Software Qual J
DOI 10.1007/s11219-016-9315-9

Availability analysis of software architecture decomposition alternatives for local recovery

Hasan Sözer[1] · Mariëlle Stoelinga[2] · Hichem Boudali[3] · Mehmet Akşit[4]

Abstract We present an efficient and easy-to-use methodology to predict—at design time—the availability of systems that support local recovery. Our analysis techniques work at the architectural level, where the software designer simply inputs the software modules' decomposition annotated with failure and repair rates. From this decomposition, we automatically generate an analytical model (a continuous-time Markov chain), from which an availability measure is then computed, in a completely automated way. A crucial step is the use of intermediate models in the input/output interactive Markov chain formalism, which makes our techniques efficient, mathematically rigorous, and easy to adapt. In particular, we use aggressive minimization techniques to keep the size of the generated state spaces small. We have applied our methodology on a realistic case study, namely the MPlayer open-source software. We have investigated four different decomposition alternatives and compared our analytical results with the measured availability on a running MPlayer. We found that our predicted results closely match the measured ones.

Keywords Dependability · Availability · Fault tolerance · Local recovery · Software architecture evaluation

✉ Hasan Sözer
hasan.sozer@ozyegin.edu.tr

[1] School of Engineering, Ozyegin University, Nişantepe Mah. Orman Sk. No: 34-36, Alemdağ – Çekmeköy, 34794 Istanbul, Turkey

[2] Formal Methods and Tools Group, Department of Computer Science, University of Twente, Enschede, The Netherlands

[3] European Space Research and Technology Centre, European Space Agency, Noordwijk, The Netherlands

[4] Software Engineering Group, Department of Computer Science, University of Twente, Enschede, The Netherlands

Published online: 03 May 2016

1 Introduction

Local recovery (Sozer and Tekinerdogan 2008; Sozer et al. 2009; Candea et al. 2004b) is an important technique to achieve fault tolerance. Whereas a global recovery strategy restarts the whole system upon detection of an error, making the entire system unavailable until its normal operational mode is reached again, local recovery strategies work at a lower level of granularity. They partition the system into several recoverable units (RUs) so that each RU consists of a number of software modules, and each RU can be recovered independently (Sozer et al. 2009). The aim is to achieve better availability: Recovering a part of the system is usually faster than recovering the whole system and, moreover, the non-affected system parts remain operational.

The availability of the system, that is, the percentage of time the system is up and running (Avizienis et al. 2004), heavily depends on the chosen software decomposition alternative, i.e., the way in which the software modules are grouped into RUs (Sozer et al. 2013). We have previously introduced a framework, called FLORA (Sozer et al. 2009) that supports the implementation of local recovery for a particular software decomposition. Despite the support of this framework, the implementation of local recovery for a decomposition alternative is still a time-consuming and a non-trivial task (Sozer et al. 2009), Hence, it is important to follow a systematic method to compare various alternatives and only implement the best one. We previously developed a tool named Recovery Designer (Sozer et al. 2013; Sozer 2009), which optimizes software architecture decomposition for local recovery based on quality estimations for decomposition alternatives. However, optimization needs input from a systematic and sound method that provides availability estimations regarding decomposition alternatives at design time.

This paper presents such a method: We take as input a decomposition of the software architecture including (1) the set of modules; (2) failure and repair rates for each module; and (3) the grouping of modules into RUs. Then our tool generates an analytical model—a continuous-time Markov chain (CTMC)—in a completely automated way. We then use standard CTMC analysis methods to predict the system availability. There exist many studies on assessing non-functional requirements (e.g., performance, availability, reliability) at the software architecture design level (Dugan and Lyu 1995; Das and Woodside 1998; Franco et al. 2012, 2014). Some of these techniques (Lai et al. 2002; Vaidyanathan and Trivedi 2005) also employ Markov models. However, existing studies mainly aim at facilitating general quality analysis of software systems. On the other hand, we particularly focus on the impact of architectural decomposition alternatives of a system for local recovery. Hereby, we apply availability analysis using a modular and compositional approach. Modular design and composition is well-established and applied approach for software design. A software system is usually (supposed to be) decomposed into loosely coupled modular units each of which is responsible for a set of cohesive tasks. This improves the maintainability of the software since changes can be better localized in the system. This principle is less often applied for analytical models to evaluate software architectures. Usually, a single model (e.g., a Markov chain, a queuing network) is built or generated for the whole system at once. A modification of the architecture can lead to changes in the monolithic analytical model at several places or the whole model must be rebuilt/generated from scratch.

In our approach, separate analytical models are generated for different architectural elements. These models are composed together based on how the architectural elements are composed with each other. Different architectural configurations can be evaluated by just using (generating) a different composition script working on the same set of models. If failure behavior of a module or the recovery strategy changes, such changes should only be reflected to the corresponding analytical model that is defined separately from the other models. A CTMC model for the overall system is generated in multiple steps. First, a set of input/output interactive Markov chains (I/O-IMCs) (Hermanns 2002; Boudali et al. 2008) are generated. I/O-IMCs augment traditional CTMCs with discrete actions, thus enabling synchronization between them. They have been used successfully to analyze a wide range of applications (Hermanns and Katoen 2000; Boudali et al. 2007b, 2008) and enable powerful analysis methods. In particular, we exploit their *compositional aggregation* to avoid state-space explosion. As the underlying formalism, we use the MIOA-syntax (Kuntz and Haverkort 2008) to conveniently specify and generate: (1) one I/O-IMC for each software module contained in an RU, modeling the failure and recovery behavior of that module; (2) two I/O-IMCs for each RU, serving as interfaces between the RU and the recovery manager (RM);[1] and (3) one I/O-IMC corresponding to the RM. By composing all the generated I/O-IMCs, we obtain a CTMC that can be then analyzed. However, to reduce the size of the generated state space, we incrementally compose one by one the I/O-IMC models and reduce the intermediate state space [by applying bisimulation minimization (Boudali et al. 2007a)] after each composition. This is precisely the compositional-aggregation technique mentioned before.

We have applied our modeling and analysis approach on a real-life software system, namely the MPlayer open-source media player. We have investigated four different decomposition alternatives and compared the availability predicted by our analytical models to the availability measurements obtained from the actual implementations. It turned out that our analytical results closely match the measured availabilities.

The contributions of this paper are the following: (1) a method to analyze the availability of local recovery architectures, relying on a (novel) translation of a local recovery architecture to a set of I/O-IMC models, and the (existing) compositional-aggregation method; (2) automated generation of scripts for composing I/O-IMC models according to the provided architecture and for performing availability analysis on the composed models; (3) experimental validation of our results, by comparing predicted and measured availability for a real-life application.

This work arose from the need to efficiently, easily, and automatically conduct quantitative analysis of the availability of various module decomposition alternatives in the context of a software recovery mechanism. Our solution based on the I/O-IMC formalism, as described in this paper, fulfilled this need.

Organization of the paper We summarize the related previous studies in Sect. 2. In Sect. 3, we introduce the local recovery concept and the background context of this work. In Sect. 4, we present our modeling approach including the detailed I/O-IMC models used for modeling local recovery. In Sect. 5, we present a case study along with experimentation results. In Sect. 6, we discuss assumptions, limitations, and the utilization of results. Finally, we conclude the paper in Sect. 7.

[1] An important component used within a software architecture that supports local recovery.

2 Related work

There are several modeling techniques to analyze system dependability and performance. These techniques mostly rely on state-based models (Lai et al. 2002), queuing networks (Das and Woodside 1998), fault trees (Dugan and Lyu 1995), or a combination of these. For instance, Lai et al. (2002) present a Markov model to compute the availability of a redundant distributed hardware/software system comprised of *N* hosts; Vaidyanathan and Trivedi (2005) present a three-state semi-Markov model to analyze the impact of rejuvenation[2] on software availability. In general, however, these models are specified manually and/or the methodology lacks a comprehensive tool support, making them less practical to use. In this work, we provide tool support to carry out such modeling/analysis, and we employ a flexible modeling approach to easily accommodate different architectural decompositions.

There have also been approaches aiming at automation of modeling and analysis (Immonen and Niemel 2008). These approaches hide the underlying formalism from the user and utilize architectural descriptions that are annotated with dependability and performance properties. Different formal models are automatically generated based on these descriptions specified with a variety of notations and languages. Franco et al. employ software architecture descriptions in Acme architecture description language (ADL) (Garlan et al. 1997) to estimate reliability (Franco et al. 2012) and availability (Franco et al. 2014). Markov models are generated based on these descriptions automatically. The authors in Majzik and Huszerl (2002) use stochastic Petri nets to model and analyze fault-tolerant CORBA applications that are modeled using UML. Hereby, the UML profile is extended to be able to incorporate the dependability properties that are required to generate Petri net models. Another work (Bernardi et al. 2011) focusing on dependability analysis extends the MARTE profile, which itself is an extension of UML for quantitative analysis of schedulability and performance. There exist many other dependability modeling and analysis techniques introduced for software systems specified with UML (Bernardi et al. 2012). Similarly, there exist other contributions (Rugina et al. 2007; Bozzano et al. 2011) using AADL as an ADL. The error model annex of AADL supports dependability analysis (Joyce 2007). Component models and models of the execution environment are also utilized as input models (Brosch et al. 2012). These studies mainly focus on general dependability analysis of software systems. Our goal in this work is not dependability analysis in general, but to be able to evaluate *architectural decomposition alternatives* of a system for local recovery in particular. We utilize architecture specifications in xADL (Dashofy et al. 2002) as input; however, any other ADL or notation can be utilized in principle.

There have been models, techniques, and tools developed for evaluating the reliability and availability of fault-tolerant systems (Geist and Trivedi 1990; Dugan and Lyu 1995; Bowles et al. 2004). However, these approaches do not consider the decomposition of an existing architecture for local recovery. They mainly focus on fault-tolerant design that is achieved with a number redundant subsystems. As far as local recovery strategies are concerned, the work of Candea et al. (2004a) improves system availability with local recovery techniques that are similar to ours. In that work, the term *micro-reboot* is used for recovering individual components of the system by restarting them, while the other components remain operational. As such, we can use the terms *micro-reboot* and *local*

[2] Proactively restarting a software component to mitigate its aging and thus its failure.

recovery interchangeably. We also basically reboot a modular unit of the system for recovering it locally. The only difference in terms of implementation is that the framework that we employ can restore the last check-pointed state after reboot. In Candea's work, components are assumed to be stateless. However, the main contribution of this paper that makes it differ from Candea's work is the evaluation of a decomposition alternative at design time, before implementation. In Candea's work, decomposition alternatives are selected with heuristic rules and they are evaluated experimentally after implementation (Candea et al. 2004a). In this paper, we introduce analytical models for the evaluation of decomposition alternatives at design time. To the best of our knowledge, this approach is novel, which makes it possible to obtain an evaluation before a costly implementation of local recovery or micro-reboot.

