Architectural Availability Analysis of Software Decomposition for Local Recovery*

Hichem Boudali[2], Hasan Sözer[1], Mariëlle Stoelinga[2]
[1]Software Engineering group, CS Dept., University of Twente, NL.
[2]Formal Methods and Tools group, CS Dept., University of Twente, NL.
E-mail: {hboudali,sozerh,marielle}@cs.utwente.nl

Abstract

Non-functional properties, such as timeliness, resource consumption and reliability are of crucial importance for today's software systems. Therefore, it is important to know the non-functional behavior before the system is put into operation. Preferably, such properties should be analyzed at design time, at an architectural level, so that changes can be made early in the system development process.

In this paper, 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 (i.e. a continuous-time Markov chain), from which various performance and dependability measures are then computed, in a way that is completely transparent to the user. 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.

*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).

1 Introduction

Local recovery is an important technique to achieve fault tolerance. Whereas a global recovery strategy restarts the whole system upon detection of an error, thus 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. Thus, a better availability is guaranteed: 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, heavily depends on the chosen software decomposition, i.e., the way in which the software modules are grouped into RUs. Since the implementation of a local recovery strategy is a time-consuming and a non-trivial task, it is important to have a quick, easy and accurate method that predicts the system availability of a given decomposition alternative at design time. In this way, we can compare various decomposition alternatives and only implement the best one.

This paper presents such a method: We take as input a software decomposition, together with failure and repair rates for each module. From this module decomposition, we generate, in a way that is completely transparent to the designer an analytical model, that is, a continuous-time Markov chain (CTMC). We then use standard CTMC analysis methods to compute the system availability; other performance and dependability measures, such as the average number of operational units, number of failures during the first hour of operation, can be obtained as well.

A key step in our framework is that we translate all architectural elements to a set of Input/Output Interactive Markov Chains (I/O-IMCs) [14, 2]. I/O-IMCs augment traditional CTMCs with discrete actions, thus enabling synchronization between them. They have been used success-

fully to analyze a wide range of applications [15, 4, 2] and enable powerful analysis methods. In particular, we employ their *compositional-aggregation* technique, which is crucial in avoiding a blow up in the underlying state space.

More specifically, we use the MIOA-syntax [16] to conveniently specify and generate: (i) for each software module contained in an RU, one I/O-IMC modeling the failure and recovery behavior of that module; (ii) for each RU, two I/O-IMCs that serve as interfaces between an RU and the recovery manager (RM)[1]; and (iii) 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 [3]) after each composition. This is precisely the compositional-aggregation technique mentioned before.

We have implemented our methodology using JAVA and the CADP toolset and integrated it into the FLORA framework [13, 21], which facilitates the decomposition and implementation of software architectures for local recovery. We have carried out our modeling and analysis 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.

Thus, 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) Implementation of this translation into the CADP tool set; in particular, the implementation involved a translation from MIOA syntax to CADP. (3) Integration of our methodology in FLORA [13, 21] as a part of the ArchStudio environment [7]. FLORA supports the implementation of local recovery for a given software decomposition and now, our analysis tool within ArchStudio offers a push button technology to predict the availability of the resulting system. (4) 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.

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

Related work. There are several modeling techniques to analyze and improve system availability. For instance, [5] presents a Markov model to compute the availability of a redundant distributed hardware/software system comprised of N hosts; [22] presents a 3-state semi-Markov model to analyze the impact of rejuvenation[2] on software availability and in [17], the authors use stochastic Petri nets to model and analyze fault-tolerant CORBA applications. In general however, these models are specified manually and/or the methodology lacks a comprehensive tool-support, making these models less practical to use.

As far as local-recovery strategies are concerned, The work in [6] improves system availability with local recovery techniques that is similar to ours. However, their evaluation techniques are different: whereas we predict the availability at design time, [6] uses heuristics to choose a decomposition alternative and evaluate it by running experiments on the actual implementation.

