EXTENSION D'UN CADRE DE COMPOSITION DE COMPORTEMENTS EN PRÉSENCE DE PANNES À L'AIDE DE TECHNIQUES DE REPRISE ET DE AKKA

EXTENSION OF THE BEHAVIOR COMPOSITION FRAMEWORK IN PRESENCE OF FAILURES USING RECOVERY TECHNIQUES AND AKKA

par

Mohammad Hosnidokht

Mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, March 8, 2018

Le 8 mars 2018

le jury a accepté le mémoire de Monsieur Mohammad Hosnidokht
dans sa version finale.

Members du jury

Professeur Richard St-Denis
Directeur de recherche
Département d'informatique

Professeur Gabriel Girard
Évaluateur interne
Département d'informatique

Professeur Bessam Abdulrazak
Président rapporteur
Département d'informatique

Sommaire

La tolérance aux fautes est une propriété indispensable à satisfaire dans la composition de services, mais atteindre un haut de niveau de tolérance aux fautes représente un défi majeur. Dans l'ère de l'informatique ubiquitaire, la composition de services est inévitable lorsque qu'une requête ne peut être réalisée par un seul service, mais par la combinaison de plusieurs services. Ce mémoire étudie la tolérance aux fautes dans le contexte d'un cadre général de composition de comportements (*behavior composition framework* en anglais). Cette approche soulève, tout d'abord, le problème de la synthèse de contrôleurs (ou compositions) de façon à coordonner un ensemble de services disponibles afin de réaliser un nouveau service, le service cible et, ensuite, celui de l'exploitation de l'ensemble des compositions afin de rendre le nouveau service tolérant aux fautes. Bien qu'une solution ait été proposée par les auteurs de ce cadre de composition, elle est incomplète et elle n'a pas été évaluée expérimentalement ou *in situ*. Ce mémoire apporte deux contributions à ce problème. D'une part, il considère le cas dans lequel le service visé par le contrôleur est temporairement ou définitivement non disponible en exploitant des techniques de reprise afin d'identifier un état cohérent du système à partir duquel il peut progresser en utilisant d'autres services ou de le laisser dans un état cohérent lorsqu'aucun service, parmi ceux disponibles, ne permet plus de progression. D'autre part, il évalue plusieurs solutions de reprise, chacune utile dans des situations particulières de pannes, à l'aide d'une étude de cas implémentée en AKKA, un outil qui permet aisément de mettre en oeuvre des systèmes réactifs, concurrents et répartis.

Abstract

Fault tolerance is an essential property to be satisfied in the composition of services, but reaching a high level of fault tolerance remains a challenge. In the area of ubiquitous computing, the composition of services is inevitable when a request cannot be carried out by a single service, but by a combination of several services. This thesis studies fault tolerance in the context of a general behavior composition framework. This approach raises, first, the problem of the synthesis of controllers (or compositions) in order to coordinate a set of available services to achieve a new service, the target service and, second, the exploitation of all compositions to make the new service fault tolerant. Although a solution has been proposed by the authors of the behavior composition framework, it is incomplete and has not been evaluated experimentally or *in situ*. This thesis brings two contributions to this problem. On one hand, it considers the case in which the service selected by the controller is temporarily or permanently unavailable by exploiting recovery techniques to identify a consistent state of the system from which it may progress using other services or leave it in a coherent state when none of the available services no longer allows progression. On the other hand, it evaluates several recovery solutions, each useful in services malfunction situations, using a case study implemented with the aid of Akka, a tool that facilitates the development of reactive, concurrent and distributed systems.

Remerciements

I would like to thank my thesis advisor Prof. Richard St-Denis at Université de Sherbrooke for his continuous support in my master study and research, his patience, motivation, enthusiasm, and immense knowledge. He consistently steered me in the right direction whenever he thought I needed it. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my M.Sc. study.

Contents

Sommaire iii

Abstract iv

Remerciements v

Contents vi

List of Figures ix

Introduction 1

Context . . . 1

Problem . . . 3

Objectives . . . 4

Methodology . . . 5

Expected Results . . . 7

Organization . . . 8

1 Preliminaries 9

1.1 Behavior Composition Framework . . . 10

1.1.1 Behavior Composition Problem . . . 12

1.1.2 Controller Synthesis . . . 13

1.1.3 Synthesis Based on an ND-simulation Relation . . . 13

1.1.4 Fault Tolerance in the Behavior Composition Framework . . . 14

1.2 Fault Tolerance . . . 16

1.2.1 Dependability Attributes . . . 16

1.2.2 Dependability Concerns . . . 18

1.2.3 Fault and Failure Classifications . . . 20

1.2.4 Fault Tolerance Techniques . . . 20

2 Review of Fault Tolerance Terminologies and Technologies 25

2.1 Stability Patterns in Fault Tolerance . . . 25

2.1.1 Circuit Breaker . . . 26

2.1.2 Bulkheads . . . 27

2.1.3 Fail Fast . . . 29
2.2 Libraries and a Toolkit for Fault Tolerance Applications . . . 30
2.2.1 Apache Ignite . . . 30
2.2.2 Failsafe . . . 32
2.2.3 Hystrix . . . 33
2.2.4 JRugged . . . 36
2.2.5 Resilience4j . . . 37
2.2.6 Akka . . . 37
2.3 Approaches for Fault Tolerance Applications . . . 42
2.3.1 Forward Recovery . . . 42
2.3.2 Backward Recovery . . . 43
2.4 Conclusion . . . 45

3 Proposed Approach: Integration of Backward Recovery into the Behavior Composition Framework 46
3.1 Colored Petri Net and Compensation Flow . . . 46
3.2 A Case Study . . . 55
3.3 Contribution . . . 57

4 Implementation of Forward and Backward Recovery in the Behavior Composition Framework 60
4.1 Assumption . . . 61
4.2 Overall Architecture . . . 62
4.3 Details about Different Phases in the Implementation . . . 64
4.3.1 The Startup Phase . . . 65
4.3.2 The Delegation Phase . . . 69
4.3.3 The Recovery Phase . . . 72
4.4 Experiments . . . 74
4.5 Generalization of the Implementation . . . 80

Conclusion 83

Appendix A 84
A.1 SMV Code for the Case Study . . . 84
A.2 TLV Output—The Controller Generator . . . 86

A.3 Transitions.json—The Controller Generator in the JSON Format . . . 91

Appendix B **97**

B.1 Constant Names 97

B.2 Constant Actions 97

B.3 Target and Behaviors States 97

B.4 Constant Behavior Names 98

B.5 Constant Message Objects and Classes 98

Appendix C **99**

C.1 The `receive` Method of an Orchestrator 99

Appendix D **102**

D.1 Log Messages 102

Appendix E **105**

E.1 Log Messages for the Normal Execution 105

Bibliography **107**

List of Figures

1.1 Main elements of the behavior composition framework 10
1.2 Monolithic architecture 18
1.3 Independent service architecture 19
1.4 Fault tolerance techniques 21
1.5 Active replication 22
1.6 Passive replication 23

2.1 Circuit breaker behavior 27
2.2 Single thread pool 28
2.3 Bulkhead structured thread pool 29
2.4 Apache Ignite integrated components 31
2.5 At least once guarantee failover in Apache Ignite 32
2.6 Closure failover in Apache Ignite 33
2.7 Service composition without Hystrix 34
2.8 Service composition with Hystrix wrapper 35
2.9 Actor lifecycle 39
2.10 Membership lifecycle 41
2.11 Description logic and satisfiability checkers relation 45

3.1 Marked CPN–TCWS$_Q$ when ws_4 fails 50
3.2 BRCPN–TCWS$_Q$ 51
3.3 Initial marking of BRCPN–TCWS$_Q$ 51
3.4 New marked BRCPN–TCWS$_Q$ 52
3.5 Executer architecture 53
3.6 Target behavior β_T 55
3.7 Available behaviors β_1, β_2 and β_3 56
3.8 Controller generator for the case study 57
3.9 Controller generator without β_2 58
3.10 Controller generator without β_3 59

List of Figures

4.1 Overall architecture 61
4.2 Actor maiblox 62
4.3 The sequence diagram for the service composition 63

Introduction

Fault tolerance is the ability of a system to function correctly despite the occurrence of faults in software or hardware. It is highly needed in most enterprise organizations especially when life-critical systems must continue to provide services in the case of system faults. Faults caused by errors (or bugs) are systematic and can be reproduced in the right conditions. Of course, it is not possible to tolerate every fault but, fault-tolerant programs are required for applications where it is essential that faults do not cause a program to have unpredictable execution behavior. The importance of implementing a fault-tolerant system is about service continuity or maintaining functionality in the case of system failures. The current software engineering practices tend to capture only normal behavior, assuming that all faults can be removed during development, but they do not guarantee the absence of errors. Formal methods can be used to address the problem of errors in program behaviors and provide means of making a rigorous, additional check.

It is assumed that failures do not arise from design faults in the program. Design faults occur when a designer (programmer), either misunderstands a specification or simply makes a mistake. So, the current work deals with service-level faults (i.e., faults due to unsuccessful operations in services and fault occurrences in service communication and functionality which can be captured by fault handlers of processes). It is important to note that normal behavior does not mean perfect behavior. For instance, after a time-out occurs, if the communication channel repeatedly sends the same message, the retransmission of a message by a sender is normal, but it may result in two copies of the same message reaching its destination and causes a catastrophic failure.

Context

Writing correct and fault-tolerant applications from scratch is too hard. For this purpose several ad-hoc frameworks and applications, such as Akka [23] which is a resilient elastic[1] distributed real-time transaction processing toolkit, have been introduced to prevent applications from trying to perform an action that is doomed to fail. AKKA uses supervisor hierarchies with let-it-crash semantics. It detects failures and encapsulates logic for preventing a failure to reoccur constantly. It stops cascading failures and improves the system's overall resiliency and fault tolerance in complex distributed systems where failures are inevitable. Although, it is one of the most popular toolkits to implement concepts related to fault tolerance, different test scenarios are needed to verify the final system functionality. It is possible to verify system's properties or potential failures in a more thorough fashion than empirical testing by using formal methods. While current studies on fault tolerance have mostly focused — from the technical viewpoint — on standards, protocols and different ad-hoc tools, formal methods can greatly increase our understanding of a system by revealing inconsistencies, ambiguities and incompletenesses that might otherwise go undetected [15]. Formal methods use mathematical tools as a complement to system testing in order to ensure correct behavior. Model checkers take as input a formal model of the system, typically described by means of state machines or transition systems, and verify if it satisfies temporal logic properties [14]. Model checking techniques are supported by tools, which facilitate their application. In case the model checker detects a violation of a desired property, a counterexample is produced to show how the system reaches the erroneous state. As systems become more complicated, and safety becomes a more important issue, a formal approach to system design offers another level of quality. Formal methods differ from other design systems through the use of formal verification schemes or interactive provers to establish the correctness of systems before they are accepted [9].

In recent years, the topic of composing behaviors has been proposed in the areas of web service [7], verification [28] and even multi-agent systems [34, 35]. Among recent studies that can either be used in service oriented computing, different formal methods, most of them with a semantics based on transition systems (e.g., automata, Petri nets, process algebras), have been used to guarantee correct behavior and service

1. Elasticity adapts to both the "workload increase" as well as "workload decrease" by "provisioning and deprovisioning" resources in an "autonomic" manner, unlike scalability, which adapts only to the "workload increase" by "provisioning" the resources in an "incremental" manner.

composition [18, 2, 29]. One promising synthesis-based model, that acts as a formal tool for service composition, is the *behavior composition framework* [19], which is quite significant, as the authors have extensively investigated the problem of behavior composition. It generally synthesizes a controller to delegate actions to suitable available behaviors. However, the behavior composition framework also raises a number of challenges in presence of failures, which is discussed in the next section.

Problem

Service composition viewed as cohesive and loosely coupled services, which should interact with each other to accomplish a specific task, makes easier modification and addition of functions and qualities to a system anytime [39]. The ability to compose services to create new services is an essential part in real world applications. However, this ability introduces not only advantages but also new challenges, most importantly in presence of failures. Indeed, a runtime failure or unavailability of a service may result in a failed execution of a composite service.

In the context of the behavior composition framework, service composition consists in combining a set of available behaviors (e.g, services) to achieve a target behavior (composite service). The overall aim of the system is to perform the actions (e.g, operations) sequentially to realize the target. The behavior composition framework includes a solution to cope with the composition problem, which consists in automatically generating compositions (or generated controllers) from an environment, available behaviors and a target behavior. This phase is a planning phase. It is based on the notion of simulation. First, a system is defined as a collection of available behaviors that operate on a common environment. Second, the enacted system behavior and enacted target behavior are made. The enacted system behavior results from the synchronous product of the environment and asynchronous product of available behaviors. Likewise, the enacted target behavior describes the evolution of the target behavior acting on the environment. Third, a simulation-based approach is exploited to generate compositions. The latter are at the end of a synthesis process, which starts with the calculation of the largest nondeterministic (ND) simulation of the enacted target behavior by the enacted system behavior. Having constructed such elements, a generated controller is introduced to select, at each step of execution, a suitable behavior, which will carry out the current required action. Several problems may, however occur due to the failure of a behavior. A behavior may temporarily stop responding or unexpectedly change its current state. The environment may unexpectedly change its current state. A behavior may become permanently unavailable and unexpectedly resume its operation after starting in a certain state.

Currently, some efforts have been made to solve the aforementioned problems. Instead of re-planning from scratch for a whole behavior or service [30, 24], an alternative approach was proposed in which behaviors are iteratively removed or added at run–time. If during an iteration no suitable choices are possible, then the controller should wait for the behavior to come back. Roughly speaking, this alternative approach exploits all compositions at runtime and deals with unexpected failures by suitably refining the solution on the fly, thus avoiding full re-planning [18]. This approach may improve the system resiliency, but there is still an obstacle to provide consistency in the system. If the behavior never comes back, a composite service may terminate in an inconsistent state and the reliability and availability of the system is violated.

So, in order to construct highly available, robust and reliable system, it is a necessity to evaluate and extend different recovery techniques and algorithms in the context of the behavior composition framework.

Objectives

This proposal investigates on the following items, especially for behavior composition in many situations and domains in which assuming full reliability and availability of behaviors is not adequate to guarantee the correct functionality of a system.

- Evaluate recovery techniques in the context of the behavior composition framework.

  Recovery techniques, namely *forward and backward recovery* techniques, are fault-tolerant execution control mechanisms based on colored Petri nets, which include a replacement and a compensation process respectively [13]. In this approach, a transactional property of each node allows to recover the system in case of failures during the execution. In a fault-tolerant service composition, each set of related services may form a transaction for which the atomicity property is a must and by failing one of them, others have to be rolled-back. However, for some services, the roll-back operation may not be available or only available partially. Therefore, failing a service may cause the whole composition ended up in an inconsistent execution due to the violation of atomicity property and results in an unreliable system.

- Evaluate the Akka fault tolerance strategies appropriate for the behavior composition framework.

  Only relatively recently actor frameworks have became mature enough to be adopted as a mainstream ad–hoc technology for building complex distributed systems and handle failures, an example being the Akka framework [23]. Compared to traditional distributed system architectures, actor frameworks present a considerably different approach to concurrency, state persistence and fault tolerance. So, using the Akka fault tolerance strategies in the context of the behavior composition framework gives the ability to activate, stop, restart and resume any of available behaviors and to instruct them to execute an operation with respect to their current state. Moreover, Akka has a supervisor strategy with full observability on available actors which can keep track (at runtime) of their current states.

- Integrate a formal solution based on colored Petri nets and supervisor strategy into the behavior composition framework.

The intentions are twofold. First, propose an approach to fault tolerance in the behavior composition framework which is based on state transition systems. Second, implement a case study in Akka using built–in strategies, behavior composition with the largest ND-simulation algorithm and recovery techniques.

Methodology

In the literature, fault-tolerant behavior composition, multiple recovery techniques and the actor model [25] with many implementations in different programming languages have been proposed to cope with some of the challenges discussed earlier.

In this work, particular attention is put on supervision and recovery techniques that exploit functions already present in the system in a different than usual way to achieve reliability with little or no intervention at the application or syntax levels.

In general, to make a system more reliable, it is important to always view a behavior as part of a supervision hierarchy. At this point it is vital to understand that supervision is about forming a recursive fault handling structure. If there is too much to do in the behavior, it will become hard to make it robust and fault tolerant, hence the recommended way in this case is to add a level of supervision. A supervisor gives the ability to activate, stop, restart and resume a behavior. In the case of the behavior composition framework, it should be noted that the supervision hierarchy has only one level.

A behavior is a container for state and actions. The supervisor has full observability on the available behaviors, that is, it can keep track (at runtime) of their current states. The supervisor must respond to subordinate failure. When a subordinate detects a failure (i.e., throws an exception), it suspends itself and all its subordinates (if any), and sends a message to its supervisor, signaling failure. Depending on the nature of the work to be supervised and the nature of the failure, the supervisor has a choice of the following four options:

- resume the subordinate, keeping its accumulated internal state;
- restart the subordinate, clearing out its accumulated internal state;
- stop the subordinate permanently;
- escalate the failure, thereby failing itself.

Each supervisor is configured with a function translating all possible failure causes (i.e., exceptions) into one of the four choices given above.

As mentioned earlier, there is a formal approach to deal with unexpected failures in the behavior composition framework, so by adding a level of supervision and integrating backward recovery into the framework, there is a capability to overcome drawbacks of the behavior composition framework and thus improve the reliability of composite services. More specifically, backward recovery is summarised in restoring the state that the system has at the beginning of the transaction. So, all the successfully executed actions, before the failure, must be compensated to undo their produced effects. Therefore, the expected recovery approach should have two techniques to cope with service failures. The forward recovery technique proposes an algorithm to find a substitution for a failed service in order to continue the execution and the backward recovery technique which is aimed to rollback all system transactions and leaves the system in a consistent state.

A fault-tolerant service composition is the one that ends up the whole transaction in a safe state upon a service failure, where the related services are also rolled-back appropriately.

Expected Results

In order to practically validate our research, the proposed methodology is adopted and illustrated with a case study. There are three service behaviors and one target behavior in the case study and each behavior includes of several actions.

Several failure cases are anticipated and the behaviors are expected to tolerate these cases, namely:

> A behavior temporarily freezes, that is, it stops responding and remains still, then eventually resumes in the same state it was in. As a result, while frozen, the controller cannot delegate actions to it.
>
> A behavior that was temporarily freezes unexpectedly resumes operation starting in a certain state. The controller can exploit such an opportunity and start delegating actions to it again.
>
> A behavior dies, that is, it becomes permanently unavailable. The controller has to completely stop delegating actions to it.
>
> A behavior throw an exception. The controller has to stop, resume or restart the behavior depending on the type of exception.

An interactive command line interface is provided to test the functionality of the system. Killing and freezing the behavior permanently result in removing the behavior and related transitions from the controller generator and stop delegating actions to the failed behavior. Afterwards, the controller generator is adjusted. So, for any further requests if there is no choice in the available behaviors, instead of waiting for a failed behavior to comes back, the backward recovery will be started and all executed actions will be rolled back to leave the system in a consistent state.

Notably, in the case of unfreezing the behaviors in any state, the controller generator is altered again and a joined behavior is considered again for action delegations. Moreover, a separate supervisor is assigned to each behavior and it is expected to handle the behavior internal exceptions. If a behavior throws an exception, the supervisor depending on the nature of the work to be supervised and the nature of the exception, has a choice of the following four options:

resume the subordinate, keeping its accumulated internal state;

restart the subordinate, clearing out its accumulated internal state;

stop the subordinate permanently;

escalate the failure, thereby failing itself.

Organization

The remainder of this thesis is structured as follows. Chapter 1 introduces a set of preliminaries that are necessary to understand the behavior composition framework and fault tolerance. Chapter 2 provides a review of fault tolerance terminologies and technologies with a set of fault-tolerant libraries and the AKKA toolkit. These two chapters cover work in the field of formalisms for composition of web services and different approaches to fault tolerance found in the literature. Chapter 3 describes the integration of backward recovery into the behavior composition framework to leave the system in a consistent state after a failure. A case study is introduced to illustrate this approach. Chapter 4 details an implementation and provides an illustration with the case study.

Chapter 1

Preliminaries

In order to understand the proposed approach, which is based on some theoretical formalisms, it is necessary to introduce some basics about transition systems and behaviors. Thus, some mathematical definitions specific to behavior composition, some concepts and techniques related to fault tolerance and an introduction of a toolkit, called Akka, are presented hereafter.

Formally, a transition system is a pair $\langle S, \rightarrow \rangle$ where S is a set of states and $\rightarrow$ is a set of state transitions (i.e., a subset of $S \times S$). The fact that there is a transition from state p to state q (i.e., $\langle p, q \rangle \in \rightarrow$), is written as $p \rightarrow q$.

A labelled transition system is a tuple $\langle S, \nabla, \rightarrow \rangle$, where S is a set of states, ∇ is a set of labels and $\rightarrow$ is a set of labelled transitions (i.e., a subset of $S \times \nabla \times S$). The fact that $\langle p, \alpha, q \rangle \in \rightarrow$ is written as $p \xrightarrow{\alpha} q$. This represents the fact that there is a transition from state p to state q with label α. If, for any given p and α, there exists at most one tuple $\langle p, \alpha, q \rangle \in \rightarrow$, then the transition system is deterministic.

A behavior β is a tuple $\langle B, A, b_0, \eta \rangle$ where:

- B is the finite set of behavior's states;
- A is a set of actions;
- $b_0 \in B$ is the initial state;
- $\eta \subseteq B \times A \times B$ is the transition relation, where $\langle b, a, b' \rangle \in \eta$, or $b \xrightarrow{a} b'$ in β, denotes that action a executed in state b may lead the behavior to successor state b'.

As a finite-state transition system, a behavior stands for the operational model of a system or a device. For example, in a flight reservation scenario, behaviors can represent different flight agencies providing a set of applicable actions such as hotel, taxi, meal and flight ticket reservations. The behavior involves two different types. The first is called partially controllable behavior, which is non-deterministic, without any knowledge about the next state after the execution of an action. The second one

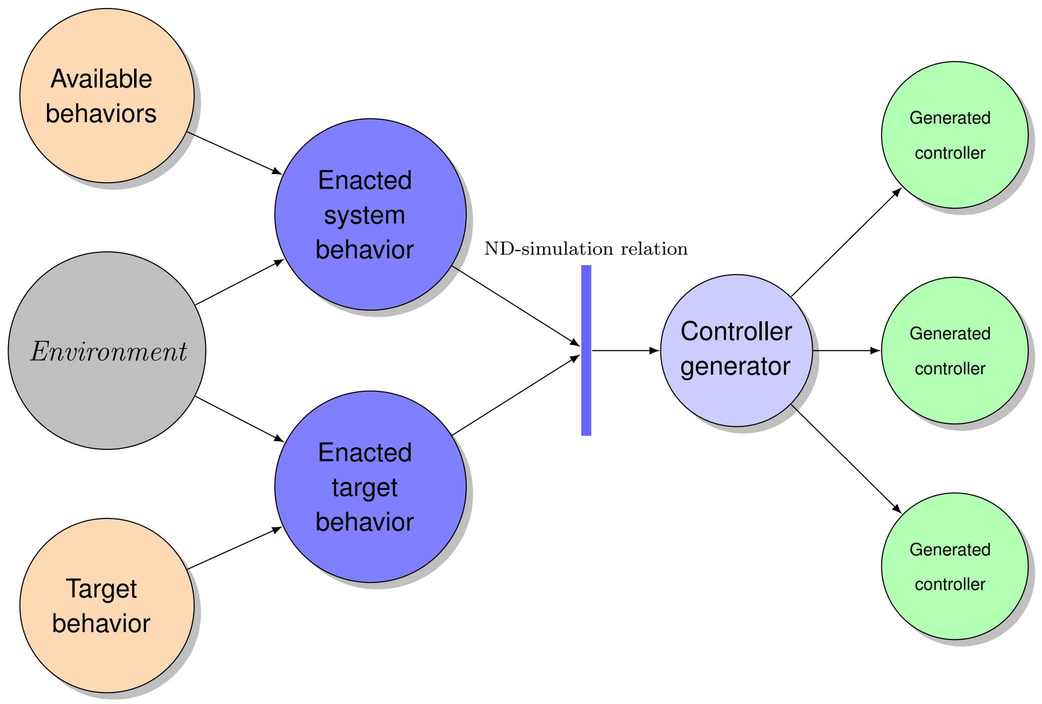

Figure 1.1: Main elements of the behavior composition framework

is called fully controllable behavior, which satisfies the following constraint. There is no state $b \in B$ and action $a \in A$ for which there exist two transitions $b \xrightarrow{a} b'$ and $b \xrightarrow{a} b''$ in β with $b' \neq b''$.

1.1 Behavior Composition Framework

The behavior composition framework is used as a formal tool with the aim of composing behaviors [19]. A general view of the behavior composition framework is illustrated in Figure 1.1. The description of the framework is borrowed from [6].

The main elements of the framework are an environment, available behaviors, a target behavior, a controller generator and generated controllers. In this framework, each behavior is an abstract model of an agent, device or software component operating on an environment, which is a shared space where actions are defined.