Previously, we introduced *Recovery Designer* (Sozer et al. 2013) for automatically finding the optimal decomposition of software architecture for local recovery. This tool addresses a set of system constraints and balances the achieved availability and performance overhead for decomposition alternatives. Hereby, the performance overhead is estimated by models derived from the source code with dynamic analysis. However, availability estimation was based on a simple heuristic objective function (Sozer et al. 2013). In this work, we incorporated model-based availability analysis for quantitative availability assessment of decomposition alternatives. In the following, we introduce the context of our work together with basic concepts and terminology we use regarding local recovery.

3 Local recovery and Recovery Designer

Recovery of errors is an essential step of fault tolerance (Avizienis et al. 2004). Local recovery is an effective approach for recovering from errors, in which the erroneous parts of a system are recovered while the other parts of the system are kept in operation. Introducing local recovery to a system imposes certain requirements to its design.

- *Isolation*: If an operational module tries to access a module that has failed or is under recovery, then errors propagate from the failed module to the operational one. To prevent this propagation, the system should be separated into a set of *Recoverable Units (RUs)* with clear boundaries and isolation between them.
- *Communication Control*: Although an RU is unavailable during its recovery, other RUs might still need to access it in the meantime. Therefore, the communication between RUs must be mediated and controlled (e.g., through blocking, queuing, and retrying of messages), so that the recovery of an RU is transparent to the other RUs. In Candea et al. (2004b), for instance, the communication is mediated by an application server. As a result, there is a need for a *Communication Manager (CM)* that mediates inter-RU communication.
- *System-Recovery Coordination*: In case recovery actions take place while the system is still operational, interference with the normal system functions is inevitable and the required recovery actions need to be coordinated. For this reason, there is a need for a *Recovery Manager (RM)* that controls and coordinates RUs for recovery.

Note that there can be different implementations of local recovery: the isolation between the different RUs can be achieved by running them on separate processes or different Java components (Hunt 2007; Candea et al. 2004b; Sozer et al. 2013); the RM and CM can be composed of multiple components or they can all be implemented in a single component; RUs can be assumed to be stateless (Candea et al. 2004b) or they can be equipped with special state stores (Sozer et al. 2009). The specific implementation is, however, not relevant for our methodology to estimate system availability. We have previously developed a framework, namely FLORA (Sozer et al. 2009) that supports the decomposition and implementation of software architecture for local recovery. We shortly introduce this framework in the following.

3.1 The FLORA framework

FLORA (Sozer et al. 2009) partitions system modules as defined by RUs and isolates these modules by assigning each RU to a separate process.[3] In addition to the specified RUs, FLORA introduces a CM[4] and a RM. The CM mediates all inter-RU communication and employs a set of communication policies (e.g., drop, queue, retry messages). The RM can detect fatal errors and can restart dead RUs.

The total system availability depends on the availability of its individual modules and the RUs' decomposition. Generally speaking, the module availability depends on its *mean time to failure (MTTF)*, i.e., the time it takes on average before a module fails, and its *mean time to repair (MTTR)*, the average time it takes for the module to restart.

Increasing the number of RUs can provide higher availability. However, this will also introduce an additional performance overhead since more modules will be isolated from each other. Here, we are considering the performance overhead caused by communication between processes and marshaling. FLORA captures all function calls across different RUs and redirects them through inter-process communication calls. Therefore, we should consider the amount of interactions between the chosen RU boundaries to keep the performance overhead low. On the other hand, keeping the modules together in one RU will increase the performance, but will result in a lower availability since the failure of one module will affect the others as well. As a result, for selecting a decomposition alternative we have to cope with a trade-off between availability and performance. *Recovery Designer* (Sozer et al. 2013) supports this trade-off decision as explained in the following.

3.2 Recovery Designer

Recovery Designer (Sozer et al. 2013) is a tool that is used for optimizing software architecture decomposition for local recovery. It supports the following activities: (1) depict the design space of the possible decomposition alternatives, (2) take as input a set of decomposition constraints and reduce the design space accordingly, (3) take as input estimations of performance and availability for the remaining feasible decomposition alternatives, and (4) balance the feasible alternatives with respect to availability and performance by utilizing optimization techniques for evaluating large design spaces.

The tool is developed as an extension of the ArchStudio environment (Dashofy et al. 2002), which works as an Eclipse plugin (ECLIPSE 2015). Figure 1 depicts the main components of this toolset and the overall process. First, the user provides an annotated software architecture description. Annotations include decomposition constraints, i.e., which modules must (not) be separated, and the failure/repair rates of the modules. A tool eliminates infeasible alternatives and generates a design space (1), which includes a set of

[3] Interaction between the RUs is redirected through Inter-Process Communication.

[4] Modeled as part of the RM as mentioned in Sect. 4.

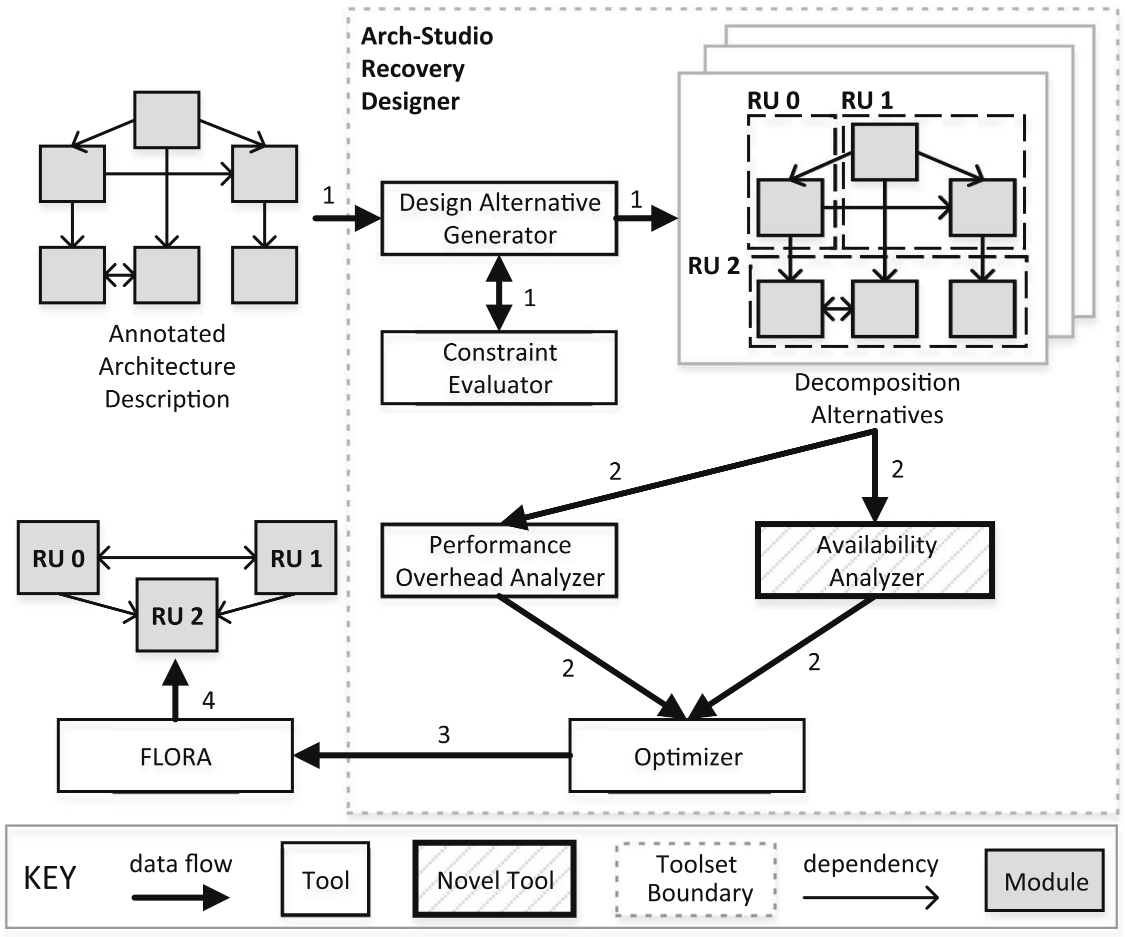

Fig. 1 The overall process and the integrated toolset

decomposition alternatives (Sozer et al. 2013). Each of these alternatives is evaluated with respect to two quality attributes: availability and performance (2). We have previously implemented *Performance Overhead Analyzer*, which estimates the performance overhead imposed by a given decomposition (Sozer et al. 2013; Sozer 2009). In this paper, we introduce *Availability Analyzer*, which provides an availability estimation for a given decomposition alternative based on analytical models. Once performance and availability estimations are known regarding each decomposition alternative, *Optimizer* (Sozer et al. 2013) can select the best alternative (3). The selected alternative is then implemented (4) with the FLORA framework (Sozer et al. 2009).

In the rest of this paper, we focus on the *Availability Analyzer* tool, which is our main contribution here. This tool is implemented in Java, and it makes use of the CADP toolset (Garavel et al. 2007). Figure 2 depicts the intermediate artifacts and data flow during the analysis process. *Availability Analyzer* takes a decomposition alternative and a set of predefined MIOA specifications as input. MIOA specifications formally define the failure and recovery behavior of system modules. First, *Availability Analyzer* generates a set of I/O-IMC models corresponding to the given decomposition alternative. It also generates a composition script, which specifies how these models must be combined. Then, this script is executed with the CADP toolset to obtain a single CTMC model by composing all the I/O-IMC models (2). Finally, the CTMC model is used for providing an availability estimation (3). Our modeling approach and the details of this process are described in the following section.