Organization of the paper. In Section 2, we introduce the local recovery concept. In Section 3, we present our modeling approach including the detailed I/O-IMC models used to model local recovery. In Section 4, we present a case study along with experimentations and a discussion of results. Finally, we conclude the paper and suggest future research directions in Section 5.

2 Local recovery

Recovery of errors is an essential step of fault tolerance [1]. 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 such propagation of errors, 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

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

the other RUs. In [10], 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 [9, 10]; the RM and CM can be composed of multiple components or they can all be implemented in a single component. The specific implementation is however not relevant for our methodology to estimate system availability.

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.

3 Modeling approach

The overall modeling/analysis procedure is divided into 4 steps:

1. The user inputs the software modules decomposition as a set partition, together with the MTTF and MTTR for each module.
2. An I/O-IMC model is automatically generated for each module, each RU[3], and the RM.
3. All the generated I/O-IMCs are automatically composed, in a fixed order, into a single I/O-IMC, describing the behavior of the whole system. During composition, the compositional-aggregation technique is used to efficiently generate the state-space.
4. The final I/O-IMC is automatically converted into a CTMC and 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. We make the following assumptions in our models:

1. The failure of any module within an RU causes the entire RU failure.
2. Errors of modules are independent and they do not propagate beyond the boundaries of RUs.
3. The recovery of an RU entails the restart of all its modules (even the ones that did not fail).
4. The failure of a module is governed by an exponential distribution (i.e. constant failure rate).
5. The CM is considered to a be part of the RM and as such it is not modeled separately.
6. The restart (or repair) of a module is governed by an exponential distribution[4]. The recovery time includes the time for restarting failed modules and also the time for error detection, error notification and diagnosis.
7. The RM always correctly detects a failing RU.
8. The RM does not fail.
9. Only one RU can be recovered at a time and the RM recovers the RUs on a first-come-first-served basis.
10. The recovery always succeeds.
11. The restart of the modules inside a given RU is sequential.

It is important to realize that these assumptions are geared to the applications we had considered (multimedia applications, like MPlayers and TVs), and that our framework enables one to easily change these assumptions (see Section 3.3).

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. Correspondingly, the failure interface 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

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

[4] 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.

recovery of the modules inside the RU. Each module, recovery interface, failure interface, and the RM has a corresponding I/O-IMC model. Fig. 1 illustrates the interaction between these different models.

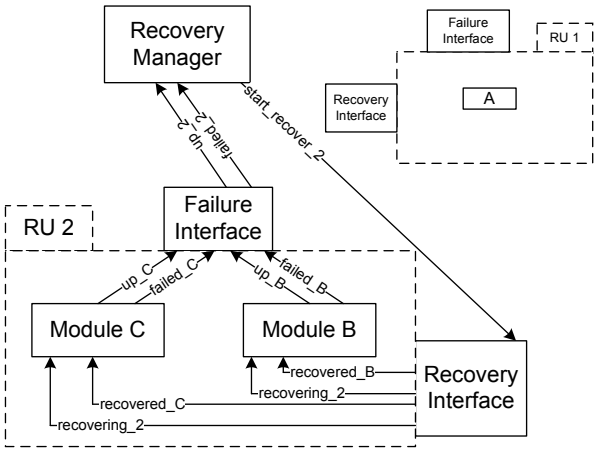


Figure 1. Interaction between the various I/O-IMC models. A dashed box indicates an RU boundary and a solid box indicates an I/O-IMC model.

3.1 The underlying I/O-IMC modeling formalism

Input/Output interactive Markov chains (I/O-IMC) [3, 2] is the underlying state-based modeling formalism we use. I/O-IMCs are a combination of Input/Output automata (I/O-automata) [19] and interactive Markov chains (IMCs) [14]. 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. Fig. 2 (taken from [2]) shows an I/O-IMC with two Markovian transitions: one from $S1$ to $S2$ with rate λ and another from $S3$ to $S4$ with rate μ. The I/O-IMC has one input action $a?$. To ensure input-enabling, we specify $a?$-self-loops in states $S3$, $S4$, and $S5$.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