Definition 1.1.1. *An environment, which is generally nondeterministic, is a tuple*

$\mathcal{E} = \langle A, E, e_0, \rho \rangle$, where A is a finite set of shared actions, E is the finite set of environment states, $e_0 \in E$ is the environment initial state and $\rho \subseteq E \times A \times E$ is the environment transition relation.

Definition 1.1.2. *An available behavior, which is generally nondeterministic, is a tuple $\mathcal{B}_i = \langle B_i, \delta_i, b_{i0}, G_i, F_i \rangle$, where B_i is the finite set of behavior states, $\delta_i \subseteq B_i \times G_i \times A \times B_i$ is the behavior transition relation, $b_{i0} \in B_i$ is the behavior initial state, G_i is a set of guards on an environment $\mathcal{E}$ with a set of shared actions A and $F_i \subseteq B_i$ is the set of behavior final states.*

Similar to Definition 1.1.2, a target behavior $\mathcal{B}_t$ is a tuple $\langle B_t, \delta_t, b_{t0}, G_t, F_t \rangle$, whereas, on the contrary to an available behavior, $\mathcal{B}_t$ is deterministic. A target behavior indicates the fully controllable desired behavior to be reached.

It should be noted that, a guard over $\mathcal{E}$ is a Boolean function $g : E \mapsto \{\top, \bot\}$. This means that the behaviors evolve with respect to the current state of $\mathcal{E}$.

Given the available behaviors and an environment, a system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ is defined as the interleaving (composition) of all available behaviors being able to operate over the shared environment.

In the case that behaviors cannot function in a standalone way, their real capabilities depend on both themselves and the environment operating on them. So, from this point, the notion of enacted behavior is defined.

Definition 1.1.3. *Given a behavior $\mathcal{B} = \langle B, \delta, b_0, G, F \rangle$ and an environment $\mathcal{E} = \langle A, E, e_0, \rho \rangle$, the enacted behavior of $\mathcal{B}$ on $\mathcal{E}$ is the tuple $\mathcal{T}_\mathcal{B} = \langle S_\mathcal{B}, A, \delta_\mathcal{B}, s_{\mathcal{B}_0}, F_\mathcal{B} \rangle$, where $S_\mathcal{B} = B \times E$ is the (finite) set of states, A is the same set of actions as defined in $\mathcal{E}$, $\delta_\mathcal{B} \subseteq S_\mathcal{B} \times A \times S_\mathcal{B}$ is the transition relation, $s_{\mathcal{B}_0} = \langle b_0, e_0 \rangle \in S_\mathcal{B}$ is the initial state and $F_\mathcal{B} = F \times E$ is the set of final states. The transition $\langle \langle b, e \rangle, a, \langle b', e' \rangle \rangle \in \delta_\mathcal{B}$ if and only if:*

— *$\langle e, a, e' \rangle \in \rho$;*

— *$\langle b, g, a, b' \rangle \in \delta$ and $g(e) = \top$.*

It means that $\mathcal{B}$ and $\mathcal{E}$ synchronize on all actions.

Given a state $\boldsymbol{b} = \langle b, e \rangle \in S_\mathcal{B}$, b and e are denoted by $beh(\boldsymbol{b})$ and $env(\boldsymbol{b})$, respectively. As depicted in Figure 1.1, the notion of enacted target behavior is defined from the target behavior on the environment.

Definition 1.1.4. *The enacted target behavior $\mathcal{T}_{\mathcal{B}_t}$ is the tuple $\langle S_{\mathcal{B}_t}, A, \delta_{\mathcal{B}_t}, s_{\mathcal{B}_{t0}}, F_{\mathcal{B}_t}\rangle$ such that $\mathcal{T}_{\mathcal{B}_t}$ is the enacted behavior of $\mathcal{B}_t$ on $\mathcal{E}$.*

All available behaviors in a system operate in the shared environment in an interleaved fashion, called the enacted system behavior.

Definition 1.1.5. *Given a system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E}\rangle$, the enacted system behavior of $\mathcal{S}$ is the tuple $\mathcal{T}_\mathcal{S} = \langle S_\mathcal{S}, A, I_n, \delta_\mathcal{S}, s_{\mathcal{S}0}, F_\mathcal{S}\rangle$, where $S_\mathcal{S} = B_1 \times \ldots \times B_n \times E$, $I_n = \{1, \ldots, n\}$ is the set of behavior indexes, $\delta_\mathcal{S} \subseteq S_\mathcal{S} \times A \times I_n \times S_\mathcal{S}$ is the transition relation, $s_{\mathcal{S}0} = \langle b_{10}, \ldots, b_{n0}, e_0\rangle$ is the initial state and $F_\mathcal{S} = \{\boldsymbol{s} \in S_\mathcal{S} \mid beh_i(\boldsymbol{s}) \in F_i \text{ for all } i \in I_n\}$ is the set of final states. The transition:*

$$\langle\langle b_1, \ldots, b_i, \ldots, b_n, e\rangle, \langle a, i\rangle, \langle b_1, \ldots, b_i', \ldots, b_n, e'\rangle\rangle \in \delta_\mathcal{S}$$

if and only if:

— $\langle e, a, e'\rangle \in \rho$;

— $\langle b_i, g_i, a, b_i'\rangle \in \delta_i$ *and* $g_i(e) = \top,\ i \in I_n$.

It means that the environment synchronizes with behavior $\mathcal{B}_i$ on action a independently of the other behaviors.

When there is no environment, the actions that belong to A are given out to available behaviors, that is, $\mathcal{B}_i = \langle B_i, A_i, \delta_i, b_{i0}, F_i\rangle$, where guards are eliminated and elements are defined as in Definition 1.1.2, but with $\delta_i \subseteq B_i \times A_i \times B_i$. In the same way $\mathcal{B}_t = \langle B_t, A_t, \delta_t, b_{t0}, F_t\rangle$, where elements are defined as in Definition 1.1.2, but with $\delta_t \subseteq B_t \times A_t \times B_t$ and $A_t \subseteq \cup_i A_i$. In that case, the notions of enacted system behavior and enacted target behavior are unnecessary. There are only the system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n\rangle$ and the target behavior $\mathcal{B}_t$. The system behavior, also denoted by $\mathcal{S}$ is the tuple $\mathcal{S} = \langle S, A, I_n, \delta, s_0, F\rangle$, where $S = B_1 \times \ldots \times B_n$, $A = \cup_i A_i$, $s_0 = \langle b_{10}, \ldots, b_{n0}\rangle$, $F = F_1 \times \ldots \times F_n$ and $\delta \subseteq S \times A \times I_n \times S$ is the system transition relation. The transition $\langle\langle b_1, \ldots, b_i, \ldots, b_n\rangle, \langle a, i\rangle, \langle b_1, \ldots, b_i', \ldots, b_n\rangle\rangle \in \delta$ if and only if $\langle b_i, a, b_i'\rangle \in \delta_i,\ i \in I_n$.

1.1.1 Behavior Composition Problem

A typical behavior composition problem consists in the synthesis of a controller in order to realize a desired target behavior by coordinating a set of available behaviors. This problem was proposed through a framework, called *automatic behavior composition* [19].

1.1.2 Controller Synthesis

In the behavior composition framework, there are two ways for synthesizing a controller generator. One way is based on an algorithm, which calculates the largest ND-simulation. The other concerns the calculation of a winning strategy of a corresponding two-player safety game by using the model checker TLV/SMV.

1.1.3 Synthesis Based on an ND-simulation Relation

Definition 1.1.6. *Let* $\boldsymbol{t} \in S_{\mathcal{B}_t}$ *and* $\boldsymbol{s} \in S_{\mathcal{S}}$*, an ND-simulation relation of* $\mathcal{T}_{\mathcal{B}_t}$ *by* $\mathcal{T}_{\mathcal{S}}$ *is a relation* $R \subseteq S_{\mathcal{B}_t} \times S_{\mathcal{S}}$*, such that* $\langle \boldsymbol{t}, \boldsymbol{s} \rangle \in R$ *implies:*

1. $env(\boldsymbol{t}\ ) = env(\boldsymbol{s})$;
2. *if* $\boldsymbol{t} \in F_{\mathcal{B}_t}$*, then* $\boldsymbol{s} \in F_{\mathcal{S}}$;
3. *for all actions* $a \in A$*, there is a* $k \in I_n$ *such that for all transitions* $\langle \boldsymbol{t}, a, \boldsymbol{t}' \rangle \in \delta_{\mathcal{B}_t}$*:*
   - — *there is a transition* $\langle \boldsymbol{s}, \langle a, k \rangle, \boldsymbol{s}' \rangle \in \delta_{\mathcal{S}}$ *with* $env(\boldsymbol{t}') = env(\boldsymbol{s}')$;
   - — *for all transitions* $\langle \boldsymbol{s}, \langle a, k \rangle, \boldsymbol{s}' \rangle \in \delta_{\mathcal{S}}$ *with* $env(\boldsymbol{t}') = env(\boldsymbol{s}')$*, it is the case that* $\langle \boldsymbol{t}', \boldsymbol{s}' \rangle \in R$.

The symbol "$\preceq$*" is used to denote that a state* $\boldsymbol{t} \in S_{\mathcal{B}_t}$ *is ND-simulated by a state* $\boldsymbol{s} \in S_{\mathcal{S}}$*,* $(\boldsymbol{t} \preceq \boldsymbol{s})$*, that is, there exists an ND-simulation relation* R *of* $\mathcal{T}_{\mathcal{B}_t}$ *by* $\mathcal{T}_{\mathcal{S}}$ *such that* $\langle \boldsymbol{t}, \boldsymbol{s} \rangle \in R$.

There exists an algorithm that computes the largest ND-simulation relation [19]. The theorem 1 in [19] proves that a controller of $\mathcal{B}_t$ on $\mathcal{S}$ exists if and only if $s_{\mathcal{B}_{t0}} \preceq s_{\mathcal{S}0}$. From the largest ND-simulation relation, a finite state machine, called controller generator, can be derived. It is formally defined as follows.

Definition 1.1.7. *A controller generator is the tuple* $CG = \langle \Sigma, A, I_n, \xi, \omega \rangle$*, where* $\Sigma = \{ \langle \boldsymbol{t}, \boldsymbol{s} \rangle \in S_{\mathcal{B}_t} \times S_{\mathcal{S}} \mid \boldsymbol{t} \preceq \boldsymbol{s} \}$ *is the set of CG states. Given a state* $\sigma = \langle \boldsymbol{t}, \boldsymbol{s} \rangle \in \Sigma$*,* $\boldsymbol{t}$ *and* $\boldsymbol{s}$ *are denoted by* $com_{\mathcal{B}_t}(\sigma)$ *and* $com_{\mathcal{S}}(\sigma)$*, respectively, and* $\xi \subseteq \Sigma \times A \times I_n \times \Sigma$ *is the CG transition relation. The transition* $\langle \sigma, \langle a, k \rangle, \sigma' \rangle \in \xi$ *if and only if:*

- — $\langle com_{\mathcal{B}_t}(\sigma), a, com_{\mathcal{B}_t}(\sigma') \rangle \in \delta_{\mathcal{B}_t}$;

— $\langle com_{\mathcal{S}}(\sigma), \langle a, k\rangle, com_{\mathcal{S}}(\sigma')\rangle \in \delta_{\mathcal{S}}$*;*

— *for all* $\langle com_{\mathcal{S}}(\sigma), \langle a, k\rangle, \boldsymbol{s}'\rangle \in \delta_{\mathcal{S}}$*,* $\langle com_{\mathcal{B}_t}(\sigma'), \boldsymbol{s}'\rangle \in \Sigma$.

The function $\omega : \Sigma \times A \to 2^{I_n}$ *is an output function defined as:*

$$\omega(\sigma, a) = \{k \mid \exists \sigma' \in \Sigma \textit{ such that } \langle \sigma, \langle a, k\rangle, \sigma'\rangle \in \xi\}.$$

Given an action and the current state of system, the output of CG is the set of available behaviors that may execute the action while preserving the largest ND-simulation relation. Notice that, computing CG from the largest ND-simulation relation just involves checking local conditions [17].

A family of generated controllers, called also *compositions* of $\mathcal{B}_t$ on $\mathcal{S}$, can be extracted from the controller generator. Notice that, in some cases, the number of generated controllers can be infinite [19].

1.1.4 Fault Tolerance in the Behavior Composition Framework

As an example, the behavior composition framework is used to illustrate forward recovery implemented in service composition.

The behavior composition framework [19] through the planning phase calculates the largest ND-simulation of the target behavior by the system (the product of all available behaviors).

The largest ND-simulation has enough information to find all possible compositions. Such compositions are not sufficiently robust to deal with failures in behavior functionality, availability or communication over network. For instance, a service for online payments as a complex and stateful behavior that guarantees the correctness of transactions might not expect to leave a transaction in an inconsistent state or wait for a long period of time to handle a request in presence of failures in any services. Five core forms of these failures are as follows as described in [18].

> A behavior temporarily freezes, that is, it stops responding and remains still, then eventually resumes in the same state it was in. As a result, while frozen, the controller cannot delegate actions to it.

A behavior unexpectedly and arbitrarily (i.e., without respecting its transition relation) changes its current state. The controller can in principle keep delegating actions to it, but it must take into account the behavior's new state.

The environment unexpectedly and arbitrarily (i.e., without respecting its transition relation) changes its current state. The controller has to take into account that this affects both the target and the available behaviors.

A behavior dies, that is, it becomes permanently unavailable. The controller has to completely stop delegating actions to it.

A behavior that was assumed dead unexpectedly resumes operation starting in a certain state. The controller can exploit this opportunity by delegating actions to the resumed behavior, again.

In another work [20], it is assumed that the controller generator always deal with fully reliable services and does not address the above cases. As a consequence, upon any of the above failures, the only option is "re-planning" from scratch for a whole new controller. Planning is finding an appropriate combination of behaviors so that requested actions and certain goals can be achieved [8]. To avoid re-planning from scratch upon any of above failures, the aforementioned simulation-based approach includes a solution that reactively (on-the-fly) or parsimoniously adjusts to failures [18] in a more robust fashion. "Just-in-time" controller generator, as a reactive solution, can delay choosing the next operation according to criteria and available information until run-time, so that the ND-simulation relation is preserved. With respect to the failure cases mentioned above, for the first case, when the behavior freezes, the controller generator just avoid to select the frozen behavior and try to find another one to delegate operations. Corresponding to second and third cases, in the case of any unexpected changes in the internal state, the same solution can be applied to deal with failures. For the last two cases, however, a reactive approach is not adequate as the behaviors stop working permanently and not resume eventually. Indeed, the difference between temporary and permanent unavailability of behaviors is in the possibility of operation delegation in controller generator. In the permanent unavailability, the behavior and all dependent compositions, need to be discarded and removed, but the temporary unavailability has a cost in execution delay and the controller generator is able to delegate operations to it as it will be resume eventually. Lastly, when a new behavior becomes available, re-computation of the largest-ND simulation is necessary, as there are more possibilities to generate a controller through a controller generator.

1.2 Fault Tolerance

A general agreement on fault tolerance definition and understanding that how it can help to guarantee availability and reliability of systems is one of the main problems in designing systems[1]. Availability and reliability terms are described in more details later, but in nutshell, the reliability measures how long the system can operate before malfunctioning, even in the presence of faulty components. The availability measures the mean proportion of time that the system is available for use. Normally, fault-tolerant systems are evaluated with respect to these two criteria.

Fault tolerance is the ability of a system to perform its functions correctly even in the presence of faults and continue normal operating without interruption [21]. The purpose of fault tolerance is to increase the dependability of a system. Dependability is the possibility to depend on a system behavior in an appropriate manner, both during normal circumstances and when some forms of fault have occurred, either in the software or hardware. The notion of dependability may also include that the system behaves appropriately even under workloads exceeding the largest workload the system can handle, perhaps with downgraded performance.

Indeed, fault tolerance ensures that the fault does not cause the overall system to malfunction, but there is also a good chance that performance will start to degrade until the busted component is replaced. Dependability is identified by several attributes.

1.2.1 Dependability Attributes

Fault tolerance is needed in many systems to ensure overall system dependability [36] because the consequences of a malfunction are more expensive than the cost

1. A design is a description of how the components interact with each other.

of preventing it. For a system to be fault tolerant, it must to provide three main dependability attributes in distributed systems [16, 4]:

1) Reliability which is the probability of a service to produce correct outputs and provide successful execution of a program up to some given time t. A reliable system does not continue to deliver results that include uncorrected corrupted data. Reliability can be characterized in terms of Mean Time Between Failure (MTBF), but the more exact term is Mean Time To Failure (MTTF):

$$Reliability = MTTF = \frac{t}{MTBF}. \tag{1.1}$$

2) Availability which is probability of a service to be operational as expected without failing. When a failure occurs, the amount of time until service is restored is the Mean Time To Repair (MTTR). It depends on MTTF (reliability). Taking the ratio of the average time that a system is available to the total time it is expected to be operational gives the formula for this attribute:

$$Availability = \frac{MTTF}{MTTF + MTTR}. \tag{1.2}$$

Availability is typically given as a percentage of the time that a system is expected to be available, e.g., 99.999 percent (five nines).

The distinction between reliability and availability is notable: reliability is the system ability to function correctly without data corruption, whereas availability measures how often the system is available for use, even though it may not be functioning correctly. For instance, a server may run forever and so has ideal availability, but may be unreliable with frequent data corruption. For instance, in passenger transportation systems, it is important that the system be continuously available and also not corrupt data when a failure occurs.

3) Safety which prevents any unauthorized access.

Moreover, integrity and maintainability are also dependability attributes.

In recent years, a gradual development from large monolithic systems [2] to systems, consisting of smaller and decoupled independent services that provide a single functionality and communicate with each other using synchronous or asynchronous

2. Single-server systems.

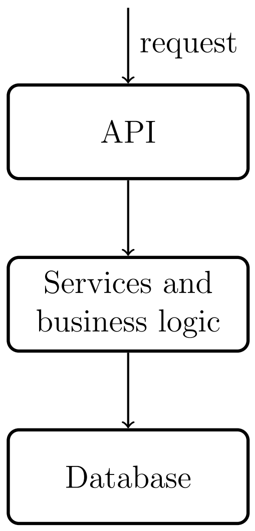

Figure 1.2: Monolithic architecture

techniques, has been observed. Figures 1.2 and 1.3 illustrate the differences in these architectures. So, dealing with various dependability-related concerns, such as faults, errors and failures, are still in the core of researches [4, 43]. Moreover, to ensure the overall system dependability, exception handling mechanisms during the entire life cycle have been advocated as one of the main approaches [36].

1.2.2 Dependability Concerns

A failure occurs when an actual running system deviates from the specified behavior. The cause of a failure is called an error. An error represents an invalid system state, one that is not allowed by the system behavior specification. The error itself is the result of a defect in the system or fault. In other words, a fault is the root cause of a failure. It means that an error is merely the symptom of a fault. A fault may not necessarily result in an error, but the same fault may result in multiple errors. Similarly, a single error may lead to multiple failures [4].

According to these concerns and the main goal for achieving fault-tolerant systems, it is required to avoid potential faults. Four approaches to reach this goal are [4]: fault prevention, fault removal, fault forecasting and fault tolerance.

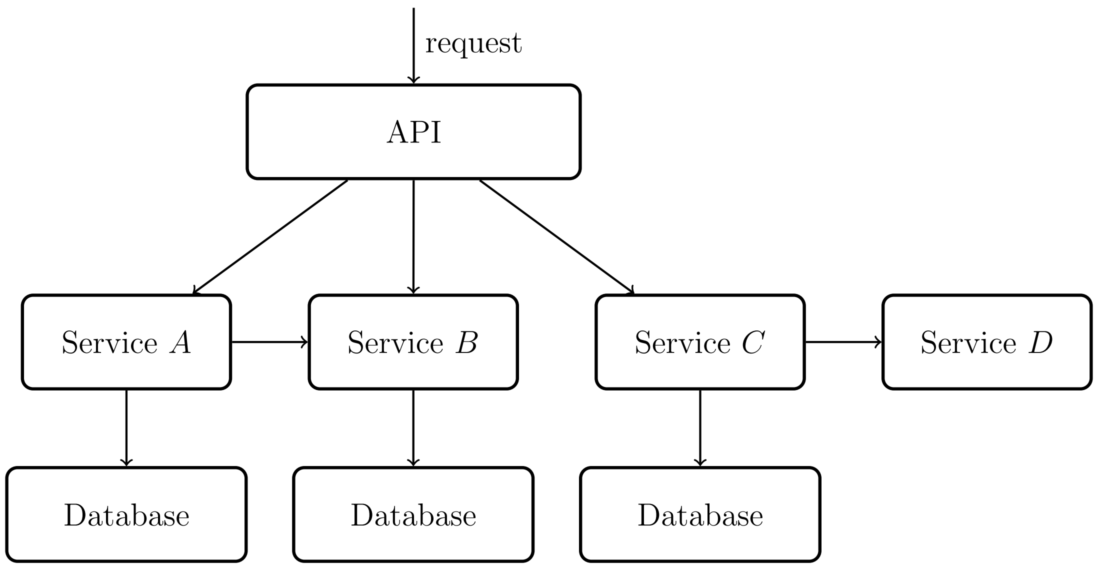

Figure 1.3: Independent service architecture

- *Fault prevention* means to prevent faults from being present in the system. The aim of fault prevention is ensuring that all possible faults are removed from the system before deployment. It is used in modelling, design, verification and validation methodologies and code inspections to avoid fault occurrences.
- *Fault removal* measures the number of faults in the system in order to remove and reduce them. The range of techniques used for fault removal includes unit testing, integration testing, regression testing and back-to-back testing.
- *Fault forecasting* copes with the future system faults that may cause a failure.
- *Fault tolerance* prevents system to be failed overally in the presence of faults. A system built with fault tolerance capabilities will manage to keep operating when a failure happens, but at a degraded level. For a system to be fault tolerant, it must be able to detect, diagnose, confine, mask, compensate and recover from faults.

From these definitions, the aim of fault prevention and fault tolerance is to provide reliability while fault removal and fault forecasting focus on providing availability. They are, however, complementary and must be taken in all phases of system development to increase dependability. For instance, fault prevention techniques are used in design phase, fault removal and forecasting in implementation phase and fault tolerance in execution phase.

1.2.3 Fault and Failure Classifications

Faults can be classified as transient, intermittent and permanent [22, 43]. A transient fault will eventually disappear by software restarting or message retransmission, whereas a permanent one, such as power breakdowns, disrupts a system functionality as desired and will remain unless it is removed. While it may seem that permanent faults are more severe, from an engineering perspective, they are much easier to diagnose and handle. A particularly problematic type of fault is an intermittent fault that recurs, often unpredictably.

Failures, can also be classified into the following categories during computation on system resources:

- Response failure—the component fails to response.
- Crash failure—the component either completely stops operating or never returns to a valid state.
- Omission failure—the component completely fails to perform its service.
- Timing failure—the component does not complete its service on time.
- Byzantine failure—it is defined as arbitrary deviations of a process from its assumed behavior it is supposed to be running.

1.2.4 Fault Tolerance Techniques

The characteristic of fault tolerance is not complete. It is obvious that there is not any system to tolerate every possible faults and there are always some combinations of events and failures that lead to the disruption of the system. However, based on fault tolerance policies, different techniques are introduced. Figure 1.4 shows these techniques and a brief definition of each is given below [5, 22].

Reactive fault tolerance is aimed to reduce the effect of a failure after it is occurred by bring it back to a latest state or if possible before occurrence of a failure. Based on this policy various techniques exists:

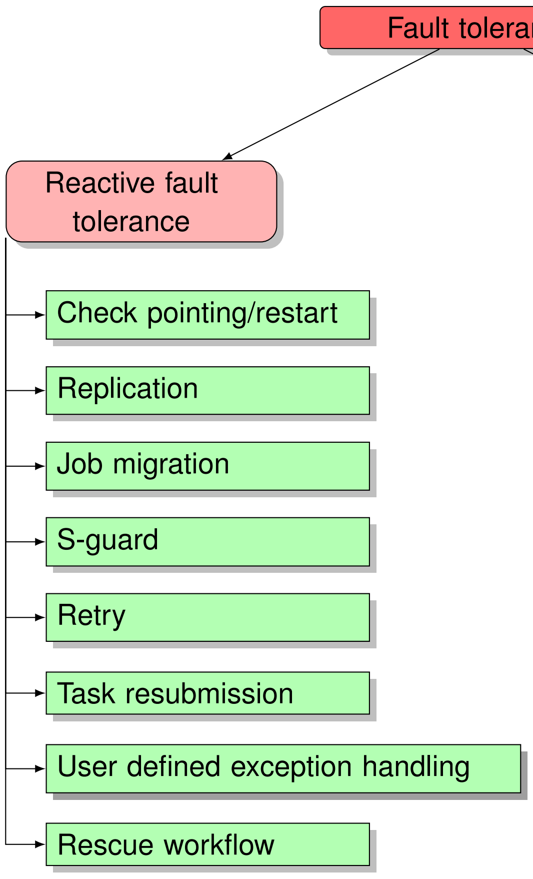

Figure 1.4: Fault tolerance techniques

- Check Pointing/restart—In the event of a failure, system is restored to a previously stored check-point rather than starting it from the beginning.
- Replication—The main idea is to create multiple copies of data or services and storing them at multiple servers and coordinating client interactions with server replicas. If one of them failed, the other ones are accessible so that performance in not affected. Data consistency is one of the replication limitations. Based on deterministic or non-deterministic processes, it is divided to active and passive replication.
  — *Active replication*, as illustrated in Figure 1.5, can be used only for deterministic processes. The request is processed by all replicated servers and,

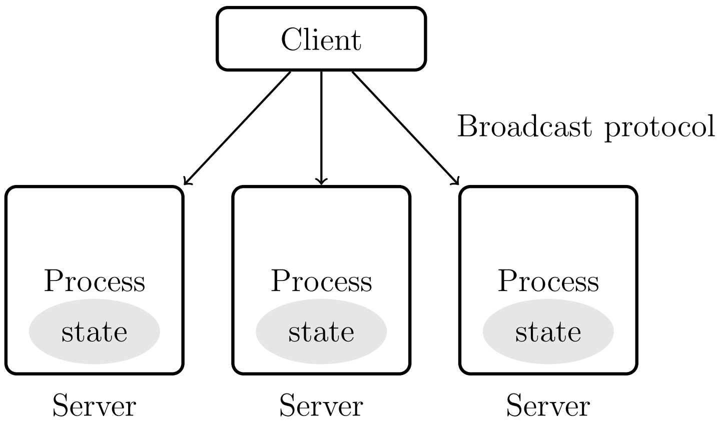

Figure 1.5: Active replication

in order to make all the servers receive the same sequence of operations, an atomic broadcast protocol must be used. An atomic broadcast protocol guarantees that either all the servers receive a message or none, plus that they all receive messages in the same order.