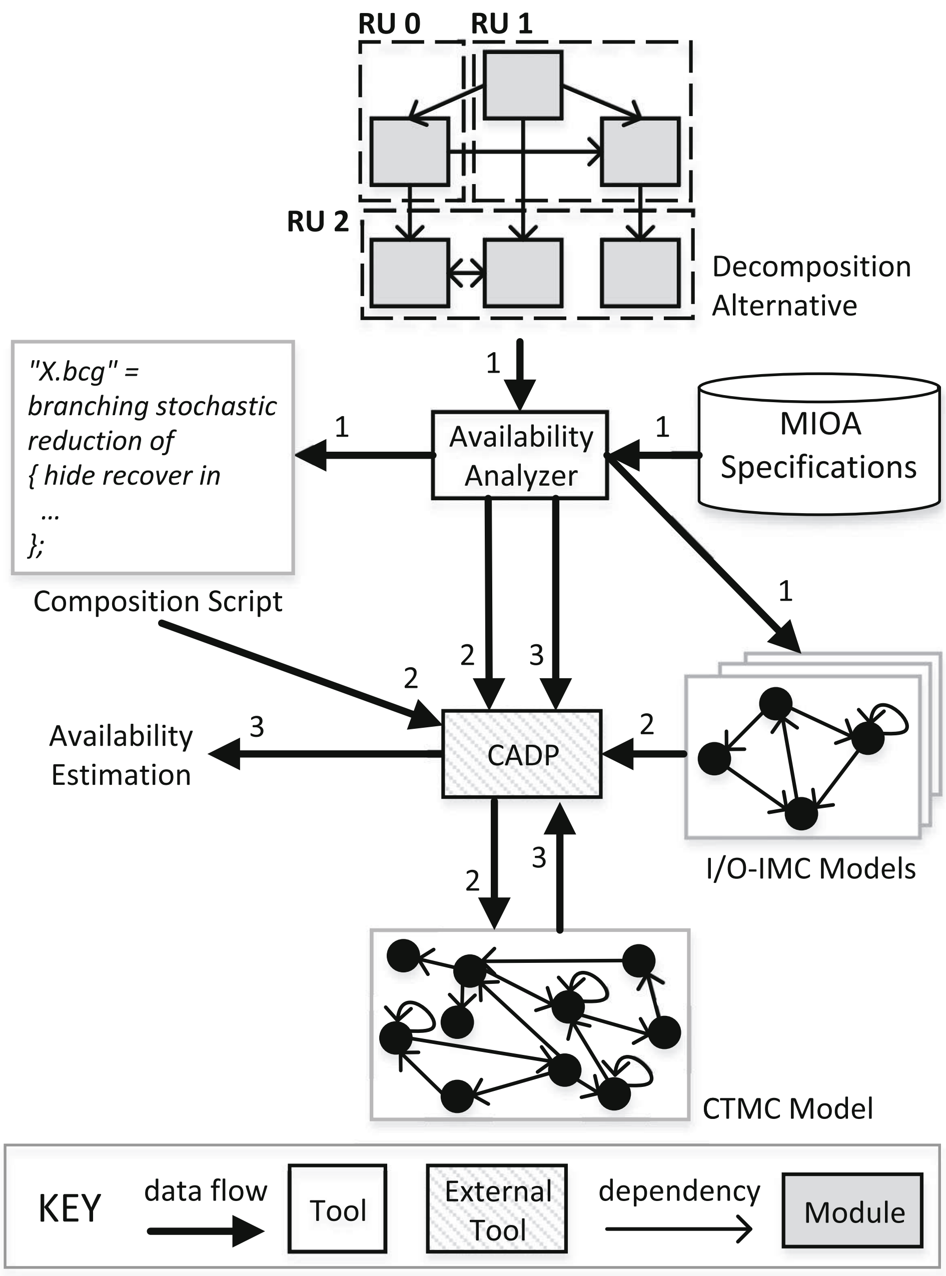

Fig. 2 Availability analysis process

4 Modeling approach

The overall modeling/analysis procedure is divided into three steps, given software modules decomposition as a set partition, together with the MTTF and MTTR for each module:

1. An I/O-IMC model is automatically generated for each module, each RU,[5] and the RM.
2. All the generated I/O-IMCs are automatically composed into a single I/O-IMC, describing the behavior of the whole system. During composition, the compositional-

[5] As described later, for each RU, two models are in fact generated.

aggregation technique is used to efficiently generate the state space. The final I/O-IMC is automatically converted into a CTMC model.

3. The output CTMC model is analyzed to compute system availability.

As mentioned above, a software system can be divided into several RUs, and each RU contains a certain number of modules. The recovery manager (RM) achieves local recovery by monitoring all the RUs and initiating recovery upon the detection of an RU failure. The RUs do not directly communicate with each other; however, they are inter-dependent given that they all interact with the RM.

Failure/recovery behavior of components in FLORA is characterized by the following properties:

P1: The failure of any module within an RU causes the entire RU failure.
P2: Errors of modules are independent and they do not propagate beyond the boundaries of RUs.
P3: The recovery of an RU entails the restart of all its modules (even the ones that did not fail).
P4: Only one RU can be recovered at a time and the RM recovers the RUs on a first-come-first-served (FCFS) basis.
P5: The restart of the modules inside a given RU is sequential.

As a modeling choice, the CM is considered to a be part of the RM, and as such, it is not modeled separately. In addition, we make the following assumptions in our models:

A1: The failure and recovery[6] of a module are governed by an exponential distribution[7] (i.e., constant failure rate).
A2: The RM does not fail and it always correctly detects a failing RU.
A3: The recovery always succeeds.

Based on experimentation, we think that these assumptions are reasonable and they can be easily changed within our framework (see Sect. 4.4).

Furthermore, the RM only interfaces with RUs and is unaware of the modules within RUs. To this end, each RU exhibits two interfaces: a *failure interface* and a *recovery interface*. The failure interface essentially listens to the failure of the modules within the RU and outputs an RU 'failure signal' upon the failure of a module. It also outputs an RU 'up signal' upon the successful restart of all the modules. The RM listens to 'failure' and 'up' signals emitted by the failure interfaces of the RUs. The recovery interface is in charge of the actual recovery of the various RU's modules. Upon the receipt of a 'start_recover' signal from the RM, it starts a sequential recovery of the modules inside the RU. Each module, recovery interface, failure interface, and the RM has a corresponding I/O-IMC model. Figure 3 illustrates the interaction between these different models.

4.1 The underlying I/O-IMC modeling formalism

Input/output interactive Markov chain (I/O-IMC) (Boudali et al. 2007a, 2008) is the underlying state-based modeling formalism we use. I/O-IMCs are a combination of input/

[6] The recovery time includes the time for restarting failed modules and also the time for error detection, error notification and diagnosis.

[7] An exponential distribution might not be, in some cases, a realistic choice; however, it is also possible to use a phase-type distribution which approximates any distribution arbitrarily closely.

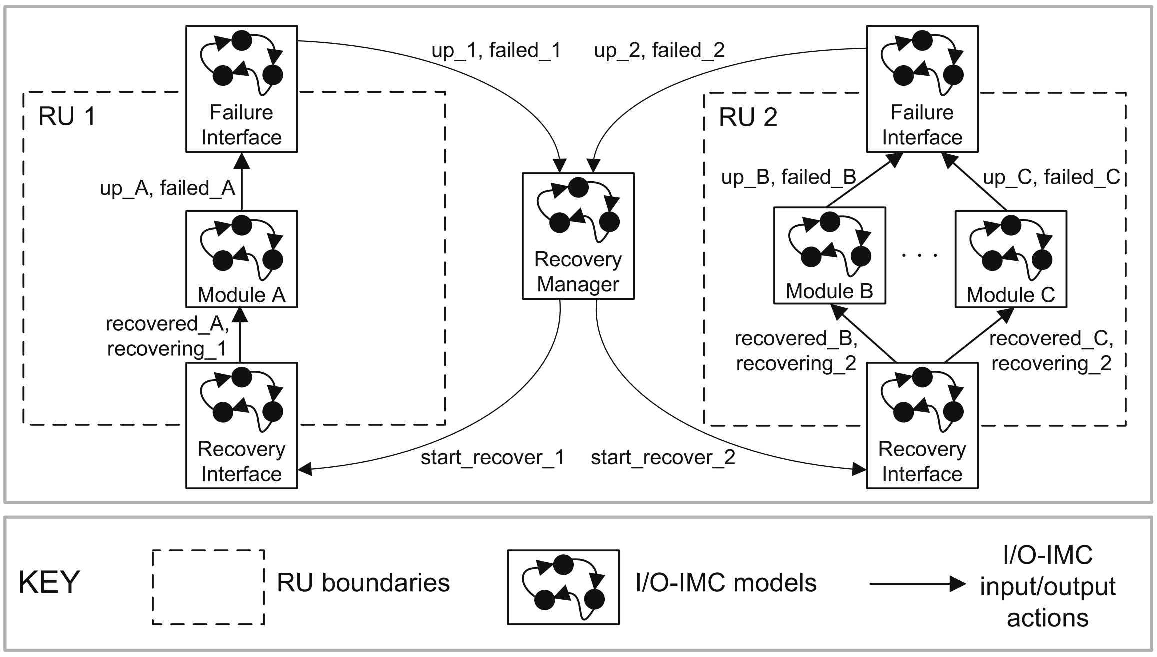

Fig. 3 Interaction between the various I/O-IMC models. A *dashed box* indicates an RU boundary, and a *solid box* indicates an I/O-IMC model

output automata (I/O-automata) (Lynch and Tuttle 1989) and interactive Markov chains (IMCs) (Hermanns 2002). 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 (or keyword 'rate'), and interactive transitions by solid lines. Figure 4 (taken from Boudali et al. 2008) 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 $S5$. 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 *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

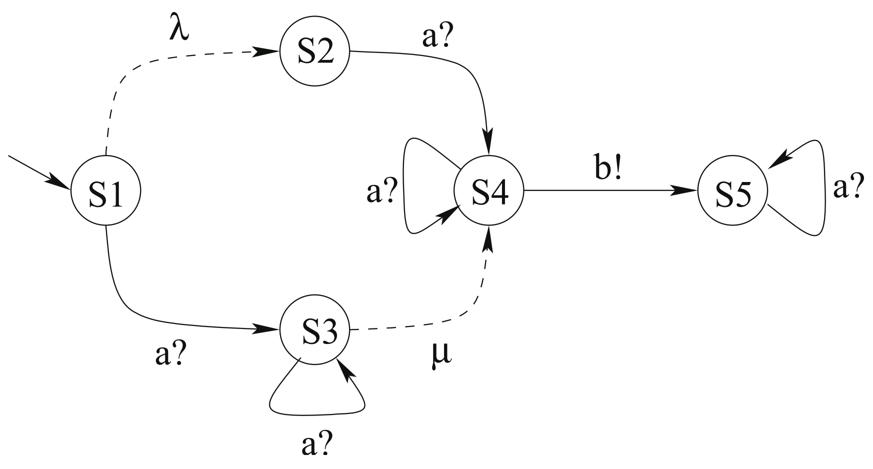

Fig. 4 An example I/O-IMC model

that same action (i.e., has 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 (details can be found in Boudali et al. 2007a).

Another important operation on I/O-IMCs is aggregation (or minimization). Aggregation is the process of transforming an I/O-IMC into a smaller and equivalent I/O-IMC. This is indeed a state-space reduction which generalizes the notion of *lumping* in CTMCs. In this work, we have used *weak bisimulation* (Boudali et al. 2007a) to aggregate I/O-IMCs. It was previously shown that weak bisimulation satisfies congruence with parallel composition and hiding, and it leads an equivalent model in terms of behavior (Boudali et al. 2007b).