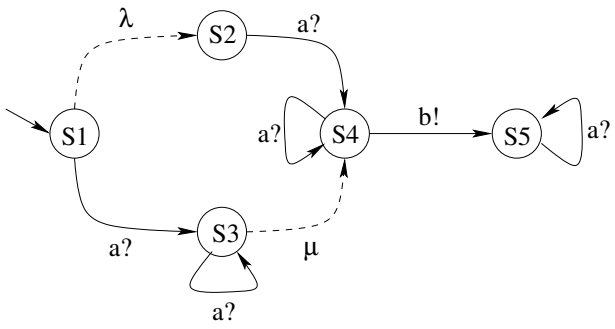


Figure 2. Example of an I/O-IMC.

either (1) they are both ready to accept the same input action or (2) one is ready to output an action $a!$ and the other is ready to receive that same action (i.e., has input action $a?$). I/O-IMCs can be combined with a parallel composition operator "||" 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 [3]).

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.e. same behavior) 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* [3] to aggregate I/O-IMCs.

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.

I/O-IMC model specification and generation We use a formal language called MIOA [16] to describe any I/O-IMC model. MIOA is based on the IOA language defined by N. Lynch et al. in [12]. 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. 4 states).

Due to the lack of space, we will not go into details of the MIOA language; however, as an example, we show the MIOA specification of the failure interface I/O-IMC model (Figure 3). Any MIOA specification is divided into 3 sections: *Signature* where input/output/internal signals and Markovian rates are specified, *States* where the states of the I/O-IMC are defined, and *Transitions* where the I/O-IMC transitions are defined in a precondition-effect style (i.e. in order for the transition to take place, the precondition, which is a true/false expression, has to hold).

In Fig. 3, the signature consists of the failed/up signals of the n modules belonging to the RU, one output signal of the RU 'failed' signal, and one output signal of the RU 'up' signal. The states of the failure interface I/O-IMC are defined 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 and 'rufailed' being false. There are four kind of possible transitions; for example, the last transition indicates that an RU 'up' signal 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.

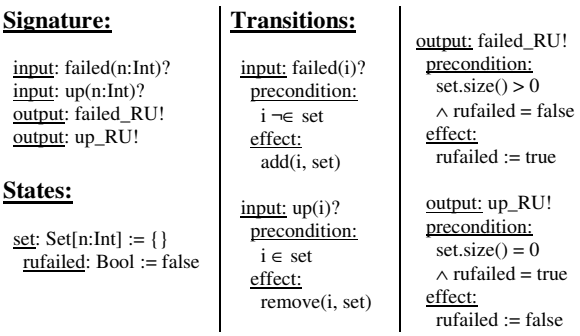


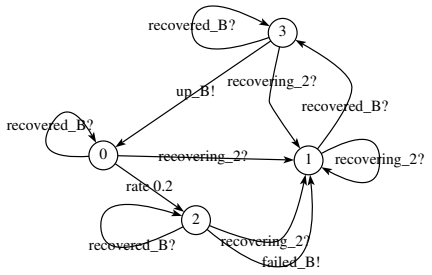
Figure 3. MIOA specification of the failure interface I/O-IMC model.

3.2 I/O-IMC models for local recovery

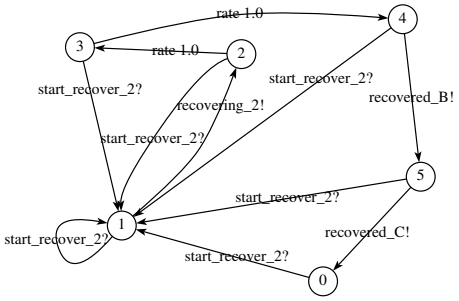
In this section, we provide details on the 4 basic I/O-IMC models used in our framework, namely the module, the failure interface, the recovery interface, and the recovery manager. The running example (fig. 1) consists of two RUs, RU 1 has one module A and RU 2 has two modules B and C, and a recovery 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. 4(a) 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. In state 2, the module notifies the failure interface of RU 2 about its failure (i.e. transition 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 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. Fig. 5 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.