— *Passive replication*, as shown in Figure 1.6, can be used for nondeterministic processes. There is only one server that processes client's request known as primary server and the other servers act as back up servers. After processing a request, the primary server updates the state on the other (backup) servers and sends back the response to the client. If the primary server fails, one of the backup servers takes its place. Response time is high as there is only one server which process many client's request.

- Job migration—In the case of failure on a particular machine while executing an operation, it can be migrated to another machine.
- S-guard—It is based on rollback recovery which is one of backward recovery approaches introduced in Chapter 1.
- Retry—It is the simplest technique as the operation is resubmitted to the same machine again and again.
- Task resubmission—Resubmitting a failed operation on the same machine or another one.

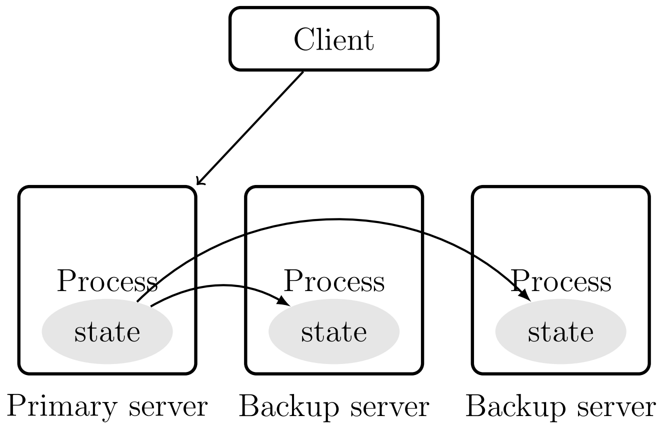

Figure 1.6: Passive replication

- User defined exception handling—The user predefines a specific treatment in the case of failure.
- Rescue workflow—It allows to continue the steps in the workflow until it becomes impossible to move forward.

Proactive fault tolerance proactively detects the faults in a component and introduces a replacement in order to avoid recovery from faults, errors and failures:

- Software rejuvenation—It is a helper approach to prevent performance degradation and other associated failures related to software aging. A typical method is the hardware or software reboot to restart the system from scratch.
- Self-healing—Automatically handle a failure in each individual instance.
- Preemptive migration—It is based on a feedback-loop control mechanism, which continually monitors and analyzes a system.

Fault tolerance can be specified quantitatively or qualitatively [37]. A quantitative approach is usually introduces as the maximum allowed failure-rate. For instance, 9–10 failures per hour. A qualitative approach includes several characteristics as follows:

- Fail-safe—When the number of system faults increases and reaches a specified threshold, it fails in a safe mode. For instance, railway signalling systems are designed to fail-safe, so that all trains stop.

- Fail-op—When a system suffers a specified number of faults, it still provides a subset of its specified behaviors.
- No single point of failure—The no single point of failure design simply asserts that no single part of a system can stop the entire from working. Instead, the failed component can be replaced or repaired before another failure occurs.
- Consistency—All information delivered by the system is equivalent to the information that would be delivered by an instance of a non-faulty system.

In addition, various replication and software diversity techniques including recovery blocks, conversations and N-version programming have been developed and widely used in industry. Although redundancy is also identified as one of the principles for designing fault-tolerant systems, there is a misconception about the difference between redundancy and fault tolerance. Redundancy means having several instances of one service, so, if a part of a system fails, there is an "extra or spare" that can operate in place of the failed component such that the system operation is uninterrupted. For instance, having two disks on the same system that are regularly backed up makes them redundant, since if one fails the other can pick up. If the entire system fails, however, both disks are useless. This is the role of fault tolerance to keep the system as a whole operational even if portions of the system fail. Fault tolerance is a requirement, not a feature.

Chapter 2

Review of Fault Tolerance Terminologies and Technologies

Clearly, when it comes to build a robust and reliable composition, such as a service or behavior composition, the concept of failure is considered as a central importance, as failures are inevitable. In the following, we report some approaches to handle failures, present in the literature, that are relevant to our work.

There are various techniques available to provide fault tolerance in the context of service composition. In the industrial world, there are a variety of tools and libraries, which are not usually based on formal theories. Apache Ignite[1], Failsafe[2], Hystrix[3], JRugged[4] and Resilience4j[5] are the most popular fault tolerance libraries in the context of informal (ad-hoc) technologies [26]. Beside these, in literature, there exists some recent research work related to fault tolerance in service and behavior composition, each corresponding to a different perspective and all equally reasonable with some weaknesses to provide reliability [31, 20, 18, 45, 33, 41, 42, 27, 13].

In the next section, different stability patterns in the context of service composition are described. In the following descriptions, the term "system" refers to a software system, which consists of an orchestrator and several services to realize a target behavior. The orchestrator selects services based on a specific policy and delegates each requested operation to a single service at a time.

2.1 Stability Patterns in Fault Tolerance

The stability patterns are designed to protect systems against common failures in services communication and collaboration. Several stability patterns have been

1. https://ignite.apache.org/
2. https://github.com/jhalterman/failsafe
3. https://github.com/Netflix/Hystrix
4. https://github.com/Comcast/jrugged
5. https://github.com/resilience4j/resilience4j

created to minimize the impact of failures. In this section, the most important ones in regard to service composition are described. By such definitions, it will be much easier to explain related fault tolerance libraries and their functionalities. It should be noted that these patterns can be used together or separately according to the problem domain.

2.1.1 Circuit Breaker

It is common in a system to delegate operations to other services running in different processes, probably on different machines across a network to fulfill the incoming requests, but there can be situations where a service can fail due to unanticipated events. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these cases it might be pointless to continually retry an operation that is unlikely to succeed, and instead, the orchestrator should quickly accept that the operation has failed and handle this failure accordingly.

Additionally, if a service is very busy, failure might lead to cascading failures. These blocked delegations might hold critical system resources such as memory, threads, database connections and so on. Consequently, these resources could become exhausted, causing a failure of other possibly unrelated parts of the system that need to use the same resources. In these situations, it would be preferable for the operation to fail immediately, and only attempt to invoke the service if it is likely to succeed. The circuit breaker pattern can prevent an orchestrator from repeatedly trying to delegate an operation to a service that is likely to fail and allows it to continue without waiting for the fault to be fixed. The circuit breaker pattern also is able to detect whether the fault has been resolved. Figure 2.1 shows the behavior of a circuit breaker described by a finite state machine with the states that mimic the functionality of an electrical circuit breaker, namely Closed, Open and Half-open. The basic idea behind the circuit breaker is very simple. Normally a circuit is closed and delegations are executed as usual. When a failure occurs in a service component and exceeds a certain threshold, the circuit breaker trips and all further calls will fail immediately without reaching the service. After some time a few requests are let through to the faulty service to test if it is up again. If they succeed, the circuit is closed and all requests are executed as before. Otherwise, the circuit remains open

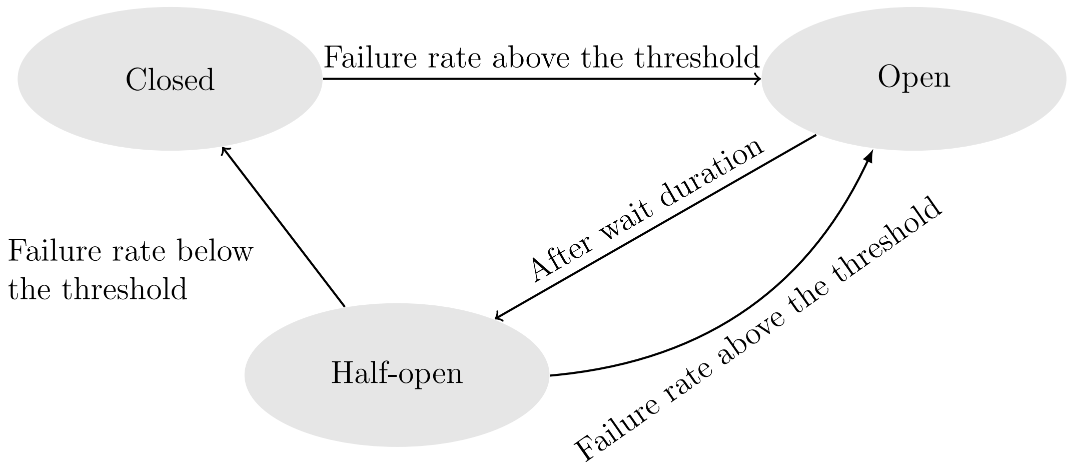
Closed
Failure rate above the threshold
Open
After wait duration
Failure rate below the threshold
Failure rate above the threshold
Half-open

Figure 2.1: Circuit breaker behavior

and the same check is done again after some time. A circuit breaker acts as a proxy for operations that might fail. The proxy should monitor the number of recent failures that have occurred and use this information to decide whether to allow the operation to proceed, or simply return an exception immediately.

2.1.2 Bulkheads

In general, the goal of the bulkhead pattern is to avoid faults in one part of a system to take the entire system down. The term comes from ships, where a ship is divided into separate watertight compartments to avoid a single hull breach to flood the entire ship; it will only flood one bulkhead.

Similar technique can be used in software systems. By partitioning a system, it is possible to confine errors to one area as opposed to taking the entire system down. These partitions can be hardware redundant, binding certain processes to certain CPUs, segmenting different services to different servers or partitioning threads into different thread groups for different functionalities.

Having system split into several independent components ensures that the critical ones will keep running when a failure occurs in one of the less important components. One implementation of bulkheads is using separate thread pools for each service provider.

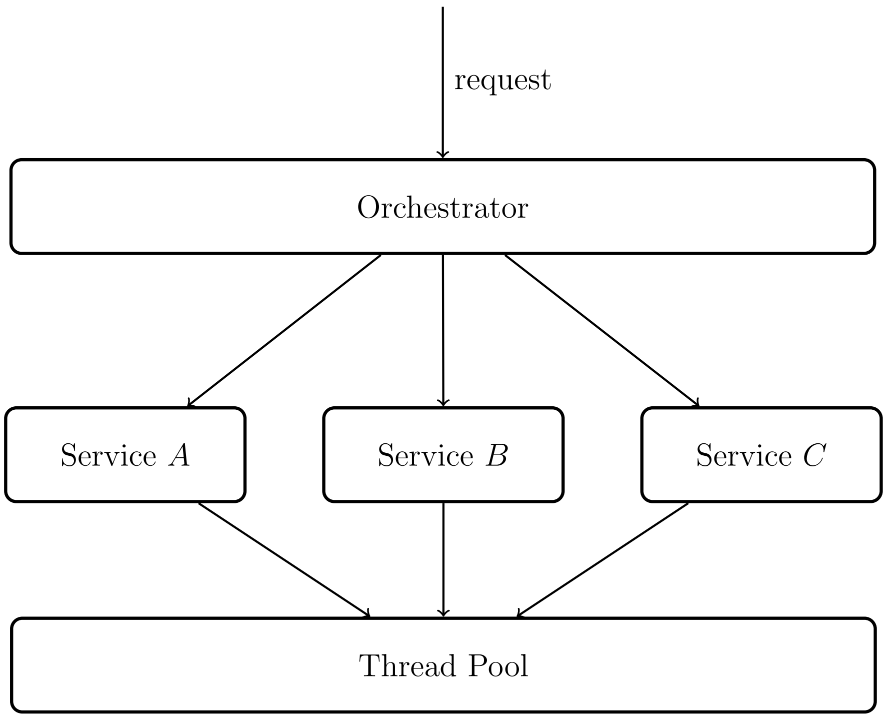

Figure 2.2: Single thread pool

The diagrams in Figures 2.2 and 2.3 show a single thread pool shared between all services and bulkhead structured thread pools assigned to individual services, respectively. In order to show the difference between them, let a multithreaded-based request, which uses three different services: A, B and C. It is assumed that there are thirty request handling threads in the thread pool. Figure 2.2 shows such a system with a shared thread pool. If requests to component C start to hang, as long as all services use the same thread pool, eventually all request handling threads will hang on waiting for an answer from C. This would make the system entirely non-responsive. If the load is high enough, all requests to C are handled slowly and we have a similar problem. Bulkhead pattern limits the number of concurrent calls to a component and would have kept the system safe in this case. Figure 2.3 shows the same system implemented by using the bulkhead pattern. By dividing a thread pool into three pools and assigning each to an individual service, in the case of any failure in C, as

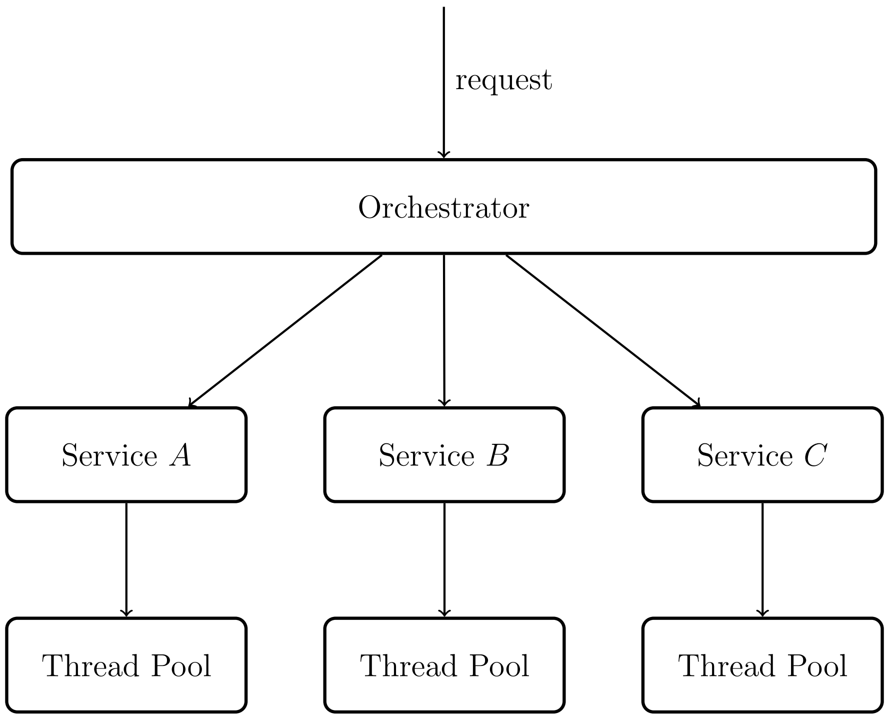

Figure 2.3: Bulkhead structured thread pool

the thread pool is isolated, at most ten request handling threads overwhelm and the other twenty threads can still handle requests in services A and B.

2.1.3 Fail Fast

The fail fast pattern refers to a lightweight form of fault tolerance, whereby a system service terminates itself immediately upon encountering an error. This is done upon encountering a serious error such that it is possible to change the service state to corrupt or inconsistent, and immediate exit is the best way to ensure that no (more) damage is done.

Services should be able to detect a potential failure before requests are sent to them for execution and fail fast. Fail-fast service component is designed to report at the first point of failure, rather than to receive a request and report the failure eventually. This allows easier diagnosis of the underlying problem, and may prevent improper behavior like long running operations in a broken service. This improves stability of the system by avoiding slow responses and helps to keep resources like CPU or memory while the system is under heavy load. Before starting the execution, the system must check the services availability and the service itself needs to try to get all necessary resources and verify their state. It should check all conditions and if a condition is not met, it can fail fast and save valuable time.

2.2 Libraries and a Toolkit for Fault Tolerance Applications

The stability patterns described in the previous section are implemented by various libraries. In the following paragraphs, the most popular fault tolerance libraries are introduced.

2.2.1 Apache Ignite

Apache Ignite is a high-performance, integrated (as shown in Figure 2.4) and distributed in-memory platform for computing and transacting on large-scale data sets in real-time [38].

The Apache Ignite service grid provides users with complete control over services being deployed on the cluster. It allows users to control how many instances of their services should be deployed on each cluster node, ensuring proper deployment and fault tolerance. The service grid guarantees continuous availability of all deployed services in case of node failures. Apache Ignite supports automatic operation failover. In case of a node crash, service operations are automatically transferred to other available nodes for re-execution. There are many conditions that may result in a failure within the node or service and a failover can be triggered. Moreover, there is an ability to choose to which node an operation should be failed over to, as it could be different for different nodes or different computations within the same node. Apache Ignite comes with a number of built-in customizable failover implementations as follows:

- *At Least Once Guarantee*, as long as there is at least one node standing, no operation will ever be lost. As illustrated in Figure 2.5, whenever a primary

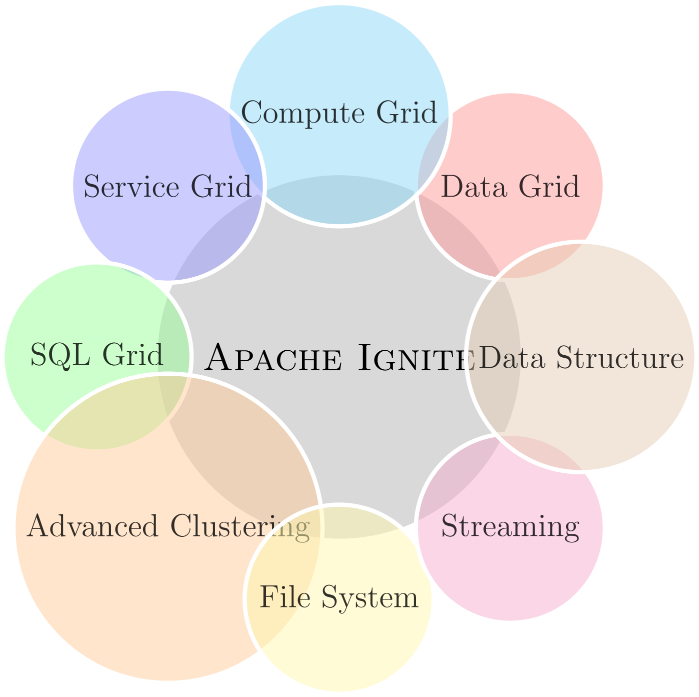

Figure 2.4: Apache Ignite integrated components

node stops or crashes, there is a node as a secondary which guarantees that the operation is not re-mapped to the same node it had failed on.

- *Closure Failover*, which creates an instance of the node with a no-failover flag set on it and triggered if a remote node either crashes or rejects execution. Figure 2.6 shows such a system failover behavior.
- *AlwaysFailOverSpi*, which always reroutes a failed operation to another node. The first attempt will be made to reroute the failed operation to a node not yet involved in any operation of the transaction. If no such nodes are available, then an attempt will be made to reroute the failed operation to one of the nodes that were involved before in the transaction. If none of the above attempts succeeds, then null will be returned. None of the stability patterns is used in this library.

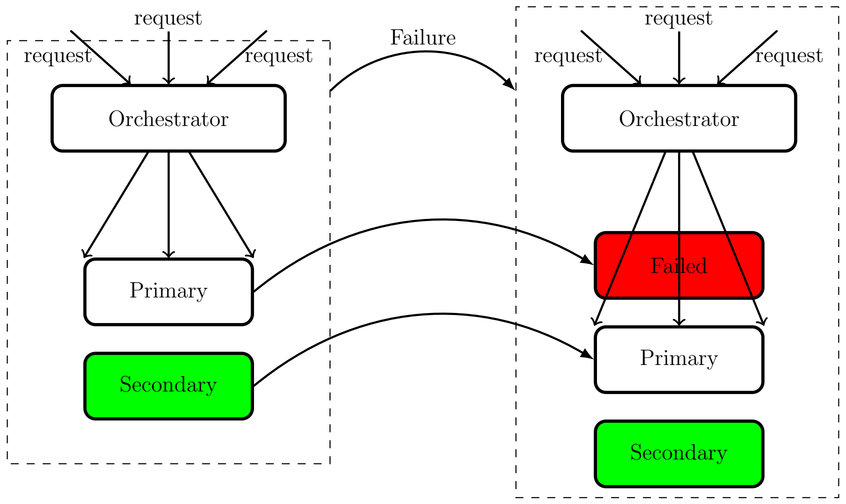

Figure 2.5: At least once guarantee failover in Apache Ignite

2.2.2 Failsafe

Failsafe is a lightweight, zero-dependency library for handling failures. It provides various fault tolerance mechanisms such as circuit breakers, fallbacks and retries. Failsafe is very similar to Resilience4j (see Section 2.2.5) but lacks some of its features such as rate limiter or caching. In comparison with Hystrix (see Section 2.2.3), Failsafe supports retries, user-supplied thread pools and configurable success thresholds. The biggest advantage of Failsafe in comparison with other fault tolerance libraries is that, it does not have any external dependencies while providing a decent set of fault tolerance mechanisms. For instance, Resilience4j uses Vavr[6] as an external library and Hystrix has many more external library dependencies such as Guava[7] and Apache Commons[8].

6. http://www.vavr.io/
7. https://github.com/google/guava
8. https://commons.apache.org/

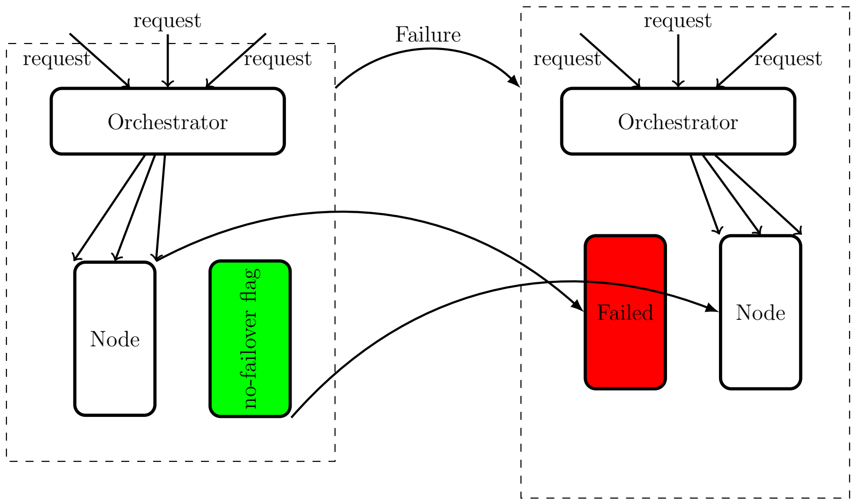

Figure 2.6: Closure failover in APACHE IGNITE

2.2.3 Hystrix

HYSTRIX, is a latency and fault tolerance library from NETFLIX[9]. It isolates integration points, stops cascading failures between services, facilitates usage of fallbacks and provides useful runtime metrics. It is designed to isolate points of access to remote systems, services and third party libraries, stop cascading failures and enable resilience in complex distributed systems where failures are inevitable. Figure 2.7 shows a part of a complex distributed architecture with many dependencies, which is not isolated from dependency failures. The system itself is at risk of being taken down.