The *compositional-aggregation* technique is a key procedure, used within the I/O-IMC formalism, for obtaining the overall system I/O-IMC model by composing, in successive iterations, a number of smaller I/O-IMCs (corresponding to the various system components) and reducing the state space of the generated I/O-IMC as the composition takes place. The compositional-aggregation technique has proved to be very effective in combating the infamous state-space explosion problem encountered in such models. The resulting system I/O-IMC reduces (in many cases) to a CTMC which can be then analyzed using standard methods to compute performance and/or dependability measures.

4.2 Specification and automatic generation of I/O-IMC models

We use a formal language called MIOA (Kuntz and Haverkort 2008) to describe any I/O-IMC model. MIOA is based on the IOA language defined in Garland et al. (2004). The MIOA language is used to describe concisely and formally an I/O-IMC in the same way the IOA language describes I/O-automata. The MIOA (or IOA) language provides programming language constructs, such as control structures and data types, to describe complex system model behaviors. Once a MIOA specification/description of an I/O-IMC model has been laid down, an algorithm explores the state space and automatically generates the corresponding I/O-IMC model. In fact, automatically deriving the I/O-IMC models becomes essential as the models grow in size. For instance, the RM I/O-IMC coordinating 7 RUs has 27,399 states and 397,285 transitions. In our framework, the RM I/O-IMC size is $O(n!)$, where n is the number of RUs. The failure and recovery interface

I/O-IMC sizes are $O(2^m)$ and $O(m)$, respectively, where m is the number of modules within the RU. The module I/O-IMC size is constant (i.e., four states).

In the following, we will first outline the basics of the MIOA language and then explain the generation of I/O-IMC models based on a MIOA specification.

The MIOA language Fig. 5 shows the MIOA specification of the failure interface I/O-IMC model as an example. Any MIOA specification is divided into three sections: (1) *Signature* where input/output/internal signals and Markovian rates are specified (Lines 2–4), (2) *States* where the states of the I/O-IMC are defined in terms of variables (Lines 5–7), and (3) *Transitions* where the I/O-IMC transitions are defined in a precondition-effect style (Lines 8–24). In order for the transition to take place, the precondition, which is a Boolean expression, has to hold.

In Fig. 5, the signature consists of the failed/up signals of the n modules belonging to the RU (Line 3), one output signal of the RU 'failed' signal, and one output signal of the RU 'up' signal (Line 4). The states of the failure interface I/O-IMC are defined (Lines 5–7) using *Set* and *Bool* data types, where 'set' (of size n) holds the names of the modules that have failed and 'rufailed' indicates if the RU has or has not failed. The initial state is also defined in the *States* section; for instance, the failure interface initial state is composed of 'set' being empty (Line 6) and 'rufailed' being false (Line 7). There are 4 kinds of possible transitions; for example, the last transition (Lines 21–24) indicates that an RU 'up' signal (up_RU!) is output if 'set' is empty (i.e., all modules are operational) and the RU has indeed failed at some point (i.e., 'rufailed' = true), and the effect of the transition is to set 'rufailed' to false.

Generation of I/O-IMC models We have implemented the I/O-IMC model generation based on the MIOA specifications. Algorithm 1 shows how the states and transitions of an I/O-IMC are generated based on its MIOA specification.

```
 1: IOIMC: failure interface
 2:   signature:
 3:     input: failed(n:Int)? up(n:Int)?
 4:     output: failed_RU! up_RU!
 5:   states:
 6:     set: Set[n:Int] := {}
 7:     rufailed: Bool := false
 8:   transitions:
 9:     input: failed(i)?
10:     effect:
11:       if i ¬∈ set
12:         add(i, set)
13:     input: up(i)?
14:     effect:
15:       if i ∈ set
16:         remove(i, set)
17:     output: failed_RU!
18:     precondition: set.size() > 0 ∧ rufailed = false
19:     effect:
20:       rufailed := true
21:     output: up_RU!
22:     precondition: set.size() = 0 ∧ rufailed = true
23:     effect:
24:       rufailed := false
```

Fig. 5 MIOA specification of the failure interface I/O-IMC model

Algorithm 1 State space exploration and I/O-IMC generation based on MIOA specification

```
1: $stateSet \leftarrow \{\}$
2: $statesToBeProcessed \leftarrow \{\}$
3: $transitionSet \leftarrow \{\}$
4: $s \leftarrow intialize$ mioa.states
5: $stateSet.add(s)$
6: $statesToBeProcessed.add(s)$
7: while $statesToBeProcessed \neq \{\}$ do
8:     $s \leftarrow statesToBeProcessed.removeLast()$
9:     for each mioa.transition $t$ do
10:        if $s.check(t.\text{precondition}) = \text{TRUE}$ then
11:            $snew \leftarrow s.apply(t.\text{effect})$
12:            if $snew \notin stateSet$ then
13:                $stateSet.add(snew)$
14:                $statesToBeProcessed.add(snew)$
15:            end if
16:            $transitionSet.add(s.id, snew.id, t.\text{signal})$
17:        else
18:            if $t.\text{signal} = input$ then
19:                $transitionSet.add(s.id, s.id, t.\text{signal})$
20:            end if
21:        end if
22:    end for
23: end while
24: $return\ new\ IOIMC(stateSet, transitionSet)$
```

The algorithm keeps track of the set of states, the states that are yet to be evaluated and the set of transitions generated, which are all initialized as empty sets (Lines 1–3). An initial state is created which comprises the initialized *state variables* as defined in the MIOA specification (Line 4). The initial state is added to the state set and the set of states to be processed (Lines 5–6). Then, the algorithm iterates over the states in the set of states to be processed until there is no state left to be processed (Lines 7–8). For each state, all the transitions that are specified in the MIOA specification are evaluated (Line 9). If the *precondition* of the transition holds, a new state is created, on which the *effects* of the transition are reflected (Lines 10–11). If the resulting state does not already exist, it is added to the set of states and the set of states to be processed (Lines 12–15). Also, a new transition from the original state to the resulting state is added to the set of transitions (Line 16). If the *precondition* of the transition does not hold for an *input signal*, then a self-transition is added to the set of transitions to ensure that the generated I/O-IMC is *input-enabled* (Lines 17–22).

4.3 I/O-IMC models for local recovery

In this section, we provide details on the four basic I/O-IMC models used in our framework, namely the module, the failure interface, the recovery interface, and the recovery manager. The running example (Fig. 3) consists of two RUs: RU 1 has one module A and RU 2 has two modules B and C, and a recovery manager. By convention, the starting state of any I/O-IMC is state 0 and the RUs are numbered starting from 1.

The module I/O-IMC Fig. 6 shows the I/O-IMC of module B. The module is initially operational in state 0, and it can fail with rate 0.2 and move to state 2. This means

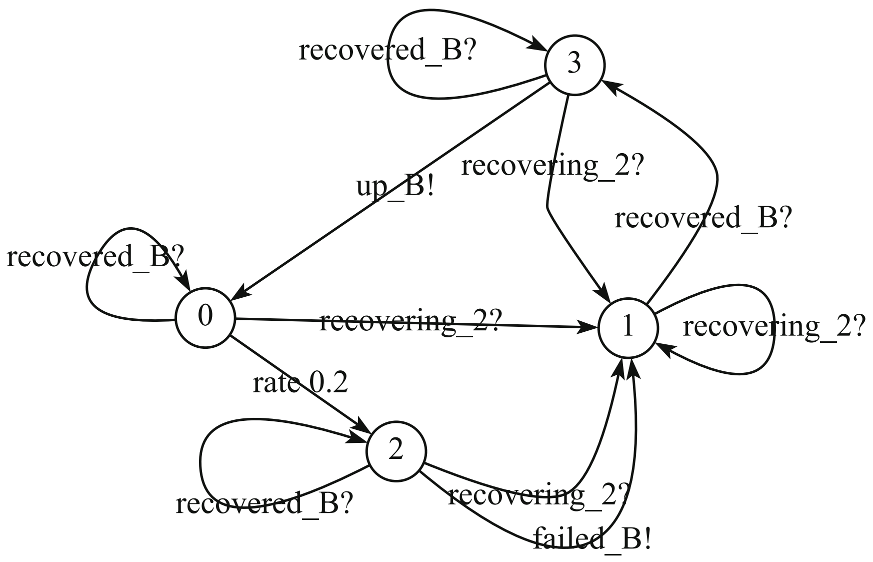

Fig. 6 The module I/O-IMC model for module B in RU 2

$MTTF = 5$ h as such $rate = 1/MTTF = 0.2$ (1/h). In state 2, the module notifies[8] the failure interface of RU 2 about its failure (i.e., transition from state 2 to 1). In state 1, the module awaits to be recovered (i.e., receiving signal 'recovered_B' from the recovery interface), and once this happens, it outputs an 'up' signal notifying the failure interface about its recovery (i.e., transition from state 3 to 0). Signal 'recovering_2' is received from the recovery interface, indicating that a recovery procedure of RU 2 has been initiated. The remaining input transitions are necessary to make the I/O-IMC input-enabled.

The failure interface I/O-IMC Figure 7 shows the I/O-IMC model of RU 2 failure interface. The failure interface simply listens to the failure signals of modules B and C, and outputs an RU 'failure' signal (i.e., 'failed_2') upon the receipt of any of these two signals. In fact, this interface behaves as an OR Boolean logic. Subsequently, the failure interface outputs an RU 'up' signal (i.e., 'up_2') when the failed module(s) has(have) output its(-their) 'up' signal(s). For instance, consider the following sequence of states: 0, 1, 4, 7, and 0; this corresponds to modules B and C being initially operational, then B fails, followed by RU 2 outputting its failure signal, then signal 'up_B' is received from module B, and finally RU 2 outputs its own 'up' signal.

The recovery interface I/O-IMC Figure 8 shows the I/O-IMC model of RU 2 recovery interface. The recovery interface receives a 'start_recover' signal from the RM (i.e., transition from state 0 to 1), allowing it to start the RU's recovery. A 'recovering' signal is then output (i.e., transition from state 1 to 2) notifying all the modules within the RU that a recovery phase has started (essentially disallowing any remaining operational module to fail). Then two sequential repairs (i.e., of B and C) take place both with rate 1 (i.e., transitions from state 2 to 3 and 3 to 4), followed by two sequential 'recovered' notifications (i.e., transitions from state 4 to 5 and 5 to 0).

The recovery manager I/O-IMC Figure 9 shows the I/O-IMC model of the RM. The RM monitors the failure of RU 1 and RU 2, and when an RU failure is detected, the RM grants its recovery by outputting a 'start_recover' signal. The RM has internally a queue of failing

[8] Note that these models are used for availability estimation only and they do not necessarily reflect software implementation. In an actual implementation, a module might not be aware of its failure to notify it. External error detection mechanisms can be employed (Sozer et al. 2009) for this purpose.