(a) The module I/O-IMC.



(b) The recovery interface I/O-IMC.

Figure 4. Module and recovery interface I/O-IMCs.

The recovery interface I/O-IMC. Fig. 4(b) shows the I/O-IMC model of RU 2 recovery interface. The recovery interface receives a 'start_recover' signal from the RM (transition 0 to 1), allowing it to start the RU's recovery. A 'recovering' signal is then output (transition 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 (transitions 2 to 3 and 3 to 4), followed by two sequential 'recovered' notifications (transitions 4 to 5 and 5 to 0).

The recovery manager I/O-IMC. Figure 6 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 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. In queuing theory literature, this is referred to as a first-come-first-served (FCFS) policy. 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 failed first, it is granted the 'start_recover' signal (transition 4 to 7), the RM then awaits for RU 1 'up' signal, and once received, RM grants the 'start_recover' signal to

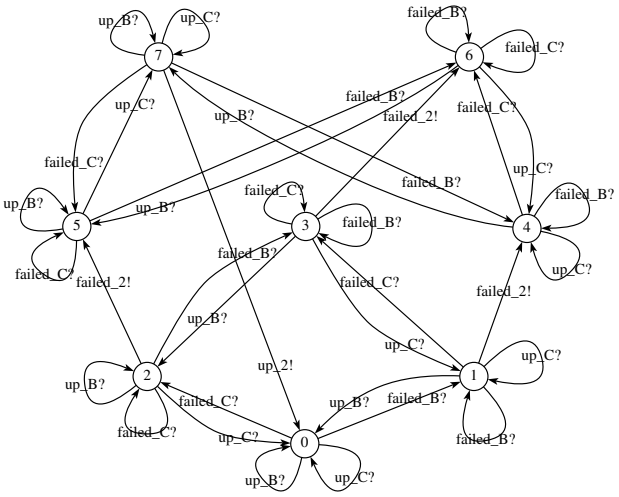


Figure 5. The failure interface I/O-IMC model.

RU 2 (as RU 2 is still in the queue of failing RUs) (transition 2 to 6). Finally, the RM receives 'up_2' and both RUs are operational again.

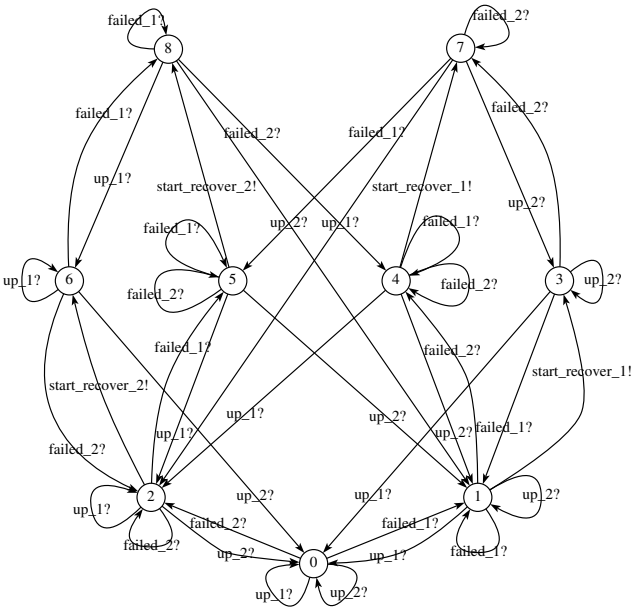


Figure 6. The recovery manager I/O-IMC model.

3.3 I/O-IMC modeling flexibility

Note that any of the four basic I/O-IMC models presented above can be, to a certain extent, locally modified without affecting the remaining models. Indeed, this modeling flexibility/extensibility and modularity is a powerful

feature of the I/O-IMC formalism [4, 2].

In particular, any of the four I/O-IMC models can be individually and easily altered to reflect the correct system behavior, e.g. one can change the recovery strategy of the RM if the FCFS strategy turns out to be not suitable. 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.