On a high volume website, a single back-end dependency becoming latent can cause all system resources to become saturated in a matter of seconds. HYSTRIX helps by providing protection and control over latency and failure from dependencies, most commonly those accessed over network. It helps stop cascading failures and allows to fail fast and rapidly recover, or fallback and gracefully degrade. One of the cornerstones of HYSTRIX is the implementation of the circuit breaker pattern. The HystrixCommand class can be configured with three properties, which affect the

9. https://www.netflix.com/

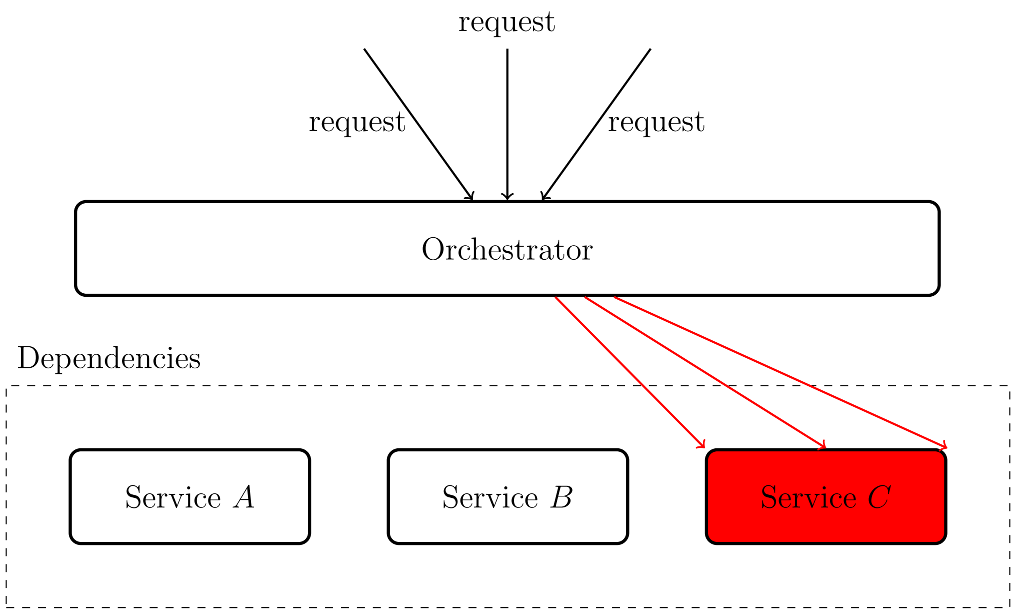

Figure 2.7: Service composition without Hystrix

behavior of circuit breakers: request volume threshold, error threshold percentage and sleep window. When a command is run for the first time, the circuit is closed and the encapsulated business logic is executed. However, if the request volume reaches the given threshold and at the same time error percentage exceeds the threshold, the circuit is switched to open state. In this state, no business logic is executed and the command either immediately throws an exception or executes a fallback if it is specified. Such behavior lasts during the sleep window and once it is over, the circuit breaker is switched to half-open state and a single request is let through. If the request fails, the circuit breaker returns to the open state for the duration of another sleep window. Otherwise, it is switched to closed state and continues to execute the business logic. Hystrix also implements the bulkheads pattern by providing a way to configure different thread pools for different dependencies, so, latency and other problems will only saturate the threads on the same pool and do not affect other dependencies. All Hystrix commands are by default executed in a new thread in order to isolate the calling side from its dependencies which may misbehave. Running commands in separate threads also allows parallel execution and higher throughput.

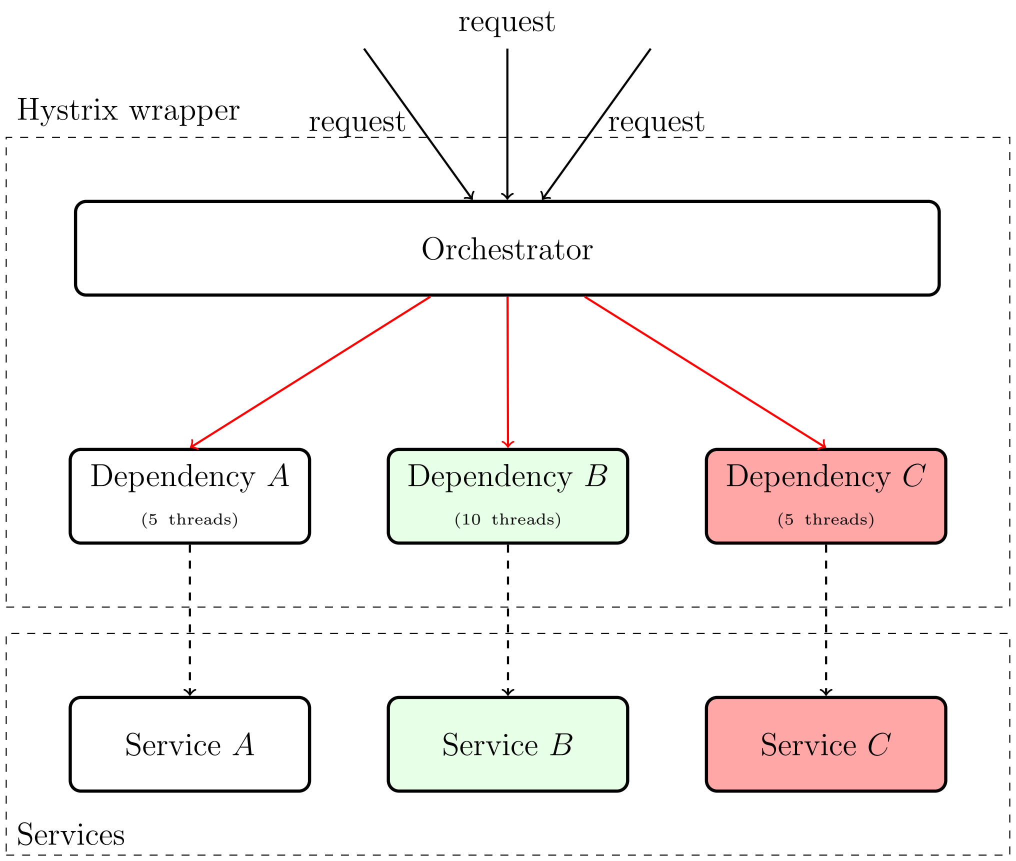

Figure 2.8: Service composition with Hystrix wrapper

When Hystrix is used to wrap each underlying dependency, the architecture as shown in Figure 2.7 changes to resemble the one in Figure 2.8.

Although using separate threads for command execution has several advantages, there is also one significant drawback, computational overhead. Queuing, scheduling and context switching, when running a command in its own thread, have some impact on the performance of the system. This is usually acceptable in exchange for the benefits it brings. The overhead might, however, be too high for low-latency requests and that is why HYSTRIX provides another similar mechanism, semaphores. In this case, all requests are executed directly in calling threads and semaphores are used to limit the number of concurrent calls to any given dependency. But semaphores are not able to deal with timeouts and when a dependency becomes latent, the parent thread remains blocked until the connection timeouts. Hystrix also provides two mechanisms to effectively deal with a large number of requests – collapsing and caching. If request collapsing is used, the requests to a single dependency are not executed immediately but are collected during a short period of time and sent at once in a single request. This leads to better utilization of both network connections and thread pools since only a single thread is used for a collapsed request. On the other hand, request caching allows to execute only the first request from a number of the same ones and return the value from the cache for all the subsequent requests. The biggest advantage of using Hystrix is that it provides a large set of configuration options. It is also quite generic and very well-designed so it can be easily integrated into other libraries or frameworks. However, this is also a little disadvantage since it is more complicated to use Hystrix directly in a project with respect to other libraries.

2.2.4 JRugged

JRugged is a library that provides simple circuit breaker implementation together with some monitoring capabilities. It makes use of the decorator design pattern to wrap potentially dangerous method calls. The decorator pattern is a design pattern that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects. JRugged provides three mechanisms to make services more robust and easier to manage: initializers, circuit breakers and performance monitors. Initializers provide a way to decouple service construction from its initialization. This mechanism is useful for services which do not have all needed resources available for initialization but will eventually get them at some point in the future. Circuit breakers can be used to throttle traffic to a failing service as described in Section 2.1.1. Performance monitors can be used to monitor runtime behavior of a service and collect useful statistics such as latency or throughput. Although, JRugged provides a simple circuit breaker implementation, which is very easy to use, it lacks the fallback feature that is expected from a fault tolerance library. Fallback is a feature to transfer an operation from a failed service to another service.

2.2.5 Resilience4j

The library RESILIENCE4J is a lightweight fault tolerance library inspired by NETFLIX HYSTRIX, but designed for Java 8 and functional programming. Various resilience mechanisms are provided by this library: circuit breaker, fallback, retry, rate limiter and caching. The library has a lot of parameters to configure, which allows to have a great control over integration in services. RESILIENCE4J works very similarly to the one implemented in HYSTRIX but in a more abstract way, so there is no need to create commands for different service methods and it does not have any other external library dependencies. A retry mechanism allows to repeat a call until it eventually succeeds or the maximum limit is reached. A rate limiter can be used to restrict the calling rate of some method in order to be below a certain threshold.

2.2.6 AKKA

Despite the algorithms and techniques which are used to build a fault-tolerant system in the context of service composition, the selection of standard frameworks, tools, programming languages and libraries in development phase to implement such algorithms is a necessity. Different techniques and software libraries are discussed above, but based on the requirements in the context of behavior composition like transition systems and the logic between them, the actor model is one of the successful choices to apply for implementation [25, 1].

Actor Model

The actor model is like a Turing machine which have a formal symbol alphabet, states and transition-rules based description of how a computation in a theoretical environment is done.

Actors in the actor model are defined as independent units of computation with isolated state. These units have two core characteristics: they can send messages asynchronously to one another and they have a mailbox which contains messages that they have received. A mailbox allows messages to be received at any time and then queued for processing. These messages are one-way and, there are no guarantees that a message will ever be received in response. The actor model is so general because it places few restrictions on systems. Asynchrony and the absence of message delivery guarantee enable modeling real distributed systems using the actor model. For example, if message delivery was guaranteed, then the model would be much less general, and only able to model systems which include complex message-delivery protocols.

Akka [40] and Orleans [10] are primary frameworks, which obey the actor model. The Ericsson company originally developed the first programming language, called Erlang, which explicitly implements the actor model [3]. Erlang is used to program large highly-reliable fault-tolerant telecommunications switching systems in Ericsson.

Akka is one of the most popular actor model frameworks that provides a complete toolkit for designing and building highly concurrent, distributed and fault-tolerant applications [40]. It is written in Scala, with language bindings provided for both Scala and Java. Akka is one of the successful choices for composition, which is based on the actor model and message passing. Actors can only be modified by the exchange of messages to avoid locking and blocking [1].

In an actor-based system, everything is an actor, in much the same way that everything is an object in object-oriented design. A key difference is that the actor model was specifically designed and architected to serve as a concurrent model, whereas the object-oriented model is not. The mechanism by which actors share information with each other, is message passing. Although message passing and copying data would be costly, fault tolerance is a more important concern than performance in this work.

Fault tolerance and recovery might be handled by some external libraries (e.g., Failsafe, JRugged, Resilience4j and so on), but Akka supports failure handling and recovery via built-in supervision strategies. Akka is based on the actor model and the comparison between them could not be adequate. However, it is worth to mention different features in Akka.

Akka Features

A finite state machine in Akka is implemented as an FSM [10] actor. It can be described as a relation of the form:

10. Finite state machine.

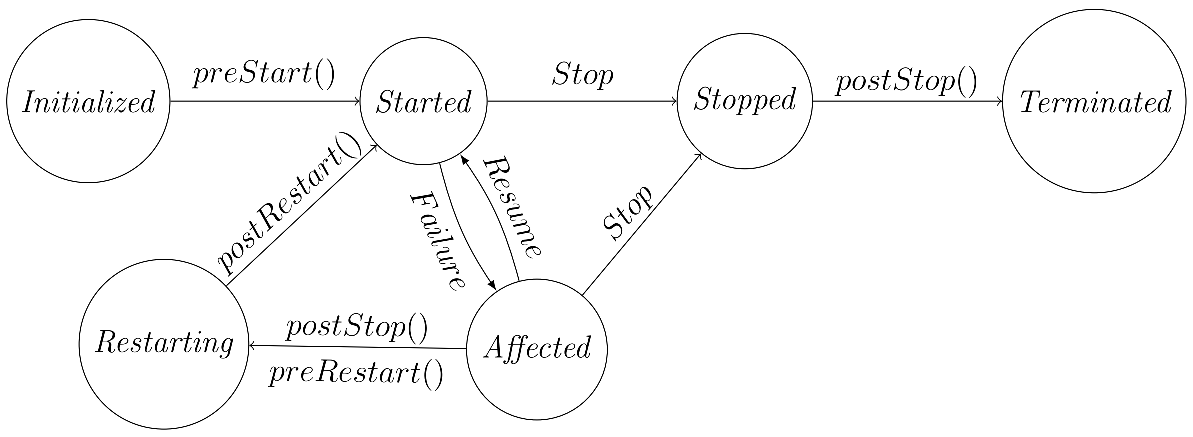

Figure 2.9: Actor lifecycle

State(S) $\times$ Event(E) $\rightarrow$ Actions(A), State(S').

It is interpreted as follows. If an actor is in state S and the event E occurs, an actor perform the actions A and make a transition to the state S'. In a plain AKKA actor, any object can be sent to the actor as a message. An FSM actor is not different, but the messages are wrapped in an instance of Event, which includes its current state.

Figure 2.9 depicts an actor lifecycle. An actor is essentially nothing more than an object that receives messages and takes actions to handle them. It is decoupled from the source of the message and its only responsibility is to properly recognize the type of message it has received and take action accordingly. Upon receiving a message, an actor may take one or more of the following actions:

- execute some operations itself (such as performing calculations, persisting data, calling an external service, and so on);
- forward the message or a derived message to another actor;
- instantiate a new actor and forward the message to it.

Alternatively, the actor may choose to ignore the message entirely (i.e., it may choose inaction) if it deems it appropriate to do so.

AKKA creates a layer between the actors and the underlying system such that actors simply need to process messages. All the complexity of creating and scheduling threads, receiving and dispatching messages and handling race conditions and synchronization, is relegated to the framework to handle transparently. Akka includes several modules to deal with the following common issues:

- *fault tolerance* via supervision hierarchies;
- *persistence* to store actor information or even take snapshots, recover after crash or restart;
- *cluster management* to group and distribute actors across physical machines.

Supervisor hierarchies are the primary and straightforward mechanism for defining the fault-tolerant behavior of the system. In an actor system, each actor is the supervisor of its children. A supervisor reacts and handles exceptions in a way that refers to a supervisor strategy. If an actor fails to handle a message, it suspends itself along with all of its children and sends a message, usually in the form of an exception, to its supervisor. A supervisor decides to apply the action just to the failed actor or to its siblings and children as well. There are two strategies in this case: *OneForOneStrategy* which applies the specified action to the failed child only and *AllForOneStrategy* which applies the specified action to all of its children.

Consequently, when a message signifying a failure reaches a supervisor, it can take one of the following actions[11]:

- Resume the child (and its children), keeping its internal state. This strategy can be applied when the child state was not corrupted by the error and it can continue functioning correctly.
- Restart the child (and its children), clearing its internal state. This strategy can be used in the opposite scenario of the one just described. If the child state has been corrupted by the error, it is necessary the reset its state before it can be used in the future.
- Stop the child (and its children) permanently. This strategy can be employed in cases where the error condition is not believed to be rectifiable, but does not jeopardize the rest of the operation being performed, which can be completed in the absence of the failed child.

11. https://doc.akka.io/docs/akka/2.5.4/scala/general/supervision.html

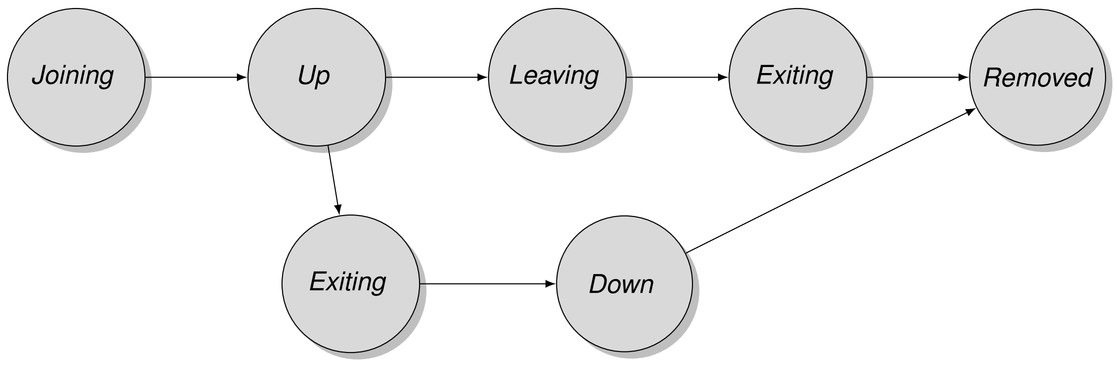

Figure 2.10: Membership lifecycle

- Stop itself and escalate the error. Employed when the supervisor does not know how to handle the failure and so it escalates it to its own supervisor.

Usage

In addition, AKKA cluster provides a fault-tolerant decentralized peer-to-peer based cluster membership service with no single point of failure or no single point of bottleneck. It is implemented by using gossip protocols and an automatic failure detector. A cluster is made up from collaborating actor systems called member nodes.

It does not matter whether the member nodes reside on the same host or on different ones, as in a typical production setting one would most probably spread the member nodes across multiple hosts to get scalability and resilience.

In a nutshell, nodes can join an existing cluster and existing member nodes can leave deliberately or by failure. Figure 2.10 shows all the possible node states and membership lifecycle.

The node begins with the joining state. Once all nodes have seen that the new node is joining through a gossip protocol, the leader will set the node in up.

If a node is leaving the cluster in a safe and expected manner then it switches to the leaving state. Once the leader sees the node in the leaving state, the leader will then move it to exiting state. Once all nodes have seen the exiting state the leader will remove the node from the cluster and mark it as removed.

If something abnormal happen on the node, it is set to unreachable. If a node is unreachable then gossip convergence is not possible and therefore any leader actions are also not possible. So, the state of the unreachable node must be changed. It must become reachable again or marked as down. The cluster can, through the leader, also auto-down a node after a configured time of unreachability.

When reviewing different frameworks it is also worth to know where AKKA is used in the industry and distributed systems. It gives insight into which features of actor systems are actually useful, and the trends that exist throughout these systems. AKKA is using in different companies, including Gilt, Huffington Post, Hootsuite, LinkedIn, Ticketfly, Walmart and WhitePages.

2.3 Approaches for Fault Tolerance Applications

Fault tolerance can be managed by forward and backward recovery techniques. Most of the recovery techniques usually refer to web services. In our proposal, the notion of service in general is, however, considered. In fact, rather than their underlying technologies, frameworks and programming languages, which are used for service implementation and communication, their main concepts for services and composition are considered.

2.3.1 Forward Recovery

In this technique, based on the availability of the other services, by suitably finding a good substitution the operation can be delegated to another candidate. QoS[12]-AWARE fault tolerant is a QoS-AWARE fault-tolerant middleware [45]. The middleware obtains nonfunctional QoS information of all service providers and find an optimal fault tolerance strategy for both stateless and stateful services through an algorithm, called FT-BABHEU. In this technique, the following fault tolerance strategies for service composition are identified: retry, recovery block, n-version programming and active. Regularly, the middleware records all information from the available services and exchange them with the responsible modules to replace the older information with new nonfunctional QoS performance information and apply them to realize the next coming requests.

12. Quality of service.

2.3.2 Backward Recovery

In order to implements backward recovery, different concepts and formalisms have been proposed in the literature which are discussed here.

In distributed and decentralized execution model, continuation-passing messaging also deals with fault tolerance [42]. The proposed architecture is composed by different nodes communicating each other by message exchange. So, execution of services depends on the result of the message interpretation without any communication with an orchestrator or central node. In contrast with a centralized approach, where all of the information must be stored in a central node, in this approach all information about the execution order and service states are carried in messages to react to their failures. In order to handle faults and backward recovery in a bulkhead fashion, nodes are defined in separate scopes. Each scope is responsible to catch the failures from the surrounded nodes and reverse the effect of the completed operations in that particular scope. Interestingly, the compensation execution is also encapsulated in the messages, which are exchanged between nodes.

As an example of a centralized solution, FACTS uses a combination of forward and backward recovery. More precisely, it combines exception handling strategies and a service transfer based termination protocol, called EXTRA, to improve the reliability of composite services [27]. Eight high-level exception handling strategies engage to repair the faulty service in this framework. EXTRA applies a new concept named vitality degree and a new taxonomy of transactional web services in order to cope with failures. Finally, if the fault has been fixed, the execution continues, otherwise it brings the TWS [13] back to a consistent termination state with minimum compensation cost according to the termination protocol.

A number of compensation mechanisms of web services based on Petri net formalism have been proposed in the recent literature [33, 41].

Compensation is elimination of the effects of any operation in the case of failure or cancellation by rollback all executed operations in the former services which participated to realize the request. Based on this definition, various recovery techniques on services such as compensation, re-execution of service operations and substitution

13. Transactional web service.

of services have been introduced.

In these works, Petri nets are deterministic and need to be constructed manually. The compensation process is, however, dynamic. The compensation process, represented by paired Petri nets, requires that all services to be compensatable. Reachability, deadlock-free and liveness of compensation are highlighted in this technique. Upon any failure in services, compensation handler is triggered to execute the backward recovery. Four different compensation patterns are presented as follows: sequence composition compensation pattern, parallel composition compensation pattern, selection composition compensation pattern and iteration composition compensation pattern.

Another approach, introduces more properties on services rather than only compensatability [13]. Considering these properties, the faulty service can be replaced with another service before backward recovery is executed. Indeed, this approach combines the forward and backward recovery and maximizes the QoS. If finding an equivalent service is not possible, backward recovery based on an unfolding process over a colored Petri net will be executed in order to leave the system in a consistent state. Unlike the previous model, the Petri nets are generated automatically and it can be applied in distributed or share memory systems.

Based on logical programming, an orchestrator is computed by exploiting a reduction to satisfiability in a well known logic of programs [20]. The propositional dynamic logics (PDL) consists of a sequence of logics for representing the evolution of states and events of dynamic systems over time. PDL models are Kripke structures, where transitions between states or events are "labeled" by names of atomic programs. Notably, as it is shown in Figure 2.11, there is a description logic for each PDL logic which are tightly coupled, so there is a necessity for satisfiability checkers. FACT, Pellet and RacerPro are different highly optimized satisfiability checkers for this purpose. Moreover, a technique for linear time temporal logic (LTL) synthesis [31], based on model checking of game structures, called safety games [32], is dealt with it recently. Unfortunately, this approach has three major drawbacks:

- Only finite-state orchestrators are returned.
- The obtained solution is not flexible, that is, if a solution has been built, which relies on an available service and such service becomes unavailable at runtime, then the solution is no longer valid and the best one can do, with this approach,

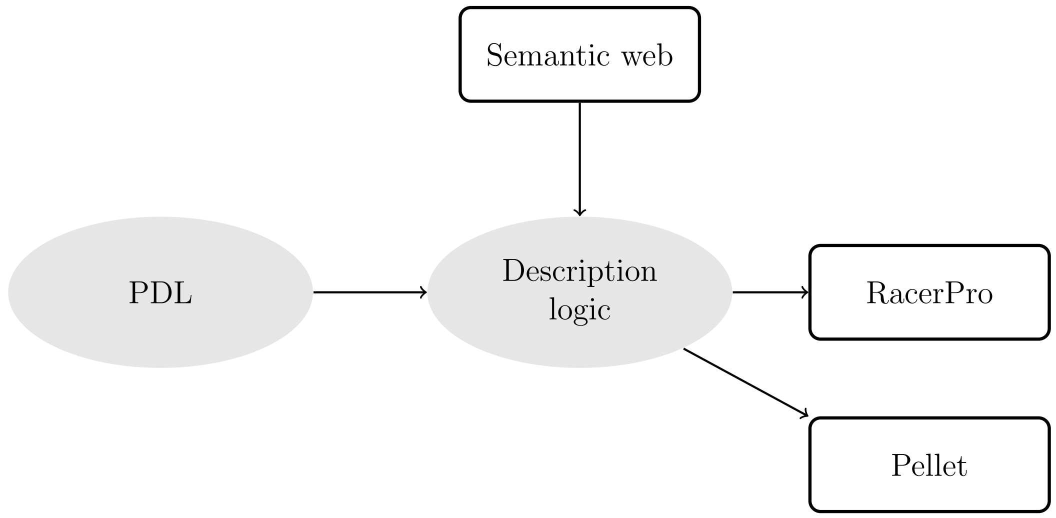

Figure 2.11: Description logic and satisfiability checkers relation

is to re-compute a new solution.

- On the practical side, due to implemented description logic reasoners' limitations, there is a possibility to synthesize a model only for some particular inputs, though it is complete with respect to checking for the existence of a model.

2.4 Conclusion

Different methods and approaches for fault tolerance have been introduced in this section. All of them have their own advantages and disadvantages. Our contribution consists in applying a supervisory strategy supported by Akka to handle internal exceptions during abnormal behavior executions and then take actions to trigger a forward or a backward recovery strategy.

Chapter 3

Proposed Approach: Integration of Backward Recovery into the Behavior Composition Framework

The behavior composition framework has been described in Section 1.1. One of the problems in the behavior composition framework, in the context of fault tolerance, is the long wait of a controller generator in order to choose a desired behavior if there is not any available behaviors to realize a target. Although, the behavior composition framework deals with unexpected failures mentioned in Section 1.1.4, by suitably refining the solution at hand, either on-the-fly or parsimoniously, the problem still exists. If a behavior fails and no other choices are possible, then the controller generator shall wait for the behavior to come back.

In order to eliminate these limitations and gain more reliability in behavior composition framework, an extendable approach for combining formal methods written in similar formal languages is proposed.