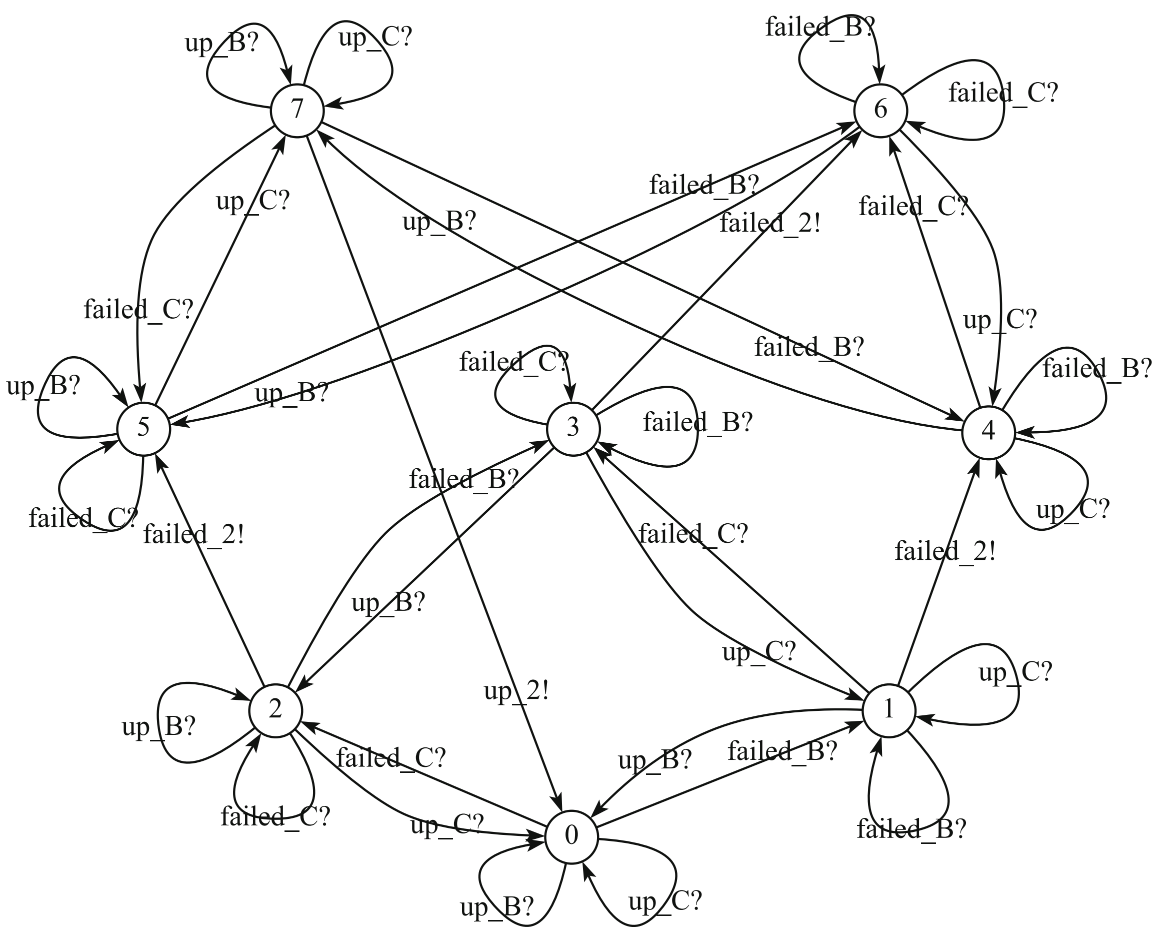

Fig. 7 The failure interface I/O-IMC model for RU 2

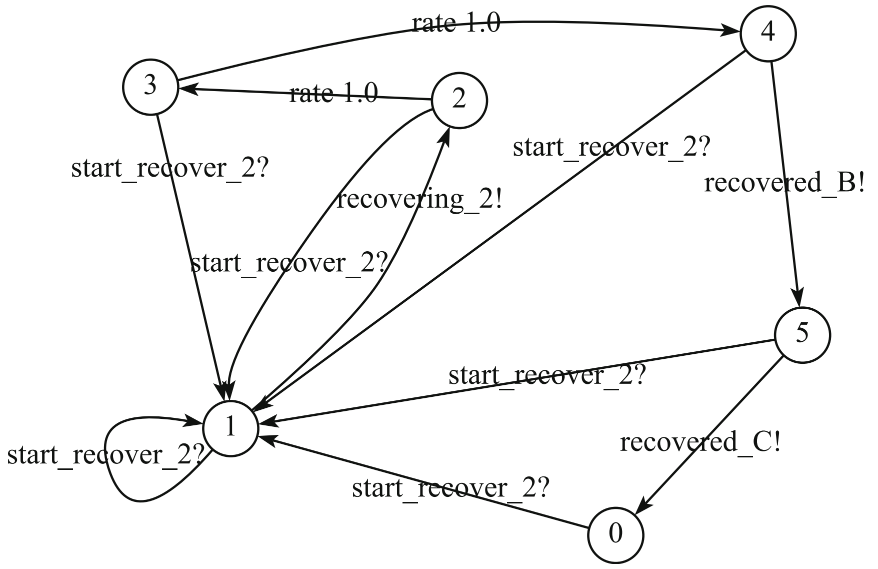

Fig. 8 The recovery interface I/O-IMC model for RU 2

RUs that keeps track of the order in which the RUs have failed. The RM recovery policy is to grant a ‘start_recover’ signal to the first failing RU (i.e., FCFS). For instance, consider the following sequence of states: 0, 1, 4, 7, 2, 6, and 0; this corresponds to both RUs being initially operational, then RU 1 fails, immediately followed by an RU 2 failure. Since RU 1

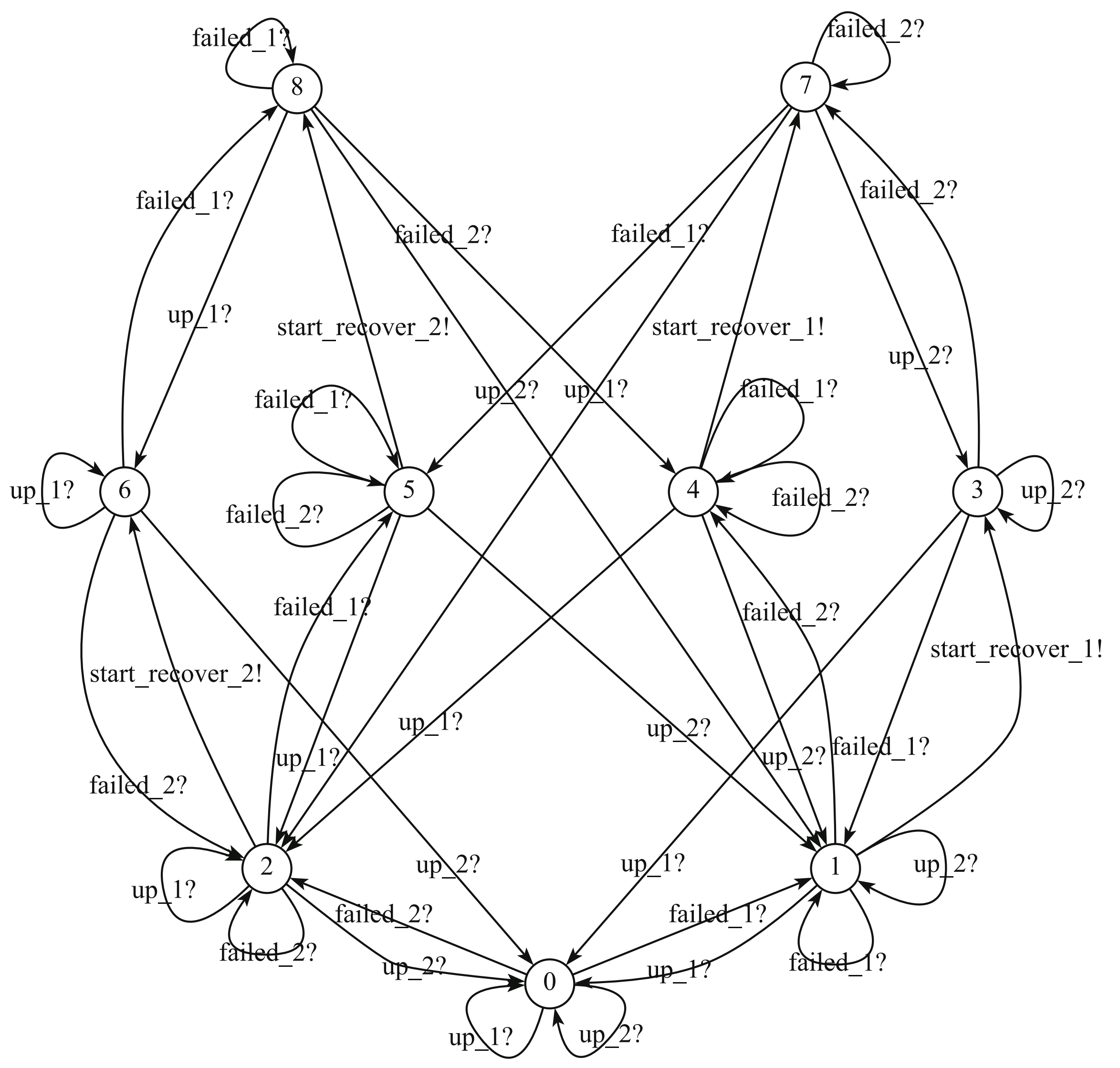

Fig. 9 The recovery manager I/O-IMC model

failed first, it is granted the 'start_recover' signal (i.e., transition from state 4–7), the RM then awaits for RU 1 'up' signal, and once received, RM grants the 'start_recover' signal to RU 2 (as RU 2 is still in the queue of failing RUs) (i.e., transition from state 2–6). Finally, the RM receives 'up_2' and both RUs are operational again.

It should be noted that I/O-IMC models are not exposed to the user. They are automatically generated and composed. Therefore, their readability and understandability are not a concern.

4.4 I/O-IMC modeling flexibility

Modeling flexibility and modularity is a powerful feature of the I/O-IMC formalism (Boudali et al. 2007b, 2008). Any of the I/O-IMC models can be locally modified without a need to modify the other models. For example, if the recovery policy is not going to be FCFS anymore, the model regarding the recovery manager (Fig. 9) should be changed only. The other models can be used as is. In principle, one can form a library of recovery manager I/O-IMC models each of which defines a different strategy. One of them can be picked during the model composition step. The complexity of these models can differ based on the defined strategy.