3.4 Implementation Details

We have automated the whole modeling and analysis procedure with a Java program. The program takes as input the modules decomposition, which is simply specified as a set partition. For example, the decomposition with three modules and two RUs, as shown in fig. 1, is specified as { [A(mttr,mttf)] [B(mttf,mttr), C(mttf,mttr)]}, where mttf and mttr are values of the module's MTTF and MTTR. Note that MTTF(MTTR) is the reciprocal of the failure(repair) rate.

All the modules and RUs (i.e. partitions) in the specification are mapped to MIOA specifications, which are used by an algorithm to explore the state-space and generate the corresponding I/O-IMCs. Based on the specified decomposition, the program also generates the corresponding I/O-IMC for the RM. All the generated I/O-IMC models are output in the Aldebaran *.aut* file format which can be processed with the CADP tool-set [11]. In addition to generating all the necessary I/O-IMCs, a composition/analysis script is also generated. This script conforms to the CADP SVL scripting language. The details about the generation of the I/O-IMC modes and the composition/analysis script can be found in [21].

After the generation step, we run the SVL script, within CADP, which composes/aggregates all the I/O-IMCs based on the modules decomposition, reduces the final I/O-IMC into a CTMC, and computes the measure of interest (i.e. the steady-state availability in this case).

4 Case study

4.1 MPlayer

We have applied local recovery to an open-source software, MPlayer [18]. 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 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).

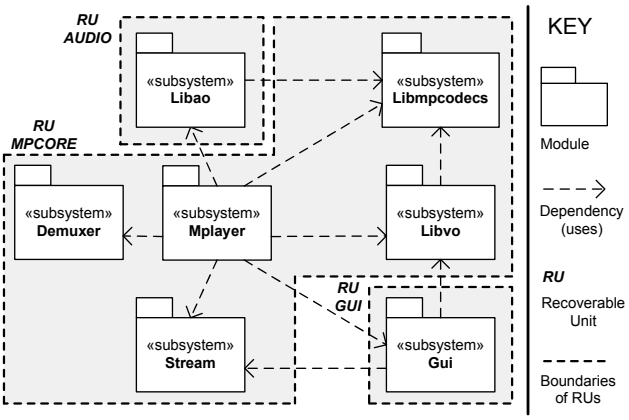


Figure 7. The modules decomposition view of the MPlayer software architecture.

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. 7. In this example, the system is partitioned into 3 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.

4.2 The FLORA framework

FLORA [13] is a framework that supports the decomposition and implementation of software architecture for local recovery. It partitions system modules as defined by RUs and isolates these modules by assigning each RU to a separate process[5]. In addition to the specified RUs, FLORA introduces a CM[6] 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.We have used FLORA for introducing local recovery to MPlayer for several decomposition alternatives.

[5]Interaction between the RUs are redirected through Inter-Process Communication.

[6]Modeled as part of the RM as mentioned in Section 3.

4.3 Experimentation and analysis

We have implemented local recovery for a total of 3 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). After a module is initialized, it creates a thread that is periodically activated every second to inject errors. The operation of the thread is shown in Algorithm 1.

Algorithm 1 Periodically activated thread for error injection

```
1:  time_init ← currentTime()
2:  while TRUE do
3:      time_elapsed ← currentTime() − time_init
4:      p ← 1 − 1/e^(time_elapsed/MTTF)
5:      r ← random()
6:      if p ≥ r then
7:          injectError()
8:          break
9:      end if
10: end while
```

The error 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 inject an error (Line 6). Possibly an error is injected by basically creating a fatal error with an illegal memory operation. This error crashes the process, on which the module is running (Line 7).

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 as the value $\frac{T_{avail}}{5}$. The results of the measured system availability are shown in Table 2 for the different alternatives. Table 2 also shows the estimated system availability based on the analytical models as described in Section 3. We have used the MTTF values shown in Table 1 both for the analytical models and the error 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.

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

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

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	N/A	99.96

For the sake of comparison, the last row shows the estimated availability for the decomposition alternative, where each module of the MPlayer is placed in a different RU (i.e. 7 RUs in total). This decomposition isolates all the modules from each other and hence, 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. Implementing local recovery for this decomposition alternative (and for any decomposition in general) is a time-consuming procedure; however, using the analytical approach we can easily and quickly get an estimated availability of such an alternative.