3.1 Colored Petri Net and Compensation Flow

Integrating another approach into the behavior composition framework, in order to solve the aforementioned problem in the framework, represents a great challenge. A general overview on different libraries and approaches in the context of fault tolerance is provided in Sections 2.2 and 2.3. The libraries are industrial-strength implementations, which typically have not been designed with respect to some formalisms. Hereafter, the aim is to apply a backward recovery technique mainly developed by using a formal approach. The technique mentioned in section 2.3.2 is the most suitable candidate for the proposed integration. In this technique, the control flow and the order of services execution, are generally represented in a Colored Petri Net (CPN) structure. As it is known, Petri nets are the main formal models used to describe static vision of a concurrent system and dynamic behavior of processes. Petri nets are also well suited to model internal operations of services and interactions among them as well as to model the processes in all phases of the service composition process [11]. The service composition process (executed by a Composer [11]) automatically discovers the services and their control flow, satisfying transactional properties that provide reliable executions. In related literature [12, 44], different transactional properties to recover the system in case of failures have been defined. The most used are the following. Let s be a service: s is *pivot* (p), if it fails it has no effect at all and allows backward recovery; s is *compensatable* (c), if it exists another service s', which can semantically undo the execution of s, then it allows semantic recovery; s is *retriable* (r), if s guarantees a successfully termination after a finite number of invocations, allowing forward recovery. The retriable property can be combined with properties p and c defining *pivot retriable* (pr) service, which allows backward and forward recovery and *compensatable retriable* (cr) service, which allows backward, forward and semantic recovery. Other properties are derived from the previous properties [11]. One of them according to the proposed integration is called atomic. A transactional composite web service is atomic ($\vec{a}$), if once all its web services component complete successfully, they cannot be undone, if one of them does not complete successfully, then backward recovery has to be done.

In the proposed integration, the notion of services in general is considered, however, it is usually refer to web services in the backward recovery. The global transactional property of a transactional composite web service (TCWS) allows recovery processes if a web service fails during the execution process. The following actions can be performed if a web service fails: retry the faulty web service, substitute the faulty web service, or compensate the executed web services. In consequence, these fault-tolerance mechanisms ensure the atomicity property of a TCWS with an all-or-nothing endeavor.

The approach leverages on automatically generated compensation flow of corresponding service composition at execution time in order to leave the system in a consistent state in the case of failure totally transparent. So, besides the TCWS, another CPN containing the compensation order for a backward recovery process is also generated automatically. Such a composition implies to understand several definitions. A query Q is defined in terms of functional conditions, QoS constraints and the required global transactional property as follows [13]. The ontology in the definition is a representation of artifact whose purpose is the exhibition on entities, defined classes and relations between them.

Definition 3.1.1. *[13] Query. Let $Onto_A$ be the integrated ontology (many ontologies could be used and integrated). A Query Q is a 4-tuple (I_Q, O_Q, W_Q, T_Q), where $I_Q = \{i \mid i \in Onto_A$ is an input attribute$\}$, $O_Q = \{o \mid o \in Onto_A$ is an output attribute whose value has to be produced by the system$\}$, $W_Q = \{(w_i, q_i) \mid w_i \in [0,1]$ with $\Sigma_i w_i = 1$ and q_i is a QoS criterion$\}$ and T_Q is the required transactional property: $T_Q \in \{T_0, T_1\}$. If $T_Q = T_0$, the system guarantees that a semantic recovery can be done by the user. If $T_Q = T_1$, the system does not guarantee the result can be compensated. In both cases, if the execution is not successful, no result is reflected to the system, i.e., nothing is changed on the system.*

A TCWS, which answers and satisfies a user query Q, is represented by a CPN. Formally it is defined as follows.

Definition 3.1.2. *[13] CPN-TCWS$_Q$. A CPN-TCWS$_Q$ is a 4-tuple (A, S, F, δ), where:*

— *A is a finite non-empty set of places, corresponding to input and output attributes of WSs in the TCWS such that $A \subset Onto_A$;*

— *S is a finite set of transitions corresponding to the set of web services (WSs) in the TCWS;*

— *$F : (A \times S) \cup (S \times A) \to 0, 1$ is a flow relation indicating the presence (1) or the absence (0) of arcs between places and transitions defined as follows: $\forall a \in A, (\exists s \in S \mid F(a,s) = 1) \Leftrightarrow$ (a is an input place of s) and $\forall s \in S, (\exists a \in A \mid F(s,a) = 1) \Leftrightarrow$ (a is an output place of s); this relation establishes the input and output execution dependencies among WSs component.*

— *δ is a color function such that $\delta : S \to \Sigma_S$ and $\Sigma_S = \{p, pr, \vec{a}, \vec{a}r, c, cr\}$ represents the transactional property (TP) of $s \in S(TP(s))$.*

As an example of a CPN-TCWS$_Q$, the reader can refer to Figure 3.1.

The global transaction property of CPN-TCWS$_Q$ ensures that if a web service, whose transaction property does not allow forward recovery, fails, then all previous executed web services can be semantically recovered by a backward recovery of the TCWS. For modeling TCWS backward and semantic recovery, another CPN associated to a CPN-TCWS$_Q$ is defined.

Definition 3.1.3. *[13] BRCPN-TCWS$_Q$. A BRCPN-TCWS$_Q$, associated to a given CPN-TCWS$_Q$=(A, S, F, δ), is a 4-tuple (A', S', F^{-1}, ζ), where:*

- *A' is a finite set of places corresponding to the CPN-TCWS$_Q$ places such that: $\forall a' \in A'$ $\exists a \in A$ associated to a' and a' has the same semantics as a.*
- *S' is a finite set of transitions corresponding to the set of compensation WSs in CPN-TCWS$_Q$ such that: $\forall s \in S,\ TP(s) \in \{c, cr\},\ \exists s' \in S'$ which compensates s.*
- *$F^{-1} : (A \times S) \cup (S \times A) \rightarrow \{0, 1\}$ is a flow relation establishing the restoring order in a backward recovery defined as: $\forall s' \in S'$ associated to $s \in S,\ \exists a' \in A'$ associated to $a \in A \mid F^{-1}(a', s') = 1 \leftrightarrow F(s, a) = 1$ and $\forall s' \in S', \exists a' \in A' \mid F^{-1}(s', a') = 1 \leftrightarrow F(a, s) = 1$.*
- *ζ is a color function such that $\zeta : S' \rightarrow \Sigma'_S$ and $\Sigma'_S = \{I, R, E, C, A\}$ represents the execution state of $s' \in S'$ (I for initial, R for running, E for executed, C for compensate and A for abandoned).*

As an example of a BRCPN-TCWS$_Q$, the reader can refer to Figure 3.2.

The marking of a CPN-TCWS$_Q$ or BRCPN-TCWS$_Q$ represents the current values of attributes that can be used for some web services to be executed or control values indicating the compensation flow, respectively. A marked CPN determines which transitions could be fired.

Definition 3.1.4. *[13] Marked CPN. A marked CPN=(A, S, F, δ) is a pair (CPN, M), where M is a function which assigns tokens (values) to places such that $\forall a \in A, M(a) \in \mathbb{N}$.*

According to marked CPN notations, for each $x \in (A \cup S)$, $(^\bullet x) = \{y \in A \cup S : F(y, x) = 1\}$ is the set of its predecessors, and $(x^\bullet) = \{y \in A \cup S : F(x, y) = 1\}$ is the set of its successors.

Definition 3.1.5. *[13] Fireable transition. A marking M enables a transition s iff all its input places contain tokens such that $\forall x \in (^\bullet s),\ M(x) \geq card(^\bullet x)$.*

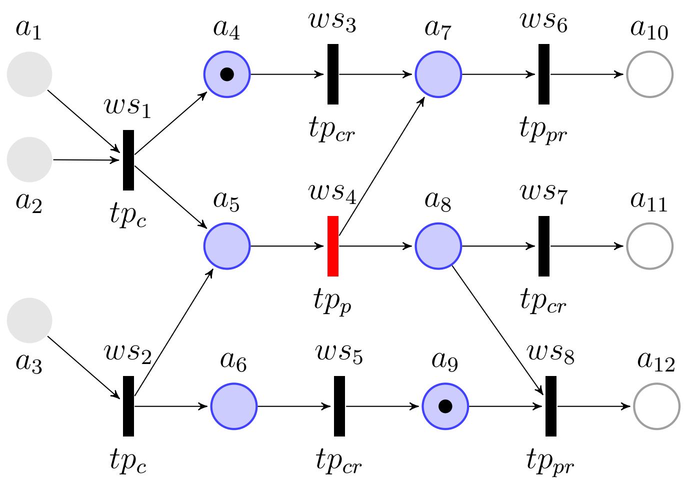

Figure 3.1: Marked CPN–TCWS$_Q$ when ws_4 fails

Depending on the initial marking, a transition is fireable (its corresponding web service can be invoked) only if all its predecessor transitions have been fired (see [13] for more details).

To support backward recovery, it is necessary to keep the trace of the service execution on the BRCPN-TCWS$_Q$. In case of any service, let say s fails, if $TP(s) \in \{pr, \vec{a}r, cr\}$, s is re-invoked until it successfully finishes or tries to replace the faulty service. Otherwise, a backward recovery is needed, i.e., all executed web services must be compensated in the inverse order they were executed.

To illustrate the backward recovery execution control, a small example is provided. Figure 3.1 shows the marked CPN-TCWS$_Q$. The corresponding BRCPN-TCWS$_Q$ is also shown in Figure 3.2. When ws_4 fails (see the red transition in Figure 3.1), the unfolding[1] of CPN-TCWS$_Q$ is halted and the initial marking on BRCPN-TCWS$_Q$, as it is given in Figure 3.3, is set to start the unfolding process, guided by Definition 3.1.6 and Definition 3.1.7. After ws'_3 and ws'_5 are fired and ws_7 is abandoned, a new marking is produced as presented in Figure 3.4.

1. Sequence of transitions.

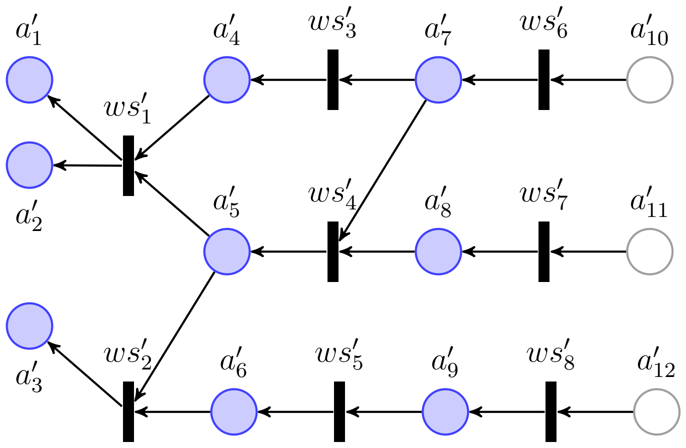

Figure 3.2: BRCPN–TCWS$_Q$

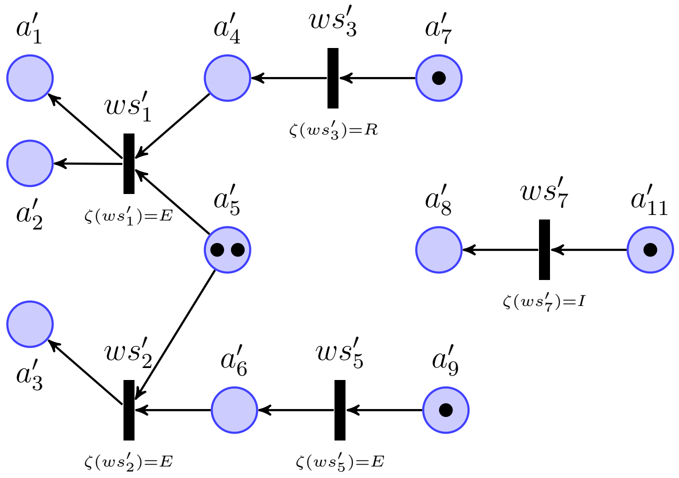

Figure 3.3: Initial marking of BRCPN–TCWS$_Q$

Definition 3.1.6. *[13] Fireable compensation transition. A marking M enables a transition s' iff all its input places contain tokens such that $\forall a' \in ({}^{\bullet}s')$, $M(a') \neq 0 \ \wedge \ \zeta(s') \notin \{A, C\}$.*

Definition 3.1.7. *[13] BRCPN-TCWS$_Q$ Firing rules. The firing of a fireable compensation transition (see Def 3.1.6) s' for a marking M defines a new marking M', such that:*

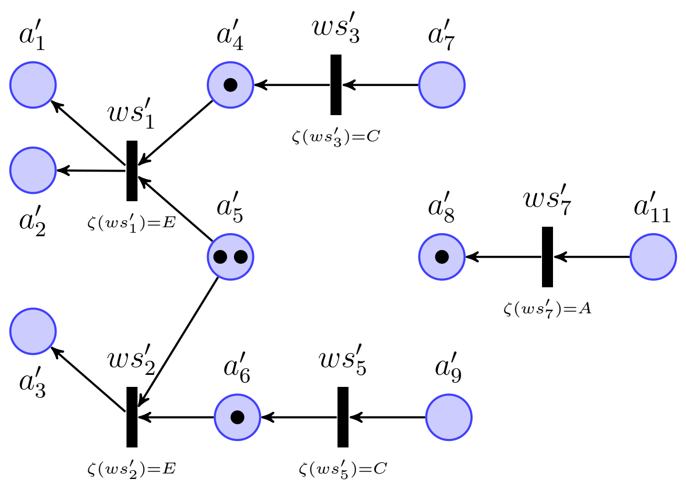

Figure 3.4: New marked BRCPN–TCWS$_Q$

— *if* $\zeta(s') = I,\ \zeta(s') \leftarrow A$ *(i.e., the corresponding* s *is abandoned before its execution);*

— *if* $\zeta(s') = R,\ \zeta(s') \leftarrow C$ *(in this case* s' *is executed after* s *finishes, then* s *is compensated);*

— *if* $\zeta(s') = E,\ \zeta(s') \leftarrow C$ *(in this case* s' *is executed, i.e.,* s *is compensated);*

— *all tokens are deleted from its input places* $(\forall x \in ({}^{\bullet}s'),\ M(x) = 0)$ *and tokens are added to its output places* $(\forall x \in (s'^{\bullet}),\ M(x) = M(x) + 1)$.

The compensation process is managed by an EXECUTER, which is a collection of components called EXECUTION ENGINE and ENGINE THREADS. Figure 3.5 depicts the overall architecture of the EXECUTER. These components are completely described by Cardinale and Rukoz in [13]. In brief, the TCWS execution and its collaboration with its peers are initiated, controlled and monitored by EXECUTION ENGINE and its ENGINE THREADS. One ENGINE THREAD is assigned to each web service in the TCWS. ENGINE THREAD is responsible to remotely invoke the web services component and set the state of the corresponding transition in BRCPN-TCWS$_Q$ to running (see definition 3.1.3). When a CPN-TCWS$_Q$ and the corresponding BRCPN-TCWS$_Q$ are received by the EXECUTION ENGINE, two dummy transitions are added to CPN-TCWS$_Q$:

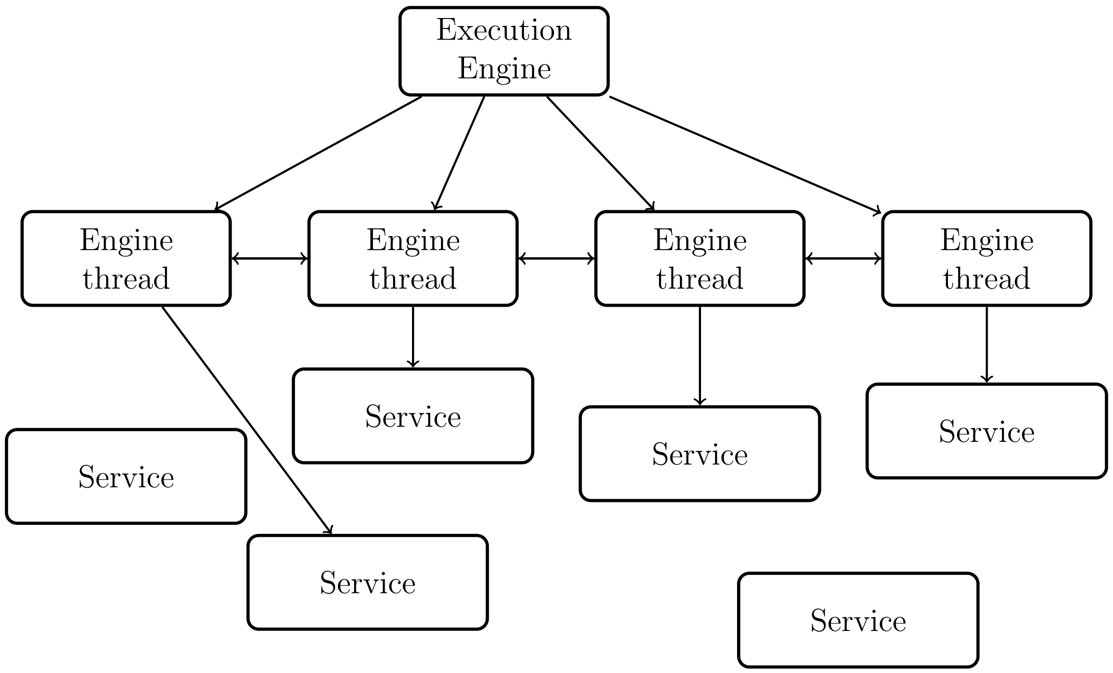

Figure 3.5: Executer architecture

— ws_{EE_i}, the first transition providing the inputs;

— ws_{EE_f}, the last transition consuming the outputs.

In the same manner, two transitions with inverse data flow relation (ws'_{EE_i} and ws'_{EE_f}) are added to BRCPN-TCWS$_Q$.

The compensation process is carried out by both Execution Engine and Engine Threads. Algorithm 1 describes the compensation protocol. In case of failure during the execution of web service, the responsible Engine Thread informs the Execution Engine and the Execution Engine marks the BRCPN-TCWS$_Q$ with the initial marking (line 4). Then, the Execution Engine sends a compensate message and control tokens to all Engine Threads (lines 6–7). When the Engine Threads receive the compensate message, the firing rules in Definition 3.1.7 is applied (lines 13–41). So, according to the provided example in Figure 3.3, the Engine Thread, which is responsible for ws'_3, waits for the corresponding web service to finishes, then ws'_3 fires and the corresponding execution state is set to C (lines 29–32), as shown in Figure 3.4. A responsible Engine Thread for ws'_5 fires ws'_5 and its

Algorithm 1 [13] Compensation Protocol

```
 1: begin
 2:     Execution Engine:
 3:     begin
 4:         $\forall a' \in A' \mid {}^{\bullet}a' = \emptyset,\ M(a') = 1 \land \forall a \in {}^{\bullet}s,\ M(a') = 1$;
 5:         /* Mark the BRCPN-TCWS$_Q$ with the Initial Marking*/
 6:         Send compensate to all ENGINE THREADS;
 7:         Send control values to ${}^{\bullet}({}^{\bullet}ws'_{EE_f})$;
 8:         Wait control values from $((ws'_{EE_i})^{\bullet})^{\bullet}$;
 9:         Return ERROR;
10:     end
11:
12:     Engine Threads:
13:     begin
14:         $ws' \leftarrow$ WS which compensate its WS;
15:         if $\zeta(ws') = A \ \lor\ \zeta(ws') = C$ then
16:             Send Control tokens to $Successors\_ETWS_{ws'}$
17:         else
18:             $InputsNeeded\_ETWS_{ws'} \leftarrow getInputs(WSDL_{ws'}, OWLS_{ws'})$;
19:             repeat
20:                 Wait Control tokens from $Predecessors\_ETWS_{ws'}$;
21:                 Set Control tokens to $InputsNeeded\_ETWS_{ws'}$;
22:             until
23:             $(\forall a' \in InputsNeeded\_ETWS_{ws'}, M(a') \neq 0)$;
24:             /* Wait its corresponding $ws'$ becomes fireable: $a'$
25:             has a control value and all transition predecessors have finished */
26:             if $\zeta(ws')$ = I then
27:                 $\zeta(ws') \leftarrow A$
28:             end if
29:             if $\zeta(ws')$ = R then
30:                 Wait $ws$ finishes;
31:                 Invoke $ws'$;
32:                 $\zeta(ws') \leftarrow C$;
33:             end if
34:             if $\zeta(ws')$ = E then
35:                 Invoke $ws'$;
36:                 $\zeta(ws') \leftarrow C$;
37:             end if
38:             Send Control tokens to $Successors\_ETWS_{ws'}$;
39:         end if
40:         Return /* ENGINE THREADS finishes */
41:     end
42: end
```

execution state is set to C (lines 34–37) and lastly, the execution state of ws'_7 is set to A (lines 26–28). The compensation process finishes when ws'_{EE_i} becomes fireable.

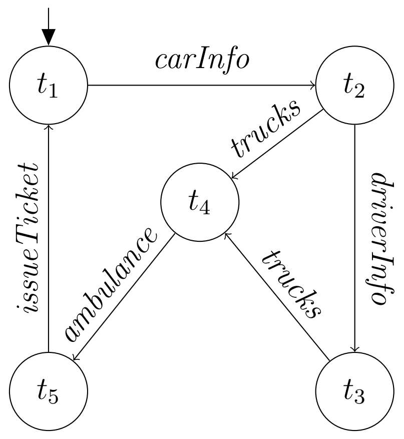

Figure 3.6: Target behavior β_T

However, backward recovery means that users do not get the desired answer to their queries, but it ensures system consistency and reliability.

3.2 A Case Study

With respect to the behavior composition framework, a real scenario is provided as a case study, which simulates a police officer task in a traffic accident. Consider a police officer who is responsible to prepare a report about an accident. The report contains driver and car informations. The police officer may call for an ambulance and tow trucks and finally issue a ticket for a driver violating a traffic law. There are three service behaviors and one target behavior to perform these actions. The actions are depicted in Figures 3.6 and 3.7: *carInfo* (ca) and *driverInfo* (di), to get the car and driver information, respectively, $trucks(tr)$ and $ambulance(am)$ to call tow trucks and an ambulance, respectively, and $issueTicket(ti)$ to issue a ticket for a driver.

Each behavior in the case study is running on a complete isolated linux server with the aid of LXC[2]. LXC is like a very lightweight virtualization, so lightweight means that there is no virtualization at all, and therefore no performance penalty.

2. LXC is a userspace interface for the linux kernel containment features. Through a powerful API and simple tools, it lets linux users easily create and manage system or application containers.

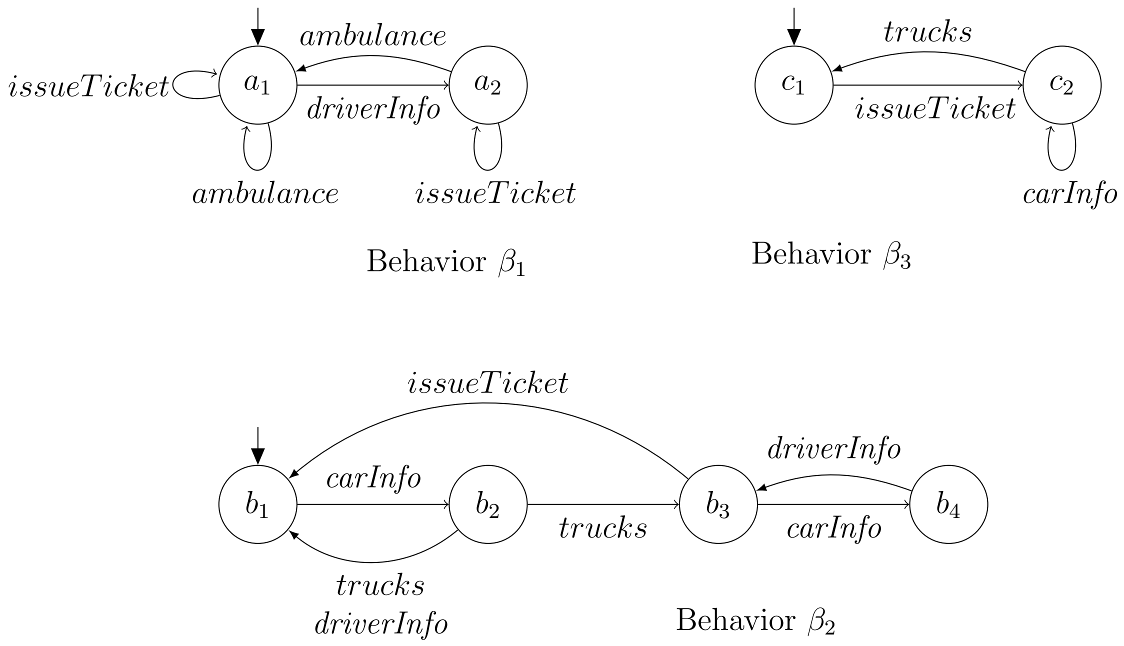

Figure 3.7: Available behaviors β_1, β_2 and β_3

Having considered such behaviors, a controller generator is shown in Figure 3.8. The controller generator is adjusted in the case of any failure in the behaviors. For instance, according to the controller generator given in Figure 3.8, if in the state $\langle a_1, b_3, c_1 \rangle$ a failure occurs in any of the behaviors β_1, β_2 or β_3, the controller generator is adjusted by filtering the failed behavior and based on the next action in target behavior (see Figure 3.6), which is *am*, the following measures are expected.