One can also alter some or all of the assumptions made above. For instance, the models can be improved to reflect the real system behavior by (1) using failure (or repair) distributions other than the exponential distribution, (2) explicitly model the communication manager and the various communication delays, or (3) allow the RM to fail. To be able to incorporate such changes, one should modify the corresponding analytical model (MIOA specification). To specify redundancy, for instance, one should introduce a different failure interface I/O-IMC model (Fig. 7), which defines how/when a failure happens. The current model is generated based on the MIOA specification shown in Fig. 5, where a failure signal (i.e., 'failed_RU') is triggered if one of the modules fails. This behavior can be modified such that the failure signal is triggered only when two or more redundant modules fail.

4.5 Implementation details

The overall data flow and the set of artifacts of *Availability Analyzer* are depicted in Fig. 2. Hereby, the input decomposition alternative is simply represented as a set partition. For example, the decomposition with three modules and two RUs, as shown in Fig. 3, is provided as { [A(mttr,mttf)] [B(mttf,mttr), C(mttf,mttr)]}, where mttf and mttr are values of the module's MTTF and MTTR (reciprocal of the failure and repair rate, respectively).

Availability Analyzer maps all the modules and RUs (i.e., partitions) in the specification to MIOA specifications, which are used by Algorithm 1 to explore the state space and generate the corresponding I/O-IMCs. Based on the specified decomposition, *Availability Analyzer* also generates the corresponding I/O-IMC for the RM. All the generated I/O-IMC models are output in a file format (*Aldebaran*) that can be processed with the CADP toolset (Garavel et al. 2007). In addition to generating all the necessary I/O-IMCs, *Availability Analyzer* generates a composition/analysis script, which conforms to the CADP SVL scripting language (Garavel and Lang 2001). Figure 10 shows a part of the generated SVL script, which composes the I/O-IMC models for the example decomposition with three modules and two RUs (See Fig. 3).

The generated composition script first composes the recovery interface I/O-IMCs of RUs with the module I/O-IMCs that are comprised by the corresponding RUs. For instance, in Fig. 10, the I/O-IMCs of modules *B* and *C* are composed with the recovery interface I/O-IMC of *RU 1* (Lines 6–10). Similarly, the module *A* I/O-IMC is composed with the recovery interface of *RU 2* (Lines 16–18). The resulting I/O-IMCs are then composed with the failure interface I/O-IMCs of the corresponding RUs. Finally, all the resulting I/O-IMCs are composed with the RM I/O-IMC. At each composition step, the common input/output actions that are only relevant for the I/O-IMCs being composed are "hidden." That is, these actions are used for the composition of I/O-IMCs and eliminated in the resulting I/O-IMC. The execution of the generated SVL script within CADP composes and aggregates all the I/O-IMCs based on the modules decomposition, reduces the final I/O-IMC into a CTMC, and computes the steady-state availability.

5 Case study

5.1 MPlayer

We have applied local recovery to an open-source software, MPlayer (2015). MPlayer is a media player, which supports many input formats, codecs, and output drivers. It has approximately 700K lines of code and it is available under the GNU General Public

```
"X.bcg" = branching stochastic reduction of(
(hide start_recover_1 in
  (hide failed_B,up_B,failed_C,up_C in
"fint_1.aut"
    |[failed_1,failed_B,up_B,failed_C,up_C]|
    (hide recovered_B in "module_B.aut"
    |[recovered_B]|
    (hide recovered_C in "module_C.aut"
    |[recovered_C]|
    "rint_1.aut"))
  )
  |[start_recover_1,failed_1,up_1]|
(hide start_recover_2 in
  (hide failed_A,up_A in "fint_2.aut"
    |[failed_2,failed_A,up_A]|
    (hide recovered_A in "module_A.aut"
    |[recovered_A]|
    "rint_2.aut")
  )
  |[start_recover_2,failed_2,up_2]|
"recmgr.aut")));
```

Fig. 10 SVL script that composes the I/O-IMC models according to the example decomposition as shown in Fig. 3

License. In our case study, we have used version v1.0rc1 of this software that is compiled on a Linux Platform (Ubuntu version 7.04).

To introduce local recovery, we have to decompose the MPlayer software architecture into a set of RUs. One possible decomposition of system modules is shown in Fig. 11. In this example, the system is partitioned into three RUs;

– *RU AUDIO*: wraps the *Libao* module, which controls the playing of audio.
– *RU GUI*: comprises the *Gui* module, which provides the graphical user interface of MPlayer.
– *RU MPCORE*: encapsulates five modules of the system; *Stream* reads the input media and provides buffering, seek and skip functions. *Demuxer* separates the input to audio and video channels. *Mplayer* connects the other modules and maintains the audio–video synchronization. *Libmpcodecs* embodies the set of available codecs. *Libvo* displays video frames.

The decomposition of the system into a set of RUs is a semiautomated process that is supported by the FLORA framework (Sozer et al. 2009). The architect/developer needs to define the set of RUs, their contents (set of modules), and dependencies in the form of a list of shared variables and exchanged function calls (interface). Then, the framework can automatically perform the decomposition by synchronizing shared variables and forwarding function calls via inter-process communication. We implemented several decompositions in this way to compare alternative decompositions and evaluate the accuracy of analytical estimations as described in the following subsection.

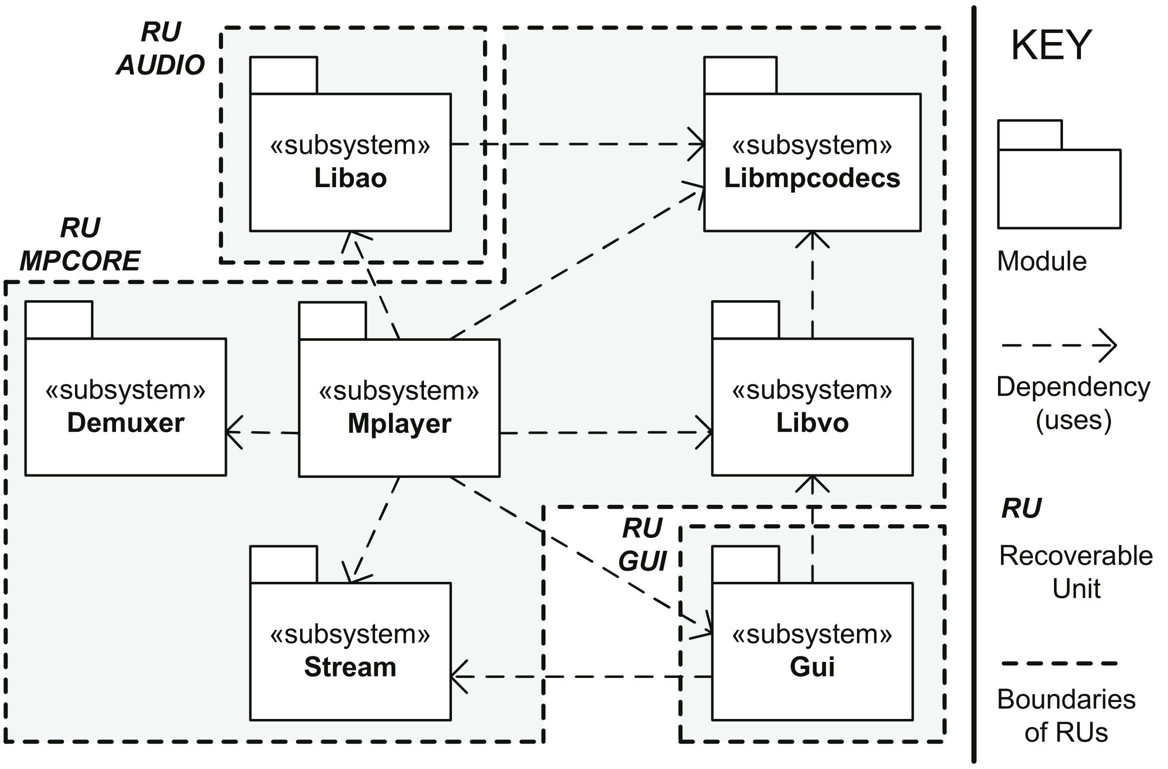

Fig. 11 The modules decomposition view of the MPlayer software architecture

5.2 Experimentation and analysis

We have implemented local recovery for a total of three decomposition alternatives of MPlayer. (1) Global recovery, where all the modules are placed in a single RU ({[Mplayer, Libmpcodecs, Libvo, Demuxer, Stream, Gui, Libao]}), (2) local recovery with two RUs, where the module *Gui* is isolated from the rest of the modules ({[Mplayer, Libmpcodecs, Libvo, Demuxer, Stream, Libao] [Gui]}), (3) local recovery with three RUs, where the module *Gui*, *Libao*, and the rest of the modules are isolated from each other ({[Mplayer, Libmpcodecs, Libvo, Demuxer, Stream] [Libao] [Gui]}).

To be able to measure and compare the availability of these three implementations, we have modified each module so that they fail with the specified failure rates (MTTF). Hereby, we do not employ a common fault/error injection (Durares and Henrique 2006) approach, where the module can fail or not depending on the error propagation behavior. Our approach can be regarded as *failure injection* (Monnet and Bertier 2007; Alvarez and Cristian 1997), where we directly make the system crash in exponentially distributed time intervals. This is in alignment with our analysis approach, in which we make use of MTTF values that represent the mean time to *failure*. In the experiment, our goal is to evaluate if the analysis is accurate assuming that the provided input failure rates are correct. Hence, we consider fault triggers and error propagation behavior out-of-scope. As such we do not simulate these.

To simulate failure rates with exponential distribution, one can generate a probability value with uniform distribution and use it as a parameter for calculating the inverse of cumulative exponential distribution to find the amount of time that passes until the next failure (Devroye 1986). This amount can be compared with the time elapsed since the last failure to decide if a new failure should be triggered or not. We simulate failures with threads that are periodically activated to crash the process they work on. These threads regularly sleep for a second; however, operating system controls when exactly they become awake. So, we perform a pseudorandom number sampling within each thread as follows. We use cumulative distribution function of exponential distribution to calculate the probability of failure based on the time elapsed since the last failure. Then we draw a random probability value according to uniform distribution and decide on triggering the failure or not by comparing this value with the calculated value. One such thread is created for each module when the module is initialized. The operation of the thread is shown in Algorithm 2.