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. In fact, the various communications between the software modules, which are modeled using interactive transitions in the I/O-IMC models, abstract away any communicaion time delay (i.e. the communication is instantaneous). 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.

5 Conclusion and future work

Local recovery is applied to achieve higher system availability and its effectiveness highly depends on the implemented software decomposition. In this paper, we have presented an easy-to-use methodology 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 estimate 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.

We have used our I/O-IMC methodology with the FLORA approach by integrating it into the ArchStudio environment, which is based on xADL [7]. One can also integrate (given some adjustments) this methodology into other modeling/analysis frameworks, such as the UML [20] or the AADL [8] modeling formalisms, to support certain dependability/performance analyses.

As for future directions, one can revisit some or all of the assumptions made in Section 3 and/or modify any of the four basic I/O-IMC models to more accurately represent the real system behavior.

Acknowledgement: We thank Pepijn Crouzen for his help using CADP and Boudewijn Haverkort for his comments on an earlier version of this paper. We also thank the anonymous reviewers for their feedback to improve this paper.

References

[1] A. Avizienis et al. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, 2004.

[2] H. Boudali, P. Crouzen, B. R. Haverkort, M. Kuntz, and M. Stoelinga. Architectural dependability evaluation with arcade. In *Proc. of the 38th IEEE/IFIP Inter. Conf. on Dependable Systems and Networks*, pages 512–521. IEEE, 2008.

[3] H. Boudali, P. Crouzen, and M. Stoelinga. A compositional semantics for Dynamic Fault Trees in terms of Interactive Markov Chains. In *ATVA'07*, pages 441–456. LNCS, 2007.

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

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

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

[7] E. Dashofy, A. van der Hoek, and R. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *ICSE'02*, pages 266–276, Orlando, Florida, 2002. ACM.

[8] P. Feiler, D. Gluch, and J. Hudak. The Architecture Analysis and Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-011, SEI, 2006.

[9] G. C. Hunt et al. Sealing OS processes to improve dependability and safety. *SIGOPS Oper. Syst. Rev.*, 41(3):341–354, 2007.

[10] G. Candea et al. Microreboot: A technique for cheap recovery. In *OSDI'04*, pages 31–44, 2004.

[11] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *CAV'07*, volume 4590 of *LNCS*, pages 158–163. Springer-Verlag, 2007.

[12] S. Garland, N. Lynch, J. Tauber, and M. Vaziri. IOA user guide and reference manual. Technical report, MIT CSAI Laboratory, Cambridge, MA, 2004.

[13] H. Sozer and B. Tekinerdogan. Introducing recovery style for modeling and analyzing system recovery. In *WICSA'08*, pages 167–176, 2008.

[14] H. Hermanns. *Interactive Markov Chains*, volume 2428 of *LNCS*. 2002.

[15] H. Hermanns and J. P. Katoen. Automated compositional Markov chain generation for a plain-old telephone system. *Sci. of Comp. Programming*, 36(1):97–127, 2000.

[16] G. W. M. Kuntz and B. R. H. M. Haverkort. Formal dependability engineering with mioa. Technical Report TR-CTIT-08-39, June 2008.

[17] I. Majzik and G. Huszerl. Towards dependability modeling of FT-CORBA architectures. In *EDCC'02*, pages 121–139. LNCS, 2002.

[18] MPlayer official website, 2008. http://www.mplayerhq.hu/.

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

[20] J. Rumbaugh, I. Jacobson, and G. Booch, editors. *The Unified Modeling Language (UML) reference manual*. Addison-Wesley Longman Ltd., Essex, UK, 1999.

[21] H. Sozer. *Architecting Fault-Tolerant Software Systems*. PhD thesis, University of Twente, 2009.

[22] K. Vaidyanathan and K. S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Trans. on Dependable and Secure Computing*, 2(2):124–137, 2005.