- There is not any choices available to realize *am*, so instead of waiting for a behavior to comes back, a backward recovery starts and β_2 will rollback *tr* and *ca* transitions.
- The controller generator is adjusted, as illustrated in Figure 3.9, and continues to realize target requests without β_2.
- The controller generator is adjusted, as shown in Figure 3.10, and continues to realize target requests without β_3.

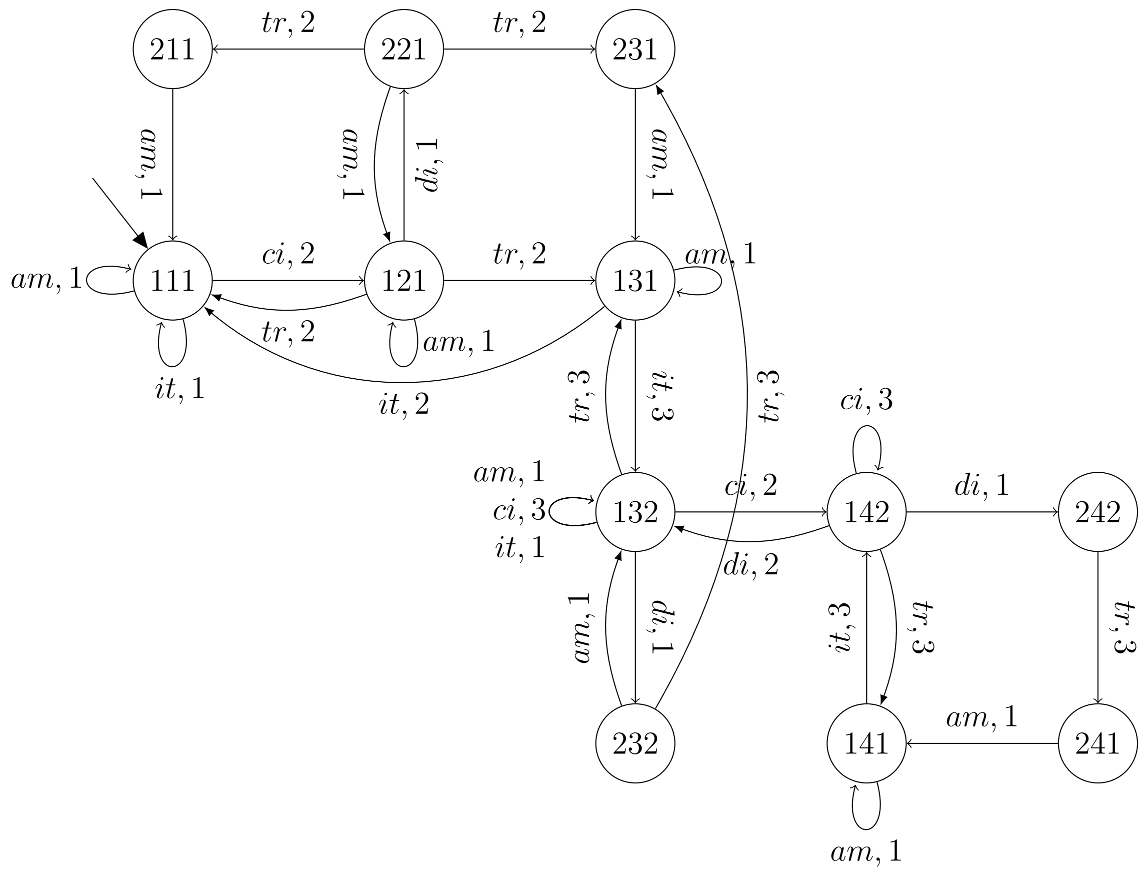

Figure 3.8: Controller generator for the case study

In the case study, Scala, as a general-purpose programming language, is used along with AKKA to simulate the system as several finite state machines. These state machines interact with each other by message passing. A AKKA cluster is also used to emulate the real environment.

3.3 Contribution

The main idea for the backward recovery in the proposed approach advocated in this thesis is inspired by the aforementioned backward recovery. In our approach the role of the EXECUTION ENGINE is given to an orchestrator to coordinate services. Service threads start the backward execution by the compensate message, but back-

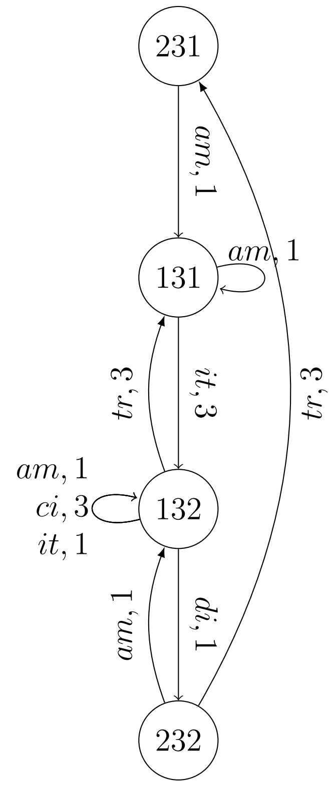

Figure 3.9: Controller generator without β_2

ward transitions are generated automatically at run time. So, in presence of failure, forward recovery is applying first. If there is not any choices to delegate an action, instead of waiting for the behavior to come back, with respect to the compensation process mentioned above, the system will be leaved in a consistent state by executing backward recovery.

In summary, this chapter makes the following contribution in the behavior composition framework:

— introduce a compensation flow to keep the trace of all transactions at run time and rollback the executed actions in the absence of available behavior in forward recovery;

— add a central node, called an orchestrator, to the framework to monitor transactions.

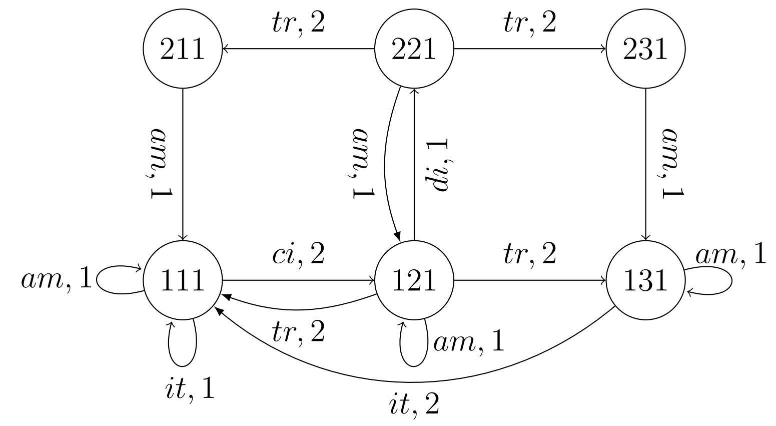

Figure 3.10: Controller generator without β_3

Chapter 4

Implementation of Forward and Backward Recovery in the Behavior Composition Framework

The implementation focuses on fault tolerance with respect to the behavior composition framework. In order to cope with failures and provide robust behavior composition, the forward recovery approach proposed in [18] is assessed, but to overcome its drawback mentioned in Chapter 3, a backward recovery technique is articulated and integrated into the behavior composition framework to provide a more adequate solution.

To the best of our knowledge, it is the first time that an implementation of the underlying interaction between components is provided with recovery techniques. The implementation works only for a specific case study, but it could be generalized to provide a commercial framework. The details about generalization is given in Section 4.5.

As a case study, the car accident scenario has been successfully implemented and the proposed approach has been adopted. The models used in this scenario are depicted in Figures 3.6 and 3.7. In Figure 3.6, β_T describes the deterministic behavior of a target. Observe that in state t_2, getting information about a driver is optional and depends on an internal choice in the target behavior. Figure 3.7 shows all nondeterministic behaviors: β_1 (with states a_1 and a_2) is able to get information about a driver, call an ambulance and issue a ticket; β_2 (with states b_1, b_2, b_3 and b_4) is able to perform all actions except calling an ambulance; and β_3 (with states c_1 and c_2) can get information about a car, call tow trucks and issue a ticket. In this example, it is assumed that all states are final.

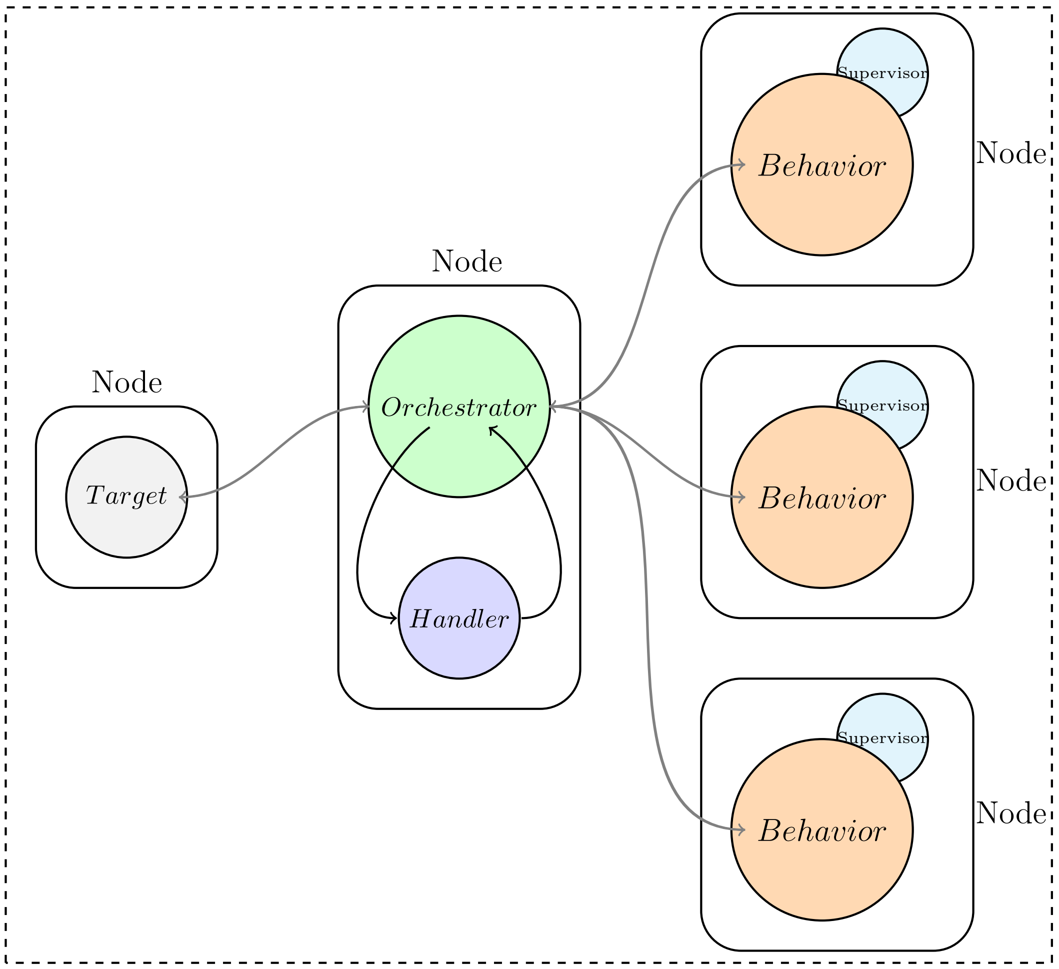

Figure 4.1: Overall architecture

4.1 Assumption

In this chapter, it is assumed that the actions are compensatable (reversible), so, when the unfolding process in the backward recovery is triggered, the actions are compensated eventually.

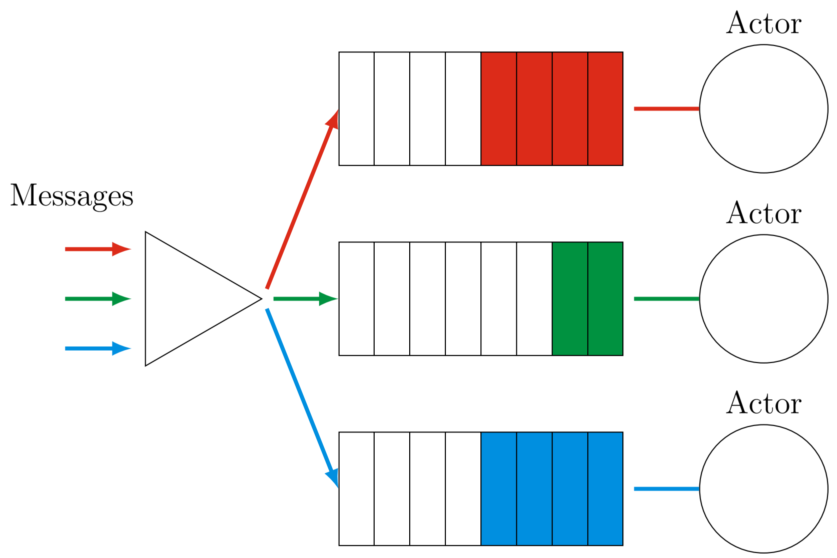

Figure 4.2: Actor maiblox

4.2 Overall Architecture

Before going into details, it is worth to introduce the overall architecture. A number of peer-to-peer Akka nodes is shown in Figure 4.1 as an overall architecture which comprise a target, an orchestrator and several service behaviors. Each node is deployed on a separate LXC container. The target node is assumed to be fully deterministic and stands for the behavior that the system as a whole needs to realize. An orchestrator coordinates services by exchanging messages and suitably controlling their activities and states. The service behavior nodes are partially controllable behaviors with a sort of nondeterministic transition systems. Each behavior is supervised and monitored by a separate actor in the same node, called supervisor. A supervisor is responsible for dealing with the failures that may arise in the corresponding behavior. All components are written in Scala and Akka is used as an underlying framework for message passing, supervision, cluster management and component interaction. The messages are passing between the nodes through the Akka mailbox. In Akka, the mailbox is a queue that holds the messages for an actor. As illustrated in Figure 4.2, there is usually a mailbox per actor. Though in some cases, where routing gets involved, there may only be one mailbox between a

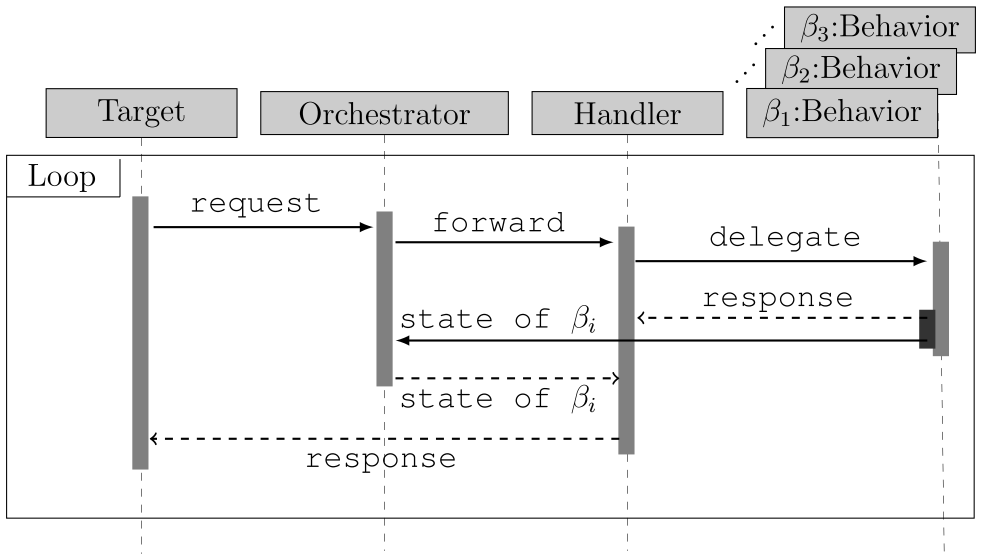

Figure 4.3: The sequence diagram for the service composition

number of actors. A mailbox is designed for communication in a peer-to-peer fashion. AKKA has several built–in routing strategies (e.g., *RoundRobinRoutingLogic*, *SmallestMailboxRoutingLogic*, *ScatterGatherFirstCompletedRoutingLogic*) to support this interaction automatically. In order to manage the relationships between components, an AKKA cluster is used. A cluster is made up of a set of member nodes. These nodes are logical members of a cluster. The identifier for each node is a "hostname:port:uid" tuple. The nodes interact with each other through an asynchronous message passing protocol using predefined communication channels. A cluster consists of a set of loosely or tightly connected nodes that work together so that, in many respects, they can be viewed as a single system. To keep the example simple, no environment (see Definition 1.1.1) is provided and it is assumed that the underlying network is reliable.

The AKKA cluster provides a fault-tolerant decentralized peer-to-peer based cluster membership service with no single point of failure or single point of bottleneck. It does this using gossip protocols and an automatic failure detector.

Figure 4.3 represents the sequences of interactions between components. Considering a target behavior that must be realized in accordance with a user requirements. When the target sends a request, an orchestrator creates a new handler and assigns it specifically to the target. Such an assignment causes all target requests forwarded to the same handler afterwards. The handler is an actor which has access to the controller generator and it is connected to one of each available behavior at a time. Whenever a behavior is connected to the handler, it is flagged as an engaged behavior in the list of behaviors in the orchestrator. This flag guarantees that each behavior is assigned to at most one handler. Then, the handler delegates the requested action to the behavior which is selected from the generated controller. The selected behavior performs the action and sends a response to the handler. At the same time, the behavior notifies the orchestrator about the state changes.

Subsequently, an orchestrator informs the handler about the changes. The handler collects information while maintaining the ability of backtrack. Finally, the response is sent to the target. Every behavior needs to subscribe to cluster changes, resubscribe and unsubscribe when the behavior starts, restarts and stops respectively.

Moreover, Akka introduces a feature to persist the behavior state, so that it can be recovered in order to resume the behavior after a failure. The technique is the most widely used in workflow persistence and recovery, i.e., periodic saving of a complete snapshot of the workflow's state.

4.3 Details about Different Phases in the Implementation

It is assumed that a controller generator has been synthesized and given as a transition system. A visual representation of a controller generator is shown in Figure 3.8. In brief, TLV/SMV, a model checker based on SMV, is invoked to synthesize a controller generator. Two input files are needed in such a tool: a SMV file which contains the SMV representation of behavior transitions and a proof script file. In this case study, the SMV file, called `car-accident.smv`, and a proof file, called `car-accident.pf`, is used to generate such a controller generator. The files and results of the model checking are provided in Appendix A. Notably, the result is converted to a JSON file for further usage in the system. The JSON file which is called *transition.json*, is also included in Appendix A.

According to the aforementioned message passing protocol, a common base trait (an alternative to interface in Java) for the messages is created. The messages must be immutable to avoid sharing mutable state. `Start` (see line 2) is used to start sending a request from the target to the orchestrator. Other messages (lines 3–7) are sent by the target during execution and flow in the system to be performed by the available behaviors.

```
1  sealed trait BCFMessage
2    case object Start extends BCFMessage
3    case object CarInfo extends BCFMessage
4    case object DriverInfo extends BCFMessage
5    case object Trucks extends BCFMessage
6    case object Ambulance extends BCFMessage
7    case object TicketIssue extends BCFMessage
```

With respect to the sequence diagram provided in Figure 4.3, the actual implementation is divided in three phases: startup, delegation and recovery.

4.3.1 The Startup Phase

In Akka all sensible configuration values are defined in a file, called *application.conf*, located in the resources directory. Typical examples of the configuration are:

— log level and logger backend;

— enable remoting;

— message serializers;

— definition of routers;

— tuning of dispatchers.

This means roughly that the default to start a system is to parse all properties in *application.conf*. For instance, the following configuration is used to enable the Akka cluster and join the `orchestrator` to the cluster. Line 2 declares the preferred log level. Lines 7–12 define hostname and port numbers for remote access and lines 14–20 subscribe an orchestrator to the cluster membership events. The last is done automatically by Akka.

```
akka {
  loglevel = "INFO"
  actor {
    provider = "akka.cluster.ClusterActorRefProvider"
    warn-about-java-serializer-usage = false
  }
  remote {
    log-remote-lifecycle-events=off
    netty.tcp {
      hostname = "10.44.102.25"
      port = 2551
    }
  }
  cluster {
    roles = ["orchestrator"]
    seed-nodes = [
      "akka.tcp://ClusterSystem@10.44.102.25:2551"
    ]
    log-info = off
  }
}
```

While constructing an actor system, it is possible to either pass these parameters in a `Config` object or use `ConfigFactory.load()` to overrides given properties. The latter is used in this example for creating the behaviors. For instance, a behavior, called `BehaviorA`, is initiated as follows. Lines 1 and 2 contain the input arguments which override a defined parameters in *application.conf* and lines 3–10 show how they are used in the actor initialization.

```
1   def initiate(name: String, port: Int, logLevel: String
2                , property: Property = Retriable, initialState: State = A1)= {
3     val conf = ConfigFactory.parseString(s"""
4       akka {
5         remote.netty.tcp.port=$port
6         loglevel = $logLevel
7       }""").withFallback(ConfigFactory.load().getConfig("BehaviorA"))
8
9     val system = ActorSystem("ClusterSystem", conf)
10    system.actorOf(Props(new BehaviorA(property, initialState)), name)
11  }
```

An orchestrator is started by running an object, called `AppRunner`. As it is shown below in line 1, `AppRunner` extends the `App` trait in Scala, which means that it is able to run directly from the command line. Line 4 creates an ActorSystem and this is the Akka container which contains all actors. An example of how to create actors in the container is the `system.actorOf(...)` in line 5. Notably, to create an actor inside other actors (actor context), `context.actorOf(...)` is used instead.

```
1  object AppRunner extends App {
2    val recoveryStack: mutable.Stack[RecoveryObject] = mutable.Stack();
3    val config = ConfigFactory.load().getConfig("Orchestrator")
4    val system = ActorSystem("ClusterSystem", config)
5    val orchestrator = system.actorOf(Props(classOf[Orchestrator]
6          , recoveryStack)
7          , UniqueNames.DISPATCHER)
8    println(" -----> What is the preferred interval between the requests:")
9    val interval:Long = scala.io.StdIn.readLine().toLong
10
11   val target = system.actorOf(Target.props(dispatcher, interval), UniqueNames.TARGET)
12   target ! Start
13 }
```

So, by running the above code from the command line, a preferred time interval between the requests is prompted in the console (see lines 8–9). A user entry is used by the target in order to send the requests in specific time intervals. Then, the `Target` is created in line 11 and the `Start` message is sent to the target (see line 12).

An orchestrator is created by extending `Actor`, `ActorLogging` and `Stash` traits.

```
  class Orchestrator extends Actor with ActorLogging with Stash {
   ...
  }
```

The `ActorLogging` is used for logging and the `Stash` trait enables an actor to temporarily stash away messages that cannot or should not be handled in the current actor's behavior.

The `Actor` trait defines a receive method as a message handler. It is expected to handle several messages which are listed below, using standard Scala pattern matching. A complete implementation of each case is provided in Appendix C.

```
  case ActorCompensated => ...
  case Terminated => ...
  case BehaviorRegistration => ...
  case ScheduleCompensation => ...
  case TargetRequest => ...
  case CurrentState => ...
  case Transition => ...
```

The behaviors are also started by running an object which extends the `App` trait, i.e., `BehaviorARunner`, `BehaviorBRunner` and `BehaviorCRunner`. In such objects, the number of instances, the behavior name, a preferred port number and a log level (i.e., debug, error, info or warn) are used to start the behaviors. The behaviors join the cluster and subscribe to cluster membership events through a similar aforementioned configuration file with different parameters values and an extra supervisory actor which is given below in line 5. With respect to the supervisory strategy, when a behavior detects a failure (i.e., throws an exception), it suspends itself and sends a message to its supervisor, signaling failure. Depending on the nature of the work to be supervised and the nature of the failure, the supervisor has a choice to stop, resume, restart and escalate the failure. For instance, the following configuration file is used for the `behaviorA`.

```
 1  akka {
 2    loglevel = "INFO"
 3    actor {
 4      provider = "akka.cluster.ClusterActorRefProvider"
 5      guardian-supervisor-strategy= "ca.sherbrooke.actor.Supervisor"
 6      warn-about-java-serializer-usage = false
 7    }
 8    remote {
 9      log-remote-lifecycle-events=off
10      netty.tcp {
11        hostname = "10.0.3.119"
12        port = 0
13      }
14    }
15    cluster {
16      roles = ["behaviorA"]
17      seed-nodes = [
18        "akka.tcp://ClusterSystem@10.44.102.25:2551"
19      ]
20      log-info = off
21    }
22  }
```

All behaviors are created by mixing the following traits: `FSM`, `ActorLogging`, `Stash` and `Registration`.

```
abstract class BehaviorA extends FSM[State, Data]
                                      with Stash
                                      with ActorLogging
                                      with Registration{}
```

The `FSM` trait specifies possible states and data values. The following states are defined in this case study and is provided in Appendix B. `A1` and `A2` is used in `behaviorA`; `B1`, `B2`, `B3` and `B4` is used in `behaviorB`; `C1`, `C2` is used in `behaviorC`. The states of the `Target` are also defined as: `T1`, `T2`, `T3`, `T4` and `T5`.