Algorithm 2 Periodically activated thread for failure injection

```
1: $time\_init \leftarrow currentTime()$
2: while TRUE do
3:    $time\_elapsed \leftarrow currentTime() - time\_init$
4:    $p \leftarrow 1 - 1/e^{time\_elapsed/MTTF}$
5:    $r \leftarrow random()$
6:    if $p \geq r$ then
7:       $time\_init \leftarrow currentTime()$
8:       $triggerFailure()$
9:       $break$
10:   end if
11:   $sleep$
12: end while
```

The failure injection thread first records the initialization time (Line 1). Then, each time it is activated, the thread calculates the time elapsed since the initialization (Line 3). The MTTF value of the corresponding module and the elapsed time is used for calculating the probability of error occurrence—assuming an exponential distribution—(Line 4). *random*() returns, from a uniform distribution, a sample value $r \in [0, 1]$ (Line 5). This value is compared to the calculated probability to decide whether or not to trigger a failure (Line 6). An illegal memory operation is performed (Line 8) to trigger a failure by crashing the process, on which the module is running.

The RM component of FLORA logs the failure and recovery times of RUs to a file during the execution of the system. For each of the implemented alternatives, we ran the system for 5 hours. Then, we have processed the log files to calculate the cumulative time T_{avail} when the RU that contains the core system module, *Mplayer*, has been operational. The whole system is unavailable if and only if this RU is unavailable. So, T_{avail} corresponds, by definition, to the system availability as a whole. We have calculated the steady-state availability of the system per hour as $\frac{T_{avail}}{5}$. The results of the measured system availability are given in Table 2 for the different alternatives. Table 2 also shows the estimated system availability based on the analytical models as described in Sect. 4.

We have used the MTTF values given in Table 1 for both the analytical models and the failure injection threads. We have measured the MTTR values from the actual implementation by calculating the mean time it takes to restart a process and the corresponding modules over 100 runs. The measured MTTR values are used in the analytical models as listed in Table 1. We previously introduced an architectural style (Sozer et al. 2013) as a specialization of xADL (Dashofy et al. 2002) and utilized a graphical user interface as an extension of ArchStudio (Dashofy et al. 2002) to be able to specify failure and repair rates for system modules. In the background, this specification is kept in the form of an XML

Table 1 Measured MTTR values and specified MTTF values for the MPlayer modules

Module	MTTR (ms)	MTTF (s)
Libao	480	60
Libmpcodecs	500	1800
Demuxer	540	1800
Mplayer	800	1800
Libvo	400	1800
Stream	400	1800
Gui	600	30

document. Listing 1 shows a simplified example snippet from such a specification, where the MTTF and MTTR values for the *Mplayer* module are defined.

