\_soft), exp(r\_M\_soft)$

The resp. probabilities can be found in line (4), and the corresponding failure rates are listed in line (5) of the BC description.

The chunk server has four failure modes, additionally to the soft- and hardware failures, both failures can be destructive, i.e., destroying one of the chunk's replicas or non-destructive, leaving all replicas intact.

(1) COMPONENT: `Chunk_i`
(2) TYPE: `HW`
(3) FAILURE MODES: `down_soft_d, down_hard_d down_soft_nd, down_hard_nd`
(4) FAILURE MODE PROBABILITIES: `0.95, 0.05`
(5) TIME-TO-FAILURES: $exp(f\_CS\_soft\_d), exp(f\_CS\_hard\_d)$ $exp(f\_CS\_soft\_nd), exp(f\_CS\_hard\_nd)$
(6) TIME-TO-REPAIRS: $exp(r\_CS\_soft\_d), exp(r\_CS\_hard\_d)$ $exp(r\_CS\_soft\_nd), exp(r\_CS\_hard\_nd)$

The replicas are a software component, and have the following SoftArc specification:

(1) COMPONENT: `Replica`
(2) TYPE: `SW`
(3) OPERATIONAL MODES: `present, lost`
(4) PRESENT-TO-LOST: `Chunk.down_soft_d | Chunk.down_hard_d`
(5) LOST-TO-PRESENT: `Master.up`
(6) LOST-TO-PRESENT: $exp(replica\_rate)$

Replicas can be either present or lost (line (3)), if a chunk server fails in a destructive way, the replica will get lost, mode switch present to lost (line (4)). Replicas will be restored, i.e., switch to mode present, if the master server is up (line (5)) at rate *exp(replica_rate)* (line (6)).

(1) RU: `CS_RU`
(2) COMPONENTS: `CS_1,...,CS_n`
(3) STRATEGY: `FCFS`

The repair unit *CS_RU* is responsible for all chunk servers (line (2)), and repairs them according to an FCFS repair strategy (line (3)).

Measures of interest. Also the GFS will be analysed with respect to its reliability, availability and survivability. We consider the GFS to be failed, if either the master server is down or if more than 25% of all M chunk servers are down. Here, we set $M = 80$, there- fore, we have 80 chunk servers. We obtain the following FT in SoftArc notation:

(1) FT: `Master.down_soft | Master.down_hard | 20oo80(Chunk_i.down_soft, Chunk_i.down_hard)`

20oo80 is a shorthand for the condition, that 20 out of 80 chunk servers have to be failed. Again, we compute the system's reliability for varying time bounds and its steady-state availability. The results for the reliability evaluation can be found in Table 4. The steady-state availability of GFS computes to 0.9999.

For survivability analysis we consider the following disaster: 40 chunkservers are down, due to a software failure, and no chunk server failed due to a hardware failure. The service level we want to reach is: the master is up, 50 chunk servers are up and all three replicas are present. In Table 5 we can find the results of the survivability analysis.

Table 4. Reliability analysis results for GFS.

Reliability	t = 10	t = 20	t = 50	t = 100
	0.9951	0.9901	0.9753	0.9497

Table 5. Survivability analysis results for GFS.

Survivability	t = 10	t = 20	t = 50	t = 100
	0.5926	0.8447	0.9914	0.9999

4 CONCLUSIONS

In this paper we have presented SoftArc, an extension of Arcade, that is capable of dealing with the dependability analysis of software architectures. To this end, we have introduced language elements that are capable of dealing with software, such as processes, process schedulers and programmes. We have provided a fully formal semantics of SoftArc in terms of stochastic reactive modules. The feasibility of our approach was shown by two software-related case studies, the NVRAM manager and the Google replicated file system. We have analysed reliability, availability and survivability measures for both case studies, using Prism as back end tool, to compute the actual numerical values of these measures.

In the future, we plan to add to SoftArc a graphical user interface to ease the specification of dependability models. It is also planned to further extend the possibilities to exploit model symmetries to obtain smaller semantical models.

REFERENCES

Alur, R. and T. Henzinger (1999). Reactive Modules. Formal Methods in System Design 15(1), 7–48.

AUTOSAR a. www.autosar.org, checked 16.12.2009.

AUTOSAR b. Requirements on Memory Services v. 2.2.1.

Avizienis, A., J.-C. Laprie, B. Randell and C. Landwehr (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1, 11–33.

Boudali, H., P. Crouzen, B.R. Haverkort, M. Kuntz and M.I.A. Stoelinga (2008). Architectural Dependability Evaluation with Arcade. In Proc. of DSN 2008, pp. 512–521.

Cloth, L. and B. Haverkort (2005). Model Checking for Survivability! In Proc. of QEST 2005, pp. 145–154.

Ghemawat, S., H. Gobioff and S.-T. Leung (2003). The Google File System. In Proc. of SOPS'03, pp. 29–43. ACM.

Haverkort, B., M. Kuntz, A. Remke, S. Roolvink and M. Stoelinga (2010). Performability Evaluation using Architectural Models Applied to a Water-Treatment Facility. In submitted for publication, under review.

Kwiatkowska, M., G. Norman and D. Parker (2004). Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. International Journal on Software Tools for Technology Transfer (STTT) 6(2), 128–142.

Parker, D. (2002). Implementation of Symbolic Model Checking for Probabilistic Systems. Ph.D. thesis, School of Computer Science, Faculty of Science, University of Birmingham.

Plotkin, G. (1981, September). A Structural Approach to Operational Semantics. technical report, Computer Science Department FN-19, DAIMI, Aarhus University.

Sanders, W.H. and L.M. Malhis (1992). Dependability Evaluation Using Composed SAN-Based Reward Models. Journal of Parallel and Distributed Computing 15, 238–254.

Sharma, V.S. and K.S. Trivedi (2007). Quantifying Software Performance, Reliability and Security: An Architecture-based Approach. The Journal of Systems and Software 80, 493–509.