The `Registration` trait as given below is used in the behavior subscription along with the behavior name and its initial state (see lines 2 and 3), which are required in an orchestrator. A `BehaviorRegistration` message, in line 11, is sent to an orchestrator for subscription through a `register` method in the behavior (see lines 6–14).

```
1   trait Registration {
2     def getBehaviorName(): String
3     def getInitialState(): Constants.State
4   }
5
6   def register(member: Member) = {
7     if (member.hasRole(UniqueNames.DISPATCHER_ROLE)) {
8       log.info(SendRegistrationRequest.msg, self.path.name)
9       context.actorSelection(RootActorPath(member.address)
10                / "user" / UniqueNames.DISPATCHER) !
11                BehaviorRegistration(getBehaviorName()
12                                , stateName, getInitialState())
13    }
14  }
```

When an orchestrator receives a `BehaviorRegistration` message, it stores the behavior's name, current state and initial state (see lines 11–12 in the above code) in a memory. Then, with the following methods, an orchestrator starts watching the behavior and subscribes to the behavior transition callback.

```
context.watch(actor)
actor ! SubscribeTransitionCallBack(self)
```

4.3.2 The Delegation Phase

Whenever an orchestrator receives a `TargetRequest`, a following case statement in `receive` method in an orchestrator is triggered.

The effect of this method call is the creation of a new handler to forward a request to it, or just forwarding a request to the handler which is already created. So, first of all, the existence of the handler in a predefined *ConcurrentHashMap*, called `targetHandlerMap`, is checked as shown in line 5 in the code on the given below. If it exists, a log message is printed and a request is forwarded to it (lines 6–9). If the handler does not exist, a new handler is created and assigned to the target. For such a purpose, an orchestrator changes its current internal behavior to `ReceiveHandlerActorMessage` in order to stash (push in a stack) any further messages and temporarily stash away them while a new handler is creating (line 10). Such a change in the internal behavior causes an orchestrator to handle concurrency. The available registered behaviors are searched in lines 11–33. They are assigned to the new handler in lines 34–41, if they are not already engaged and their current state is equal with their initial state.

```
1  case TargetRequest(targetState: State, action: Action, restart) => {
2    val entry = targetHandlerMap.entrySet().stream()
3          .filter(entry => entry.getKey == sender())
4          .findAny()
5    if (entry.isPresent) {
6     log.info(ForwardMessage.msg, entry.get().getValue.path.name)
7     val actor = entry.get().getValue
8     actor forward Request(targetState, action, restart)
9    } else {
10     become(ReceiveHandlerActorMessage)
11     val behaviorAEntry = behaviorARoutees.entrySet()
12                 .stream()
13       .filter(elem =>
14         elem.getValue
15         .currentState == elem.getValue.initialState)
16       .filter(elem =>
17         elem.getValue.isAvailable).findAny()
18     behaviorAEntry.ifPresent(e => e.getValue.isAvailable = false)
19     val behaviorBEntry = behaviorBRoutees.entrySet()
20                 .stream()
21       .filter(elem =>
22         elem.getValue
23         .currentState == elem.getValue.initialState)
24       .filter(elem =>
25         elem.getValue.isAvailable).findAny()
26     behaviorBEntry.ifPresent(e => e.getValue.isAvailable = false)
27     val behaviorCEntry = behaviorCRoutees.entrySet()
28                 .stream()
29       .filter(elem =>
30         elem.getValue
31         .currentState == elem.getValue.initialState)
32       .filter(elem => elem.getValue.isAvailable).findAny()
33     behaviorCEntry.ifPresent(e => e.getValue.isAvailable = false)
34     val handlerActor = system
35                 .actorOf(HandlerActor.props(
36        if (behaviorAEntry.isPresent) Some(behaviorAEntry.get())
37        else None
38       , if (behaviorBEntry.isPresent) Some(behaviorBEntry.get())
39         else None
40       , if (behaviorCEntry.isPresent) Some(behaviorCEntry.get())
41         else None, self)
42       , sender().path.name + UniqueNames.HANDLER)
43     log.info(HandlerActorCreatedMessage.msg
44                         , handlerActor.path.name)
45     context.watch(handlerActor)
46     targetHandlerMap.put(sender(), handlerActor)
47     behaviorAEntry
48      .ifPresent(e => e.getValue.handlerActor = Some(handlerActor))
49     behaviorBEntry
50      .ifPresent(e => e.getValue.handlerActor = Some(handlerActor))
51     behaviorCEntry
52      .ifPresent(e => e.getValue.handlerActor = Some(handlerActor))
53     handlerActor !
54 InitializeRequest(targetState, action, restart, sender())
55   }
56 }
```

Then, an orchestrator starts watching the handler in line 45 and lastly sends a `InitializeRequest` message to the handler. The handler sends back an acknowledge message, called `OK`, as given below. As soon as the acknowledgement is received, an orchestrator unstashs (pop from stack) the stash messages (see line 4) and prepends them to the actors mailbox in the same order as they have been received originally.

```
1   def ReceiveHandlerActorMessage: Actor.Receive = {
2     case OK =>
3       log.info(HandlerActorMessage.msg)
4       unstashAll()
5       become(receive)
6   }
```

Next, once a request is forwarded to the handler, the handler extracts a controller by traversing a `controllerGenerator` and filter the nodes as follows. Line 2 filters the nodes by the current state of a target. Lines 3 and 4 filter them by a requested action and their activeness respectively. Finally, from lines 5 to 16, nodes are filtered by the current state of the behaviors assigned to the handler.

```
1   controllerGenerator
2       .filter(node => node.targetState == targetState)
3       .filter(node => node.action == action)
4       .filter(node => node.isActive == true)
5       .filter(node => node.behaviorAState ==
6         (if (behaviorAEntry.isDefined)
7           behaviorAEntry.get.getValue.currentState
8          else node.behaviorAState))
9       .filter(node => node.behaviorBState ==
10        (if (behaviorBEntry.isDefined)
11          behaviorBEntry.get.getValue.currentState
12         else node.behaviorBState))
13      .filter(node => node.behaviorCState ==
14        (if (behaviorCEntry.isDefined)
15          behaviorCEntry.get.getValue.currentState
16         else node.behaviorCState))
```

The last step is delegating an action to the behavior asynchronously and returns a Future[1], which represents a possible response from the behavior (see lines 1–2). A response will be send asynchronously to the target (see line 4) and at the same time an orchestrator is notified on behavior transition. The last is done implicitly according to the subscription of an orchestrator to the cluster events.

1. A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion and to retrieve the result of the computation.

```
behaviorA ? AskRequest(action) map {
  case Result(res) =>
    log.info(BehaviorResult.msg, res, targetRef.path.name)
    targetRef ! Constants.TargetResult
}
```

Upon the behavior transition, an orchestrator sends an object to the handler that wraps the reference, old state and new state of the behavior. The object is pushed to a recovery state holder, called `recoveryStack`, which is created in the handler to keep track of all transitions. The `recoveryStack` will be changed by other behaviors during execution and it is used for backward recovery in recovery phase.

```
recoveryStack.push(RecoveryObject(actor, oldState, newState))
```

The backward recovery is started by calling a `compensate` method in the handler, when there is not any available behavior to realize the target request. The compensation is explained in the next section.

```
def compensate: Unit = {
  (...)
}
```

4.3.3 The Recovery Phase

The recovery phase is divided in two sub-phases, forward and backward recovery. Traditionally, in case of any failure, the service center (orchestrator) needs to try different providers until it finds an appropriate one and then call a service. Along the lines of the concepts found in Section 1.1.4, forward recovery is applied implicitly in the system. It is done automatically on the fly by unfolding an algorithm which traverse a controller generator nodes and revise it into a more efficient version by removing all unavailable nodes and related transitions. This is done by two case statements of the `receive` method in the handler, called `BehaviorTerminated` and `BehaviorTransition`. For instance, in the case of behavior termination in the following statement, the behavior reference is extracted in line 2 and availability property of any object in `recoveryStack`, which has reference to the terminated behavior, changes to `false`. Afterwards, from lines 5–15, the terminated behavior is also removed from the controller generator to stop any further assignments and delegations.

```
case BehaviorTerminated(ref) => {
  val terminatedBehavior = ref.path.name
  recoveryStack.filter(obj => obj.ref == ref)
               .foreach(obj => obj.isAvailable = false)
  terminatedBehavior match {
    case name if name.startsWith(UniqueNames.BEHAVIOR_A) =>
      controllerGenerator.filter(node => node.behavior == A)
                         .foreach(node => node.isActive = false)
    case name if name.startsWith(UniqueNames.BEHAVIOR_B) =>
      controllerGenerator.filter(node => node.behavior == B)
                         .foreach(node => node.isActive = false)
    case name if name.startsWith(UniqueNames.BEHAVIOR_C) =>
      controllerGenerator.filter(node => node.behavior == C)
                         .foreach(node => node.isActive = false)
  }
}
case BehaviorTransition(actor:ActorRef, oldState: State, newState: State) => {
  log.info(Transition.msg, actor.path.name, oldState, newState)
  actor.path.name match {
    case name if name.startsWith(UniqueNames.BEHAVIOR_A) =>
      applyTransition(A, actor, oldState, newState)
    case name if name.startsWith(UniqueNames.BEHAVIOR_B) =>
      applyTransition(B, actor, oldState, newState)
    case name if name.startsWith(UniqueNames.BEHAVIOR_C) =>
      applyTransition(C, actor, oldState, newState)
  }
  sender() ! OK
}
```

Unlike the other recovery techniques presented in Section 2.3, which had deterministic transitions and the corresponding backward transitions are generated before the execution phase started, in the suggested approach the backward transitions are generating at runtime with the aid of `recoveryStack` to support nondeterministic transitions.

The backward recovery phase is the last option in the fault tolerance implementation. It is first carried out by the handler and if it is needed, an orchestrator will be involved. If the handler could not find any behavior to delegate an action due to the forward recovery process, a backward recovery is started by invoking a method, called `compensate`.

So, in the piece of code given on the next page, first, in line 2, the `recoveryStack` is checked. If it is not empty, the last object is popped up (see line 6). The popped object contains a reference to the behavior which has performed an action. The behavior availability is checked by the handler in line 7. If the behavior is available, after pattern matching, a `Compensate` message is sent to the behavior through the lines 12, 18 and 24 to roll back the performed action. Otherwise, a message, called `ScheduleCompensation` is sent to an `orchestrator` in line 38. An orchestrator starts a scheduler to send the same message to the behavior in specific intervals. Sending in specific intervals improves consistency and establish the repeatability condition which is required for proving the approach's soundness.

```
def compensate: Unit = {
  if (recoveryStack.isEmpty) {
    log.info(SuccessfullyCompensated.msg, self.path.name)
    orchestrator ! ActorCompensated
  } else {
    val obj = recoveryStack.pop()
    if (obj.isAvailable) {
      if(obj.to.property == Compensatable) {
        log.info(CompensateMsg.msg, obj)
        obj.to match {
          case A1 | A2 => {
            behaviorA ? Compensate(obj) map {
              case CompensatedSuccessfully =>
                compensate
            }
          }
          case B1 | B2 | B3 | B4 => {
            behaviorB ? Compensate(obj) map {
              case CompensatedSuccessfully =>
                compensate
            }
          }
          case C1 | C2  => {
            behaviorC ? Compensate(obj) map {
              case CompensatedSuccessfully =>
                compensate
            }
          }
          case to =>
            log.info(UnableToCompensateMsg.msg, to)
        }
      } else {
        log.info(NotCompensatableMsg.msg, obj, obj.to.property)
      }
    } else {
      log.info(CompensationOnScheduleMsg.msg, obj.ref.path.name)
      recoveryStack.push(obj)
      orchestrator ! ScheduleCompensation(recoveryStack)
    }
  }
}
```

4.4 Experiments

In order to validate the implementation and test a system reaction in a normal situations and different failures (i.e., killing, freezing, unfreezing and throwing an exception), three LXC containers have been created and started. Running the following command lists all containers created in a system.

```
sudo lxc-ls --fancy

NAME              STATE    IPV4        IPV6  AUTOSTART
-----------------------------------------------------
node-behavior-A  RUNNING  10.0.3.119  -     NO
node-behavior-B  RUNNING  10.0.3.242  -     NO
node-behavior-C  RUNNING  10.0.3.241  -     NO
```

By default, log messages are printed to STDOUT in Akka. Logging is performed asynchronously to ensure that logging has minimal performance impact. Logging generally means IO and locks, which can slow down the operations of the code if it was performed synchronously. Normally, Akka logs messages includes the following lines, which are displayed in each log message. The first part in line 1 is a log level, the second part is a date and time, line 2 is the default name assigned by Akka to each actor and the last line is the identifier for each node. For simplicity, it is ignored in the rest of log messages.

```
1  [INFO] [01/15/2018 16:00:40.787]
2  [ClusterSystem-akka.actor.default-dispatcher-16]
3  [akka.tcp://ClusterSystem@10.0.3.119:2553]
```

Akka has a few configuration options for very low level debugging. These make more sense in development than in production. This config option exists in *application.conf* file and four methods (i.e., `ERROR`, `WARNING`, `INFO` and `DEBUG`) could be set as illustrated in the following box.

```
akka {
  loglevel = "DEBUG"
}
```

The log message may contain argument placeholders, which will be substituted if the log level is enabled. For instance, to log message about the current states, a `CurrentStates` with three placeholders is defined as follows. The placeholders could be substituted with accurate values in the code. In our case study, all log messages are defined in the *Common.scala* file which is provided in Appendix D.

```
 case object CurrentStates extends HandlerActorLogMessage {
    override def content: String = "Current states are {}, {}, {}."
 }
```

Starting an *Orchestrator* on a main machine, a preferred time interval between requests is prompted and then the following messages prints out.

```
Starting remoting
Remoting started; listening on addresses
        :[akka.tcp://ClusterSystem@10.44.102.25:2551]
integration-dispatcher PreStart Hook....
```

Now, it is a time to start the behaviors on each node. Three behaviors, named `behaviorATest`, `behaviorBTest` and `behaviorCTest` are created on the created nodes on ports: `2553`, `2554`, `2555` respectively. For instance, running the `BehaviorARunner` in the console prompts the following information: number of instances, behavior name, port number and preferred log level. Then, it starts the behavior, named `behaviorATest`.

```
[info]  -----> How many Instances:
1
[info]  -----> What is the 1th behavior name:
Test
[info]  -----> Which port:
2553
[info]  -----> What is the preferred Log level:
[info]  -----> DEBUG > ERROR > INFO > WARN :
DEBUG
```

A success message as given below prints out in the console, afterwards. The same output with different behavior name is expected on starting each behavior.

```
---------------------------- START BEHAVIOR LOG ---------------
behaviorATest is going to send a registration request.
---------------------------- END OF BEHAVIOR LOG ---------------
```

While the behaviors are starting, the registration message on orchestrator console is printed.

```
--------------- START DISPATCHER LOG --------------------------
behaviorATest is Registered successfully in integration-dispatcher
--------------- END OF DISPATCHER LOG --------------------------

--------------- START DISPATCHER LOG --------------------------
behaviorATest is in state A1
--------------- END OF DISPATCHER LOG --------------------------
```

By giving 30 000 milliseconds for the time interval between the requests, a target starts sending a request to an orchestrator.

```
-- START TARGET LOG -------------------------------------------
T1 is sending a request for car information.
-- END OF TARGET LOG -------------------------------------------

--------------- START DISPATCHER LOG --------------------------
targethandlerActor is created to handle the request....
--------------- END OF DISPATCHER LOG --------------------------

----------------------------------------- START HANDLER LOG ----
 Current states are A1, B1, C1.
----------------------------------------- END OF HANDLER LOG ----

----------------------------------------- START HANDLER LOG ----
B is selected for CarInfo
----------------------------------------- END OF HANDLER LOG ----

--------------- START DISPATCHER LOG --------------------------
Handler Actor created successfully.
--------------- END OF DISPATCHER LOG --------------------------

----------------------------------------- START HANDLER LOG ----
Car information request has been done, states moving from B1 to B2
        and result is sending to target
----------------------------------------- END OF HANDLER LOG ----

---------------------------- START BEHAVIOR LOG ---------------
behaviorBTest is going from B1 to B2.
---------------------------- END OF BEHAVIOR LOG ---------------
```

Following, the execution prints related log messages which is provided in Appendix E. In a nutshell, the following groups of messages are printing in the console by the target, handler and behavior respectively, on each request.

```
T2 is sending a request for trucks.
Current states are A1, B2, C1.
B is selected for Trucks.
behaviorBTest is going from B2 to B3.

T4 is sending a request for ambulance.
Current states are A1, B3, C1.
A is selected for Ambulance.
behaviorATest is going from A1 to A1.

T5 is sending a request for issue the ticket.
Current states are A1, B3, C1.
B is selected for TicketIssue.
behaviorBTest is going from B3 to B1.
```

Testing system reaction to different failures required to create another object which is also runnable through the command line, called *CommandRunner*. Running this object prints out the following options to select and affect a normal behavior of the system.

```
-----> Please enter the commands number:
-----> 1- Kill
-----> 2- Freeze
-----> 3- Unfreeze
-----> 4- Throw Exception
```

Selecting any of the commands given above lists all available behaviors to be affected. So, in our case study, it is as follows.

```
0- behaviorATest
1- behaviorBTest
2- behaviorCTest
```

The first command in the *CommandRunner*, named `Kill`, causes the behavior to terminate and leave the cluster. To show the system reaction and log messages, it is assumed that the controller generator is in state $\langle a_1, b_3, c_1\rangle$ and the target is going to send a *it* request. Killing the `behaviorBTest` results in removing it from the controller generator and prints the following message.

```
---------------------------- START BEHAVIOR LOG ---------------
behaviorBTest is terminated and controller generator is going to adjust.
---------------------------- END OF BEHAVIOR LOG ---------------
```

So, according to the Figure 3.8, the request must be sent to the `behaviorCTest` as an alternative. In this situation, the following output is expected.

```
T5 is sending a request for issue the ticket.
Current states are A1, B3, C1.
C is selected for TicketIssue.
Ticket issue request has been done,
        states moving from C1 to C2 and result is sending to target.
behaviorCTest is going from C1 to C2.
```

Continuing this scenario, the next coming requests from the target would be *ci*, *tr*, *am* and *it*. The first two requests are handled by `behaviorCTest` and the third one is handled by `behaviorATest`. In such a state ($\langle a_1, b_3, c_1\rangle$), if the `behaviorCTest` kills, no alternative behavior is available to fulfill the fourth request, so, due to the lack of behaviors to delegate an action, the *Compensation* starts and a `compensate` method is call. The following messages is printed afterwards, in the console.

```
1  behaviorCTest is terminated and controller generator is going to adjust.
2
3  T5 is sending a request for issue the ticket.
4  Current states are A1, B3, C1.
5
6  There are no Candidate Controllers for TicketIssue
7          and the recovery stack is Stack(RecoveryObject(behaviorATest,A1,A1,true)
8          , RecoveryObject(behaviorCTest,C2,C1,false)
9          , RecoveryObject(behaviorCTest,C2,C2,false))
10
11 RecoveryObject(behaviorATest,A1,A1,true) is going to compensate.
12 behaviorATest is going from A1 to A1.
13 behaviorCTest is not available and compensation message is scheduled to send later.
```

Line 1 in the above log indicates that `behaviorCTest` is terminated and lines 6–9 show the objects on top of `recoveryStack`. The first object is popped up from the stack and compensated as shown in line 11. Line 13 indicates that the second object is popped up, but the `behaviorCTest` is not available and it is scheduled to send the compensation message later.

The `Freeze` command effect is similar to the `Kill` command except the behavior remains in the cluster environment and could be rejoin by the `Unfreeze` command. The `Unfreeze` command gives a chance to declare the preferred state in the behavior to resume.

A few well-known exception types has been chosen in order to demonstrate the application of the fault handling directives described in supervision. Selecting the last option from the listed commands after running *CommandRunner* prints out these exception types.

```
ResumeException
StopException
RestartException
Other Exception
```

The supervisor is configured with a function, translating all possible failure causes (i.e. exceptions) into one of the four choices given in lines 10, 16, 22 and 28. Resume, Stop, Restart are well described in Section 2.2.6. Escalate is used if the defined strategy does not cover the exception that was thrown.

```
1  class Supervisor extends SupervisorStrategyConfigurator {
2
3   import scala.concurrent.duration._
4
5   val symbol: String = "SUPERVISOR"
6
7   override def create(): SupervisorStrategy =
8              OneForOneStrategy(maxNrOfRetries=2
9                , withinTimeRange = 1 second) {
10    case ResumeException => {
11      println(s"\n $symbol " +
12        s"\n The behavior is going to Resume. " +
13        s"\n $symbol")
14      Resume
15    }
16    case StopException => {
17      println(s"\n $symbol " +
18        s"\n The behavior is going to Stop. " +
19        s"\n $symbol")
20      Stop
21    }
22    case RestartException => {
23      println(s"\n $symbol " +
24        s"\n The behavior is going to Restart. " +
25        s"\n $symbol")
26      Restart
27    }
28    case _:Exception => {
29      println(s"\n $symbol " +
30        s"\n The behavior is going to Escalate the Exception. " +
31        s"\n $symbol")
32      Escalate
33    }
34  }
35 }
```

In the above code, `maxNrOfRetries` and `withinTimeRange` properties are set to 2 and 1 respectively in lines 8–9, which means the strategy restarts a child up to 2 restarts per second. The child actor is stopped if the restart count exceeds `maxNrOfRetries` during the `withinTimeRange` duration.

Several test scenarios with respect to the case study have been done to check all possible malfunctioning of behaviors and the corresponding recovery procedures. The observed results are conformed to the expected reactions.

4.5 Generalization of the Implementation

The car accident scenario is an early sample of implementation in the context of the behavior composition framework. It is a prototype to provide specifications for a real, working system rather than a theoretical one and generally used to evaluate a forward and backward recovery techniques. Although, the forward and backward methods are implemented specifically for this example, it is possible to make code more general by removal of special-case conditionals and excessive hard coded details.

In this example, the actors are created with constant states, actions and names. For instance, the states of the target behavior have been defined within the object, named *Constants*, in a Scala file, called *commom.scala*. In this file all actions, messages, names and states of the behaviors and the actors used in the example are hard coded. The file is provided in Appendix B.

So by this definition, generalization could be achieved by apply the following rules. One approach could use the `ActorSystem.actorOf` method that takes an `ActorSystem` and some `Props` as constructor arguments to construct an actor in startup phase. The method is used in starting up the behaviors in Section 4.3.1. But, the main point is reading the arguments from an external file instead of aforementioned file with hard coded values for states, actions, messages and names. For such a purpose as mentioned in Section 4.3.1, since sensible values are provided in *application.conf*, the settings can be amended to change the default behavior or adapted for specific runtime environments to be used in the code on demand.

For instance, the following properties could be appended to the default configuration of the `BehaviorA` in *application.conf*. It is notable, providing values for these properties depends on the project rules and protocols.

```
BehaviorA {
  akka {
    (...)
  }
  name {
    default = behaviorA
  }
  states {
    initial= A1
    first= A1
    second= A2
  }
  actions {
    A1_1 = DriverInfo
    A1_2 = Ambulance
    A1_3 = TicketIssue
  }
}
```

The next approach is more and less similar with the first one, but with respect to the separation of concerns principle. Separate files, named "application.json" or “application.properties”, could be used in the root of the class path, as in Akka, the default is to parse all files found from the root of the class path.

So, providing these properties in an external files and parsing them on application startup or on demand makes the implementation more general for being used in other scenarios and use cases.

Conclusion

In this thesis, we have focused on the problem of fault tolerance in relation with service composition in the context of the behavior composition framework. Different formal and informal approaches and methods have been investigated with the aim to define a formal strategy to overcome obstacles met by the behavior composition framework in the case of failures in which the controller is compelled to wait for failed services in order to continue to operate and realize a target behavior. In particular, an integrated approach has been adopted with the capabilities of recovering the system into a consistent state in order to increase system reliability. Hence, the integration of backward recovery seems more appropriate in critical situations.

A prototype has been developed with the aid of Akka and an experiment has been conducted from a case study to validate the proposed approach. With respect to Akka, other development frameworks, such as JADE, do not provide all the facilities (e.g., service cluster management, finite state machines, supervisory control of services, automatic fault detection mechanisms) required in the prototyping of fault-tolerant service applications based on a formal method for service composition.

This work can be seen as a starting point for a deeper investigation of several aspects of system reliability and robustness in the behavior composition framework. For instance, it should be evaluated in a multitude varieties of real situations that involve web applications, which generally involve service composition.