Listing 1 A sample XML code snippet for the specification of MTTF and MTTR values.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<instance:xArch ...>
    <types:archStructure ... xsi:type="types:ArchStructure">
        <types:description ...>module view</types:description>
        <types:component ...>
            <types:description ...>Mplayer</types:description>
            <types:ReliabilityInterface ...>
                <types:MTTF ...>1800</types:MTTF>
                <types:MTTR ...>800</types:MTTR>
            </types:ReliabilityInterface>
        </types:component>
         ...
    </types:archStructure>
</instance:xArch>
```

For the sake of comparison, the last row of Table 2 shows the estimated availability for the most extreme decomposition alternative. In this alternative, each module of the MPlayer is placed in a different RU (i.e., seven RUs in total) to isolate all the modules from each other. Hence, it leads to the highest availability that can be achieved (although such an alternative might not always be feasible due to constraints imposed by the domain, deployment, and performance requirements). The measured availability is not shown here since we do not have an implementation for this decomposition alternative. It is very hard to actually implement this alternative by refactoring legacy source code. In general, implementing local recovery for any decomposition is a time-consuming procedure. However, one can build analytical models for decomposition alternatives including the one

Table 2 Comparison between the estimated and measured system availability

Decomposition alternative	Measured availability	Estimated availability
all modules in 1 RU	83.27	83.60
Gui, *the rest*	92.31	93.25
Gui, *Libao*, *the rest*	97.75	98.70
each module in a separate RU	–	99.96

in which each module is placed in a separate RU. As such, it is possible to get an estimation for any alternative. In the following, we discuss model size and the analysis time for all the decomposition alternatives listed in Table 2.

Table 3 lists the number of states in the final CTMC models after composition and the time it took to perform the analysis on an average machine (i.e., single core, $2GHz$ computer with $1GB$ memory). Hereby, the analysis time also includes the time it took to perform the composition/aggregation operations on intermediate models. Also note that the table lists the size of the final, reduced CTMC models. Intermediate models used during the analysis process can be much larger. For instance, the largest CTMC encountered during the analysis of the first decomposition alternative had 9966 states, although the final CTMC has only eight states.

In Table 2, we observe that the measured availability and the estimated availability values (in %) are quite close to each other. In general, the measured availability is lower than the estimated availability. This is due in part to the communication delays in the actual implementation, which are not accounted for in the analytical models. Communication among system modules is modeled using interactive transitions in the I/O-IMC models, which are instantaneous. Hence, communication time delays are abstracted away in these models. However, in reality, the recovery time includes the time for error detection, diagnosis, and communication among multiple processes, which are subject to delays due to process context switching and inter-process communication overhead.

6 Discussion

In this work, we use the *recovery style* (Sozer and Tekinerdogan 2008) to document different architectural design alternatives for local recovery. This style was introduced for making a local recovery design explicit at the architecture design level. On the one hand, it is not a generically applicable style like common styles in practice such as the layered architectural style. On the other hand, it is a specialization of, and therefore conforms to, a generic viewtype, namely the module viewtype introduced in the Views and Beyond approach (Clements et al. 2010). This specialization approach is also aligned with the IEEE 1471 standard (Maier et al. 2001), which does not commit to any set of styles because of the existence of different concerns that need to be addressed for different systems. Therefore, the set of views should not be fixed and multiple viewpoints might be introduced by specializing existing generic viewpoints to document a particular concern. To the best of our knowledge, there is no other architectural style proposed in the literature to specifically document a local recovery design.

In this paper, we evaluated four different decomposition alternatives. The number of such alternatives grows exponentially with respect to the number of modules in the system. It turns out that the total number of ways to partition a set of n elements into arbitrary

Table 3 The size and the analysis time of the automatically generated I/O-IMC models

Decomposition alternative	Number of states	Analysis time
all modules in 1 RU	8	1 min 6 s
Gui, *the rest*	20	1 min 26 s
Gui, *Libao*, *the rest*	60	1 min 49 s
each module in a separate RU	13700	4 min 47 s

number of non-empty, disjoint sets is counted by the *n*th *Bell number* (Sozer et al. 2013). This number is 15 and 877 when the number of modules *n* is 4 and 7, respectively. It becomes more than a billion when *n* is 15. Therefore, an exhaustive search in this design space might not be always feasible. For this reason, we previously introduced a hill-climbing approach (Sozer et al. 2013), where we start from a decomposition and search the design space through neighbor alternatives. A neighbor alternative is defined as the same decomposition except a transfer of a module to a different RU. A new RU is created by moving a module to a new RU or an RU is removed when its only module is moved into another RU. The algorithm terminates when none of the neighbor alternatives leads to improvement with respect to the objective function. The drawback of this approach is that the algorithm can get stuck in a local optimum point (Sozer et al. 2013). In this paper, we focused on estimating availability for a given decomposition alternatives. These estimations can be used as input for optimization.

As explained in Sect. 3.1, we have to cope with a trade-off between availability and performance overhead to select a decomposition alternative for local recovery. Therefore, we also need to estimate performance overhead for each decomposition alternative. To this aim, we previously developed a tool (Sozer et al. 2013), which can analyze source code to calculate the number of function calls between the selected RU boundaries. We measured the overhead caused per exchanged function and used it for estimating the overall overhead caused by a decomposition alternative by calculating the number functions exchanged among the RUs.

Figure 12 depicts a comparison between the estimated performance overhead and the estimated availability for the four decomposition alternatives listed in Table 2. The vertical axis on the left-hand side corresponds to availability. The vertical axis on the right-hand side corresponds to performance overhead. The ranges of these two axes are different (75–100 vs. 0–25); however, they have the same scale. The horizontal axis marks the number of RUs included in the corresponding decomposition alternative. In fact,

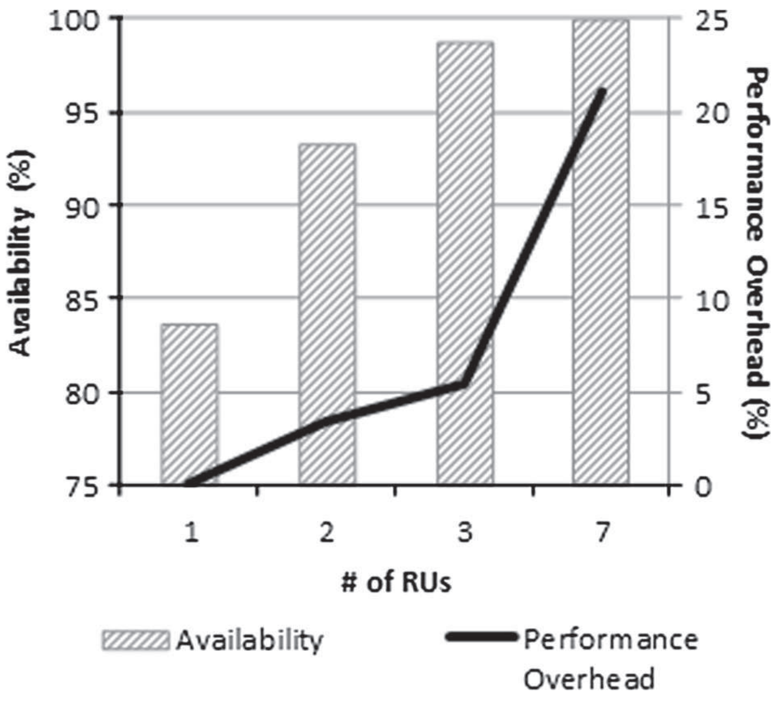

Fig. 12 The trade-off between availability and performance overhead

availability and performance overhead measures are not dependent only on the number of RUs, but also how these RUs are defined. If the number of RUs is equal to 1 or if it is equal to the number of modules, then there is only one decomposition alternative. This is, however, not the case if the number of RUs is more than 1 and less than the number of modules. In Fig. 12, we compare decomposition alternatives with two and three RUs as well. Hereby, we refer to the particular decomposition alternatives that are listed in the second and third rows of Table 2, respectively. Such comparisons can guide the designer to make trade-off decisions and select a decomposition accordingly. For instance, Figure 12 shows that increasing the number of RUs from 3 to 7 leads to only a minor improvement in availability compared with the significant increase in performance overhead.

7 Conclusion

Local recovery is applied to achieve higher system availability, and its effectiveness highly depends on the implemented software decomposition. In this paper, we presented a method that provides quantitative means to compare different software decomposition alternatives in terms of their availabilities. We have automated the whole analysis procedure with a Java/CADP-based tool. Local recovery was implemented for the open-source MPlayer software, and we have applied our quantitative approach to estimating the availability for four different decomposition alternatives. We have implemented three of these decomposition alternatives and the estimated availabilities turned out to be very close to the actual measured availabilities.

As further extensions of our work, one can revisit some or all of the assumptions made in Sect. 4 and/or modify any of the four basic I/O-IMC models to more accurately represent the real system behavior.

Acknowledgments We thank Pepijn Crouzen for his help using CADP and Boudewijn Haverkort for his comments on an earlier version of this paper. This work has been carried out as a part of the TRADER project under the responsibilities of the Embedded Systems Institute. This work is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program; by the Netherlands Organization for Scientific Research (NWO) under FOCUS/BRICKS Grant Number 642.000.505 (MOQS); and by the EU under Grants Numbers IST-004527 (ARTIST2) and FP7-ICT-2007-1 (QUASIMODO).

References

Alvarez, G., & Cristian, F. (1997). Centralized failure injection for distributed, fault-tolerant protocol testing. In *Proceedings of the 17th international conference on distributed computing systems*, pp. 78–85.

Avizienis, A., Laprie, J. C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, *1*(1), 11–33.

Bernardi, S., Merseguer, J., & Petriu, D. (2011). A dependability profile within MARTE. *Software and Systems Modeling*, *10*(3), 313–336.

Bernardi, S., Merseguer, J., Petriu, D., & Dorina, C. (2012). Dependability modeling and analysis of software systems specified with UML. *ACM Computing Surveys*, *45*(1), 1–48.

Boudali, H., Crouzen, P., & Stoelinga, M. (2007a). A compositional semantics for dynamic fault trees in terms of Interactive Markov Chains. In *Proceedings of the 5th international symposium on automated technology for verification and analysis, lecture notes on computer science (LNCS)*, pp. 441–456.

Boudali, H., Crouzen, P., & Stoelinga, M. (2007b). Dynamic fault tree analysis using input/output interactive Markov chains. In *Proceedings of the 37th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pp. 708–717.

Boudali, H., Crouzen, P., Haverkort, B. R., Kuntz, M., & Stoelinga, M. (2008). Architectural dependability evaluation with arcade. In *Proceedings of the 38th IEEE/IFIP international conference on dependable systems and networks (DSN)*, IEEE, pp. 512–521.

Bowles, J., Dobbins, J., & Gregory, J. (2004). Approximate reliability and availability models for high availability and fault-tolerant systems with repair. *Quality and Reliability Engineering International*, *20*(7), 679–697.

Bozzano, M., Cimatti, A., Katoen, J. P., Nguyen, V. Y., Noll, T., & Roveri, M. (2011). Safety, dependability and performance analysis of extended AADL models. *The Computer Journal*, *54*(5), 754–775.

Brosch, F., Koziolek, H., Buhnova, B., & Reussner, R. (2012). Architecture-based reliability prediction with the palladio component model. *IEEE Transactions on Software Engineering*, *38*(6), 1319–1339.

Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., & Fox, A. (2004b). Microreboot: A technique for cheap recovery. In *Proceedings of the 6th symposium on operating systems design and implementation (OSDI)*, San Francisco, CA, pp. 31–44.

Candea, G., Cutler, J., & Fox, A. (2004a). Improving availability with recursive micro-reboots: A soft-state system case study. *Performance Evaluation*, *56*(1–4), 213–248.

Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., et al. (2010). *Documenting software architectures: Views and beyond* (2nd ed.). Reading, MA: Addison-Wesley.

Das, O., & Woodside, C. (1998). The fault-tolerant layered queueing network model for performability of distributed systems. In *Proceedings of the international performance and dependability symposium (IPDS)* (pp. 132–141). Durham, NC.

Dashofy, E., van der Hoek, A., & Taylor, R. (2002). An infrastructure for the rapid development of XML-based architecture description languages. In *International conference on software engineering (ICSE)* (pp. 266–276). Orlando, Florida: ACM.

Devroye, L. (1986). *Non-uniform random variate generation*. Berlin: Springer.

Dugan, J., & Lyu, M. (1995). Dependability modeling for fault-tolerant software and systems. In M. R. Lyu (Ed.), *Software fault tolerance, chapter 5* (pp. 109–138). London: Wiley.

Durares, J., & Henrique, S. (2006). Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, *32*(11), 849–867.

ECLIPSE (2015) Eclipse foundation. http://www.eclipse.org/

Franco, J., Barbosa, R., & Zenha-Rela, M. (2012). Automated reliability prediction from formal architectural descriptions. In *Proceedings of joint working IEEE/IFIP conference on software architecture (WICSA) and European conference on software architecture (ECSA)*, pp. 302–309.

Franco, J., Barbosa, R., & Zenha-Rela, M. (2014). Availability evaluation of software architectures through formal methods. In *Proceedings of the 9th international conference on the quality of information and communications technology (QUATIC)*, pp. 282–287.

Garavel, H., & Lang, F. (2001). SVL: A scripting language for compositional verification. In *Proceedings of the international conference on formal techniques for networked and distributed systems (FORTE)*, pp. 377–394.

Garavel, H., Lang, F., Mateescu, R., & Serwe, W. (2007). CADP 2006: A toolbox for the construction and analysis of distributed processes. In *Computer-aided verification (CAV), Springer, Lecture Notes on Computer Science (LNCS), vol. 4590*, pp. 158–163.

Garlan, D., Monroe, R., & Wile, D. (1997). Acme: An architecture description interchange language. In *Proceedings of conference of the centre for advanced studies on collaborative research (CASCON)*, pp. 169–183.

Garland, S., Lynch, N., Tauber, J., & Vaziri, M. (2004). *IOA user guide and reference manual*. Tech. rep., MIT CSAI Laboratory, Cambridge, MA.

Geist, R., & Trivedi, K. (1990). Reliability estimation of fault-tolerant systems: Tools and techniques. *IEEE Computer*, *23*(7), 52–61.

Hermanns, H. (2002). *Interactive Markov Chains: The quest for quantified quality, lecture notes on computer science (LNCS), vol. 2428.*

Hermanns, H., & Katoen, J. P. (2000). Automated compositional Markov chain generation for a plain-old telephone system. *Science of Computer Programming*, *36*(1), 97–127.

Hunt, G. C., et al. (2007). Sealing OS processes to improve dependability and safety. *ACM SIGOPS Operating Systems Review*, *41*(3), 341–354.

Immonen, A., & Niemel, E. (2008). Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, *7*(1), 49–65.

Joyce, J. (2007). Architecting dependable systems with the sae architecture analysis and description language (AADL). In R. de Lemos, C. Gacek, & A. Romanovsky (Eds.), *Architecting dependable systems, IV, lecture notes in computer science, vol. 4615* (pp. 1–13). Berlin: Springer.

Kuntz, M., & Haverkort, B. R. (2008). *Formal dependability engineering with MIOA*. Technical Report TR-CTIT-08-39.

Lai, C. D., et al. (2002). A model for availability analysis of distributed software/hardware systems. *Information and Software Technology*, *44*(6), 343–350.

Lynch, N., & Tuttle, M. (1989). An introduction to input/output automata. *CWI Quarterly*, *2*(3), 219–246.

Maier, M., Emery, D., & Hilliard, R. (2001). Software architecture: Introducing IEEE standard 1471. *IEEE Computer*, *34*(4), 107–109.

Majzik, I., & Huszerl, G. (2002). Towards dependability modeling of FT-CORBA architectures. In *Proceedings of the 4th European dependable computing conference, lecture notes on computer science (LNCS)*, pp. 121–139.

Monnet, S., & Bertier, M. (2007). Using failure injection mechanisms to experiment and evaluate a grid failure detector. In M. Dayde, J. Palma, A. Coutinho, E. Pacitti, & J. Lopes (Eds.), *High performance computing for computational science, vol. 4395* (pp. 610–621). Berlin, Heidelberg: Springer.

MPLAYER (2015) MPlayer official website. http://www.mplayerhq.hu/

Rugina, A. E., Kanoun, K., & Kaaniche, M. (2007). A system dependability modeling framework using AADL and GSPNs. In R. de Lemos, C. Gacek, & A. Romanovsky (Eds.), *Architecting dependable systems, IV, lecture notes in computer science, vol. 4615* (pp. 14–38). Berlin: Springer.

Sozer, H. (2009). *Architecting fault-tolerant software systems*. Ph.D. thesis, University of Twente, Enschede, The Netherlands.

Sozer, H., & Tekinerdogan, B. (2008). Introducing recovery style for modeling and analyzing system recovery. In *Proceedings of the 7th working IEEE/IFIP conference on software architecture (WICSA)*, (pp. 167–176). Canada: Vancouver.

Sozer, H., Tekinerdogan, B., & Aksit, M. (2009). FLORA: A framework for decomposing software architecture to introduce local recovery. *Software: Practice and Experience*, *39*(10), 869–889.

Sozer, H., Tekinerdogan, B., & Aksit, M. (2013). Optimizing decomposition of software architecture for local recovery. *Software Quality Journal*, *21*(2), 203–240.

Vaidyanathan, K., & Trivedi, K. S. (2005). A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, *2*(2), 124–137.

Hasan Sözer received his B.Sc. and M.Sc. degrees in Computer Engineering from Bilkent University, Turkey, in 2002 and 2004, respectively. He received his Ph.D. degree in 2009 from the University of Twente, The Netherlands. From 2002 until 2005, he worked as a software engineer at Aselsan Inc. in Turkey. From 2009 until 2011, he worked as a postdoctoral researcher at the University of Twente. He is currently an assistant professor at Özyeğin University.

Mariëlle Stoelinga is an associate professor at the University of Twente, The Netherlands, where she leads a team on quantitative analysis and risk management of computer systems. She holds an M.Sc. and Ph.D. degrees from Radboud University Nijmegen, The Netherlands, and has spent several years as a postdoc at the University of California at Santa Cruz, USA.

Hichem Boudali received the Computer Science Engineering degree (MSc equivalent) from the Polytechnic School and Applied Sciences at the Universite' Libre de Bruxelles, Brussels, Belgium, in 2002, the M.Sc. degree in Electrical Engineering from the University of Virginia, and the Ph.D. degree in Computer Engineering from the School of Engineering and Applied Science at the University of Virginia, Charlottesville, in 2005. He is a member of the IEEE.

Mehmet Akşit holds an M.Sc. degree from the Eindhoven University of Technology and a Ph.D. degree from the University of Twente. Currently, he is working as a full professor at the Department of Computer Science, University of Twente, and affiliated with the Institute Center for Telematics and Information Technology.