Appendix A

A.1 SMV Code for the Case Study

```
MODULE main
VAR
  env: system Env(sys.index);
  sys: system Sys;
DEFINE
  good := (sys.initial & env.initial) | -- intial state is "good" by definition
                      !(env.failure); -- services are always required not to fail

MODULE Sys
VAR
 index : 0..3; -- num of services, 0 used for init
INIT
  index = 0
TRANS
  case
    index=0 : next(index)!=0;
    index!=0 : next(index)!=0;
  esac
DEFINE
   initial := (index=0);

MODULE Env(index)
-- Represents the evolution of available services, seen as a whole
VAR
  operation  : {start_op,ca,di,tr,am,it};
  target : Target(operation);     -- "produces" operations
  s1 : Service1(index,operation); -- "consumes" current index and operation
  s2 : Service2(index,operation);
  s3 : Service3(index,operation);
DEFINE
  initial := (s1.initial & s2.initial & s3.initial & target.initial & operation=start_op);
  failure := (s1.failure |s2.failure |s3.failure) |
             (target.final & !(s1.final & s2.final & s3.final));

-- Target service-----------
MODULE Target(op) --op is an output parameter
VAR
  state : {start_st,t1,t2,t3,t4,t5};
INIT
  state = start_st & op = start_op
TRANS
  case
    state = start_st & op = start_op : next(state) = t1 & next(op) in {ca};
    state = t1 & op = ca : next(state) = t2 & next(op) in {tr,di} ;
    state = t2 & op = tr : next(state) = t4 & next(op) in {am};
    state = t2 & op = di : next(state) = t3 & next(op) in {tr};
    state = t3 & op = tr : next(state) = t4 & next(op) in {am};
    state = t4 & op = am : next(state) = t5 & next(op) in {it};
    state = t5 & op = it : next(state) = t1 & next(op) in {ca};
  esac
DEFINE
  initial := state=start_st & op=start_op;
  final := state in {t1}; -- final state(s)
-- end of target service -----------


-- Available service #1 -----------
MODULE Service1(index,operation)
VAR
  state : {start_st,a1,a2};

INIT
  state=start_st

TRANS
  case
    state=start_st & operation=start_op  & index=0: next(state)=a1;
    (index != 1) : next(state) = state; -- if not selected, remain still
    (state=a1 & operation = am) : next(state) in {a1};
    (state=a1 & operation = it) : next(state) in {a1};
    (state=a1 & operation = di) : next(state) in {a2};
    (state=a2 & operation = am) : next(state) in {a1};
    (state=a2 & operation = it) : next(state) in {a2};
  esac
DEFINE
  initial := state=start_st & operation=start_op  & index = 0 ;
  failure := index = 1 & !((state = a1 & operation in {am,it,di}) | (state = a2 & operation in {it,am}));
  final := state in {a1,a2};
-- end of available service #1 -----------

-- Available service #2 -----------
MODULE Service2(index,operation)
VAR
  state : {start_st,b1,b2,b3,b4};

INIT
  state=start_st

TRANS
  case
    state=start_st & operation=start_op  & index=0: next(state)=b1;
    (index != 2) : next(state) = state; -- if not selected, remain still
    (state=b1 & operation = ca) : next(state) in {b2};
    (state=b2 & operation = di) : next(state) in {b1};
    (state=b2 & operation = tr) : next(state) in {b3,b1};
    (state=b3 & operation = it) : next(state) in {b1};
    (state=b3 & operation = ca) : next(state) in {b4};
    (state=b4 & operation = di) : next(state) in {b3};
  esac
DEFINE
  initial := state=start_st & operation=start_op  & index = 0 ;
  failure :=index = 2 & !((state = b1 & operation in {ca}) |
                       (state = b2 & operation in {tr,di}) |
                       (state = b3 & operation in {it,ca}) |
                       (state = b4 & operation in {di}));
  final := state in {b1,b2,b3,b4};
-- end of available service #2 -----------

-- Available service #3 -----------
MODULE Service3(index,operation)
VAR
  state : {start_st,c1,c2};

INIT
  state=start_st

TRANS
  case
    state=start_st & operation=start_op  & index=0: next(state)=c1;
    (index != 3) : next(state) = state; -- if not selected, remain still
    (state=c1 & operation = it) : next(state) in {c2};
    (state=c2 & operation = tr) : next(state) in {c1};
    (state=c2 & operation = ca) : next(state) in {c2};
  esac
DEFINE
  initial := state=start_st & operation=start_op  & index = 0 ;
  failure :=index = 3 & !((state = c1 & operation in {it}) | (state = c2 & operation in {ca,tr}));
  final := state in {c1,c2};
-- end of available service #3 -----------
```

A.2 TLV Output—The Controller Generator

```
Check Realizability

Specification is realizable

Check that a symbolic strategy is correct

Transition relation is complete

All winning states satisfy invariant

Automaton States

State 1
env.operation = start_op               env.target.state = start_st
env.s1.state = start_st                env.s2.state = start_st
env.s3.state = start_st                sys.index = 0

State 2
env.operation = ca env.target.state = t1                  env.s1.state = a1
env.s2.state = b1  env.s3.state = c1  sys.index = 2

State 3
env.operation = tr env.target.state = t2                  env.s1.state = a1
env.s2.state = b2  env.s3.state = c1  sys.index = 2

State 4
env.operation = di env.target.state = t2                  env.s1.state = a1
env.s2.state = b2  env.s3.state = c1  sys.index = 1

State 5
env.operation = tr env.target.state = t3                  env.s1.state = a2
env.s2.state = b2  env.s3.state = c1  sys.index = 2

State 6
env.operation = am env.target.state = t4                  env.s1.state = a2
env.s2.state = b3  env.s3.state = c1  sys.index = 1

State 7
env.operation = am env.target.state = t4                  env.s1.state = a2
env.s2.state = b1  env.s3.state = c1  sys.index = 1

State 8
env.operation = it env.target.state = t5                  env.s1.state = a1
env.s2.state = b1  env.s3.state = c1  sys.index = 1

State 9
env.operation = it env.target.state = t5                  env.s1.state = a1
env.s2.state = b1  env.s3.state = c1  sys.index = 3

State 10
env.operation = ca env.target.state = t1                  env.s1.state = a1
env.s2.state = b1  env.s3.state = c2  sys.index = 2

State 11
env.operation = ca env.target.state = t1                  env.s1.state = a1
env.s2.state = b1  env.s3.state = c2  sys.index = 3

State 12
env.operation = tr env.target.state = t2                  env.s1.state = a1
env.s2.state = b1  env.s3.state = c2  sys.index = 3

State 13
env.operation = di env.target.state = t2                  env.s1.state = a1
env.s2.state = b1  env.s3.state = c2  sys.index = 1
```

```
State 14
env.operation = tr env.target.state = t3                  env.s1.state = a2
env.s2.state = b1  env.s3.state = c2  sys.index = 3

State 15
env.operation = am env.target.state = t4                  env.s1.state = a1
env.s2.state = b1  env.s3.state = c1  sys.index = 1

State 16
env.operation = tr env.target.state = t2                  env.s1.state = a1
env.s2.state = b2  env.s3.state = c2  sys.index = 2

State 17
env.operation = tr env.target.state = t2                  env.s1.state = a1
env.s2.state = b2  env.s3.state = c2  sys.index = 3

State 18
env.operation = di env.target.state = t2                  env.s1.state = a1
env.s2.state = b2  env.s3.state = c2  sys.index = 2

State 19
env.operation = di env.target.state = t2                  env.s1.state = a1
env.s2.state = b2  env.s3.state = c2  sys.index = 1

State 20
env.operation = tr env.target.state = t3                  env.s1.state = a2
env.s2.state = b2  env.s3.state = c2  sys.index = 2

State 21
env.operation = tr env.target.state = t3                  env.s1.state = a2
env.s2.state = b2  env.s3.state = c2  sys.index = 3

State 22
env.operation = am env.target.state = t4                  env.s1.state = a2
env.s2.state = b2  env.s3.state = c1  sys.index = 1

State 23
env.operation = it env.target.state = t5                  env.s1.state = a1
env.s2.state = b2  env.s3.state = c1  sys.index = 3

State 24
env.operation = ca env.target.state = t1                  env.s1.state = a1
env.s2.state = b2  env.s3.state = c2  sys.index = 3

State 25
env.operation = am env.target.state = t4                  env.s1.state = a2
env.s2.state = b3  env.s3.state = c2  sys.index = 1

State 26
env.operation = am env.target.state = t4                  env.s1.state = a2
env.s2.state = b1  env.s3.state = c2  sys.index = 1

State 27
env.operation = it env.target.state = t5                  env.s1.state = a1
env.s2.state = b1  env.s3.state = c2  sys.index = 1

State 28
env.operation = it env.target.state = t5                  env.s1.state = a1
env.s2.state = b3  env.s3.state = c2  sys.index = 2

State 29
env.operation = it env.target.state = t5                  env.s1.state = a1
env.s2.state = b3  env.s3.state = c2  sys.index = 1

State 30
env.operation = ca env.target.state = t1                  env.s1.state = a1
env.s2.state = b3  env.s3.state = c2  sys.index = 2

State 31
env.operation = ca env.target.state = t1                  env.s1.state = a1
env.s2.state = b3  env.s3.state = c2  sys.index = 3

State 32
env.operation = tr env.target.state = t2                  env.s1.state = a1
env.s2.state = b3  env.s3.state = c2  sys.index = 3

State 33
env.operation = di env.target.state = t2                  env.s1.state = a1
env.s2.state = b3  env.s3.state = c2  sys.index = 1

State 34
env.operation = tr env.target.state = t3                  env.s1.state = a2
env.s2.state = b3  env.s3.state = c2  sys.index = 3

State 35
env.operation = am env.target.state = t4                  env.s1.state = a1
env.s2.state = b3  env.s3.state = c1  sys.index = 1

State 36
env.operation = it env.target.state = t5                  env.s1.state = a1
env.s2.state = b3  env.s3.state = c1  sys.index = 2

State 37
env.operation = it env.target.state = t5                  env.s1.state = a1
env.s2.state = b3  env.s3.state = c1  sys.index = 3

State 38
env.operation = tr env.target.state = t2                  env.s1.state = a1
env.s2.state = b4  env.s3.state = c2  sys.index = 3

State 39
env.operation = di env.target.state = t2                  env.s1.state = a1
env.s2.state = b4  env.s3.state = c2  sys.index = 2

State 40
env.operation = di env.target.state = t2                  env.s1.state = a1
env.s2.state = b4  env.s3.state = c2  sys.index = 1

State 41
env.operation = tr env.target.state = t3                  env.s1.state = a2
env.s2.state = b4  env.s3.state = c2  sys.index = 3

State 42
env.operation = am env.target.state = t4                  env.s1.state = a2
env.s2.state = b4  env.s3.state = c1  sys.index = 1

State 43
env.operation = it env.target.state = t5                  env.s1.state = a1
env.s2.state = b4  env.s3.state = c1  sys.index = 3

State 44
env.operation = ca env.target.state = t1                  env.s1.state = a1
env.s2.state = b4  env.s3.state = c2  sys.index = 3

State 45
env.operation = tr env.target.state = t3                  env.s1.state = a1
env.s2.state = b3  env.s3.state = c2  sys.index = 3

State 46
env.operation = am env.target.state = t4                  env.s1.state = a1
env.s2.state = b4  env.s3.state = c1  sys.index = 1

State 47
```

```
env.operation = tr env.target.state = t3                  env.s1.state = a1
env.s2.state = b1  env.s3.state = c2  sys.index = 3

State 48
env.operation = am env.target.state = t4                  env.s1.state = a1
env.s2.state = b2  env.s3.state = c1  sys.index = 1

State 49
env.operation = am env.target.state = t4                  env.s1.state = a1
env.s2.state = b3  env.s3.state = c2  sys.index = 1

State 50
env.operation = am env.target.state = t4                  env.s1.state = a1
env.s2.state = b1  env.s3.state = c2  sys.index = 1


Automaton Transitions

From 1 to  2
From 2 to  3 4
From 3 to  15 35
From 4 to  5
From 5 to  6 7
From 6 to  36 37
From 7 to  8 9
From 8 to  2
From 9 to  10 11
From 10 to  16 17 18 19
From 11 to  12 13
From 12 to  15
From 13 to  14
From 14 to  7
From 15 to  8 9
From 16 to  49 50
From 17 to  48
From 18 to  47
From 19 to  20 21
From 20 to  25 26
From 21 to  22
From 22 to  23
From 23 to  24
From 24 to  16 17 18 19
From 25 to  28 29
From 26 to  27
From 27 to  10 11
From 28 to  10 11
From 29 to  30 31
From 30 to  38 39 40
From 31 to  32 33
From 32 to  35
From 33 to  34
From 34 to  6
From 35 to  36 37
From 36 to  2
From 37 to  30 31
From 38 to  46
From 39 to  45
From 40 to  41
From 41 to  42
From 42 to  43
From 43 to  44
From 44 to  38 39 40
From 45 to  35
From 46 to  43
From 47 to  15
From 48 to  23
From 49 to  28 29
From 50 to  27

Automaton has 50 states, and 79 transitions
```

A.3 Transitions.json—The Controller Generator in the JSON Format

```
[
  {
    "targetState": "T1",
    "behaviorAState": "A1",
    "behaviorBState": "B1",
    "behaviorCState": "C1",
    "action": "CarInfo",
    "behavior": "B"
  },
  {
    "targetState": "T5",
    "behaviorAState": "A1",
    "behaviorBState": "B1",
    "behaviorCState": "C1",
    "action": "TicketIssue",
    "behavior": "C"
  },
  {
    "targetState": "T5",
    "behaviorAState": "A1",
    "behaviorBState": "B1",
    "behaviorCState": "C1",
    "action": "TicketIssue",
    "behavior": "A"
  },
  {
    "targetState": "T4",
    "behaviorAState": "A1",
    "behaviorBState": "B1",
    "behaviorCState": "C1",
    "action": "Ambulance",
    "behavior": "A"
  },
  {
    "targetState": "T2",
    "behaviorAState": "A1",
    "behaviorBState": "B2",
    "behaviorCState": "C1",
    "action": "DriverInfo",
    "behavior": "A"
  },
  {
    "targetState": "T4",
    "behaviorAState": "A1",
    "behaviorBState": "B2",
    "behaviorCState": "C1",
    "action": "Ambulance",
    "behavior": "A"
  },
  {
    "targetState": "T2",
    "behaviorAState": "A1",
    "behaviorBState": "B2",
    "behaviorCState": "C1",
    "action": "Trucks",
    "behavior": "B"
  },
  {
    "targetState": "T5",
    "behaviorAState": "A1",
    "behaviorBState": "B2",
    "behaviorCState": "C1",
    "action": "TicketIssue",
    "behavior": "C"
  },
  {
    "targetState": "T4",
    "behaviorAState": "A2",
    "behaviorBState": "B2",
    "behaviorCState": "C1",
    "action": "Ambulance",
    "behavior": "A"
  },
  {
    "targetState": "T3",
    "behaviorAState": "A2",
    "behaviorBState": "B2",
    "behaviorCState": "C1",
    "action": "Trucks",
    "behavior": "B"
  },
  {
    "targetState": "T4",
    "behaviorAState": "A2",
    "behaviorBState": "B1",
    "behaviorCState": "C1",
    "action": "Ambulance",
    "behavior": "A"
  },
  {
    "targetState": "T1",
    "behaviorAState": "A1",
    "behaviorBState": "B1",
    "behaviorCState": "C2",
    "action": "CarInfo",
    "behavior": "B"
  },
  {
    "targetState": "T2",
    "behaviorAState": "A1",
    "behaviorBState": "B1",
    "behaviorCState": "C2",
    "action": "DriverInfo",
    "behavior": "A"
  },
  {
    "targetState": "T4",
    "behaviorAState": "A1",
    "behaviorBState": "B1",
    "behaviorCState": "C2",
    "action": "Ambulance",
    "behavior": "A"
  },
  {
    "targetState": "T1",
    "behaviorAState": "A1",
    "behaviorBState": "B1",
    "behaviorCState": "C2",
    "action": "CarInfo",
    "behavior": "C"
  },
  {
    "targetState": "T5",
    "behaviorAState": "A1",
    "behaviorBState": "B1",
    "behaviorCState": "C2",
    "action": "TicketIssue",
    "behavior": "A"
  },
{
  "targetState": "T2",
  "behaviorAState": "A1",
  "behaviorBState": "B1",
  "behaviorCState": "C2",
  "action": "Trucks",
  "behavior": "C"
},
{
  "targetState": "T3",
  "behaviorAState": "A1",
  "behaviorBState": "B1",
  "behaviorCState": "C2",
  "action": "Trucks",
  "behavior": "C"
},
{
  "targetState": "T2",
  "behaviorAState": "A1",
  "behaviorBState": "B2",
  "behaviorCState": "C2",
  "action": "Trucks",
  "behavior": "C"
},
{
  "targetState": "T2",
  "behaviorAState": "A1",
  "behaviorBState": "B2",
  "behaviorCState": "C2",
  "action": "Trucks",
  "behavior": "B"
},
{
  "targetState": "T1",
  "behaviorAState": "A1",
  "behaviorBState": "B2",
  "behaviorCState": "C2",
  "action": "CarInfo",
  "behavior": "C"
},
{
  "targetState": "T2",
  "behaviorAState": "A1",
  "behaviorBState": "B2",
  "behaviorCState": "C2",
  "action": "DriverInfo",
  "behavior": "A"
},
{
  "targetState": "T2",
  "behaviorAState": "A1",
  "behaviorBState": "B2",
  "behaviorCState": "C2",
  "action": "DriverInfo",
  "behavior": "B"
},
{
  "targetState": "T3",
  "behaviorAState": "A2",
  "behaviorBState": "B2",
  "behaviorCState": "C2",
  "action": "Trucks",
  "behavior": "B"
},
{
  "targetState": "T3",
  "behaviorAState": "A2",
    "behaviorBState": "B2",
    "behaviorCState": "C2",
    "action": "Trucks",
    "behavior": "C"
  },
  {
    "targetState": "T4",
    "behaviorAState": "A2",
    "behaviorBState": "B1",
    "behaviorCState": "C2",
    "action": "Ambulance",
    "behavior": "A"
  },
  {
    "targetState": "T3",
    "behaviorAState": "A2",
    "behaviorBState": "B1",
    "behaviorCState": "C2",
    "action": "Trucks",
    "behavior": "C"
  },
  {
    "targetState": "T4",
    "behaviorAState": "A1",
    "behaviorBState": "B3",
    "behaviorCState": "C1",
    "action": "Ambulance",
    "behavior": "A"
  },
  {
    "targetState": "T5",
    "behaviorAState": "A1",
    "behaviorBState": "B3",
    "behaviorCState": "C1",
    "action": "TicketIssue",
    "behavior": "B"
  },
  {
    "targetState": "T5",
    "behaviorAState": "A1",
    "behaviorBState": "B3",
    "behaviorCState": "C1",
    "action": "TicketIssue",
    "behavior": "C"
  },
  {
    "targetState": "T2",
    "behaviorAState": "A1",
    "behaviorBState": "B3",
    "behaviorCState": "C2",
    "action": "Trucks",
    "behavior": "C"
  },
  {
    "targetState": "T3",
    "behaviorAState": "A1",
    "behaviorBState": "B3",
    "behaviorCState": "C2",
    "action": "Trucks",
    "behavior": "C"
  },
  {
    "targetState": "T1",
    "behaviorAState": "A1",
    "behaviorBState": "B3",
    "behaviorCState": "C2",
    "action": "CarInfo",
    "behavior": "C"
  },
  {
    "targetState": "T1",
    "behaviorAState": "A1",
    "behaviorBState": "B3",
    "behaviorCState": "C2",
    "action": "CarInfo",
    "behavior": "B"
  },
  {
    "targetState": "T5",
    "behaviorAState": "A1",
    "behaviorBState": "B3",
    "behaviorCState": "C2",
    "action": "TicketIssue",
    "behavior": "A"
  },
  {
    "targetState": "T5",
    "behaviorAState": "A1",
    "behaviorBState": "B3",
    "behaviorCState": "C2",
    "action": "TicketIssue",
    "behavior": "B"
  },
  {
    "targetState": "T4",
    "behaviorAState": "A1",
    "behaviorBState": "B3",
    "behaviorCState": "C2",
    "action": "Ambulance",
    "behavior": "A"
  },
  {
    "targetState": "T2",
    "behaviorAState": "A1",
    "behaviorBState": "B3",
    "behaviorCState": "C2",
    "action": "DriverInfo",
    "behavior": "A"
  },
  {
    "targetState": "T4",
    "behaviorAState": "A2",
    "behaviorBState": "B3",
    "behaviorCState": "C2",
    "action": "Ambulance",
    "behavior": "A"
  },
  {
    "targetState": "T3",
    "behaviorAState": "A2",
    "behaviorBState": "B3",
    "behaviorCState": "C2",
    "action": "Trucks",
    "behavior": "C"
  },
  {
    "targetState": "T4",
    "behaviorAState": "A2",
    "behaviorBState": "B3",
    "behaviorCState": "C1",
    "action": "Ambulance",
    "behavior": "A"
  },
  {
    "targetState": "T1",
    "behaviorAState": "A1",
    "behaviorBState": "B4",
    "behaviorCState": "C2",
    "action": "CarInfo",
    "behavior": "C"
  },
  {
    "targetState": "T2",
    "behaviorAState": "A1",
    "behaviorBState": "B4",
    "behaviorCState": "C2",
    "action": "Trucks",
    "behavior": "C"
  },
  {
    "targetState": "T2",
    "behaviorAState": "A1",
    "behaviorBState": "B4",
    "behaviorCState": "C2",
    "action": "DriverInfo",
    "behavior": "A"
  },
  {
    "targetState": "T2",
    "behaviorAState": "A1",
    "behaviorBState": "B4",
    "behaviorCState": "C2",
    "action": "DriverInfo",
    "behavior": "B"
  },
  {
    "targetState": "T5",
    "behaviorAState": "A1",
    "behaviorBState": "B4",
    "behaviorCState": "C1",
    "action": "TicketIssue",
    "behavior": "C"
  },
  {
    "targetState": "T4",
    "behaviorAState": "A1",
    "behaviorBState": "B4",
    "behaviorCState": "C1",
    "action": "Ambulance",
    "behavior": "A"
  },
  {
    "targetState": "T3",
    "behaviorAState": "A2",
    "behaviorBState": "B4",
    "behaviorCState": "C2",
    "action": "Trucks",
    "behavior": "C"
  },
  {
    "targetState": "T4",
    "behaviorAState": "A2",
    "behaviorBState": "B4",
    "behaviorCState": "C1",
    "action": "Ambulance",
    "behavior": "A"
  }
]
```

Appendix B

This appendix provides the Common.scala file, which containts a list of immutable objects.

B.1 Constant Names

```
  val HANDLER = "handlerActor"
  val BEHAVIOR_A = "behaviorA"
  val BEHAVIOR_B = "behaviorB"
  val BEHAVIOR_C = "behaviorC"
  val DISPATCHER = "integration-dispatcher"
  val TARGET = "target"
  val DISPATCHER_ROLE = "dispatcher"
  val CONTROLLER_GENERATOR = "controllerGenerator"
  val CAR_INFO = "car information"
  val DRIVER_INFO = "driver information"
  val TRUCKS = "trucks"
  val AMBULANCE = "ambulance"
  val TICKET_ISSUE = "issue the ticket"
```

B.2 Constant Actions

```
  sealed trait Action
  case object CarInfo extends Action
  case object DriverInfo extends Action
  case object Trucks extends Action
  case object Ambulance extends Action
  case object TicketIssue extends Action
  case object Start extends Action
  case object NoAction extends Action
```

B.3 Target and Behaviors States

```
sealed trait State {
    val property: Property = NoProp
  }
  case object A1 extends State {
    override val property = Compensatable
  }
  case object A2 extends State {
    override val property = Compensatable
  }
  case object B1 extends State {
    override val property = Compensatable
  }
  case object B2 extends State {
    override val property = Compensatable
  }
  case object B3 extends State {
    override val property = Compensatable
  }
```

```
case object B4 extends State {
  override val property = Compensatable
}
case object C1 extends State {
  override val property = Compensatable
}
case object C2 extends State {
  override val property = Compensatable
}
case object Freeze extends State
case object NoState extends State

case object T1 extends State
case object T2 extends State
case object T3 extends State
case object T4 extends State
case object T5 extends State
```

B.4 Constant Behavior Names

```
sealed trait Behavior
case object A extends Behavior
case object B extends Behavior
case object C extends Behavior
case object NoBehavior extends Behavior
```

B.5 Constant Message Objects and Classes

```
object Constants {

  case object GetResults
  case object Failure
  case object OK
  case object RestartTarget
  case object Ack
  case object AskForAvailableBehaviors
  case object CompensatedSuccessfully
  case object ActorCompensated
  case object RequestForHandler
  case object ResumeException extends Exception
  case object StopException extends Exception
  case object RestartException extends Exception

  case class BehaviorRegistration(behavior: String, currentState: State, initialState: State)
  case class Recompute(terminatedBehaviorName : String, isRegistered: Boolean = true)
  case class KillBehavior(behavior : ActorRef)
  case class FreezeBehavior(behavior : ActorRef)
  case class UnFreezeBehavior(behavior : ActorRef, state: State)
  case class GotoException(behavior : ActorRef, exception: Exception = new RuntimeException)
  case class TargetRequest(targetState: State, action: Action, restart: Boolean = false)
  case class Request(targetState: State, action: Action, restart: Boolean = false)
  case class InitializeRequest(targetState: State, action: Action, restart: Boolean = false
                                                                 , target: ActorRef)
  case class BehaviorTransition(actor:ActorRef, oldState: State, newState: State)
  case class BehaviorTerminated(actor:ActorRef)
  case class Handler(handler: ActorRef)
  case class GenerateHandler(behAEntry: Optional[Entry[Option[ActorRef], BehaviorStatus]]
       , behBEntry: Optional[Entry[Option[ActorRef], BehaviorStatus]]
       , behCEntry: Optional[Entry[Option[ActorRef], BehaviorStatus]], sender: ActorRef, utiltiyActor: ActorRef)
  case class SelectedControllers(list: List[Node], action: Action, sender: ActorRef)
  case class Choose(targetState: State, action: Action, stateHolder: ConcurrentHashMap[ActorRef, BehaviorStatus]
                                                      , sender: ActorRef)
  case class Compensate(recoveryObject: RecoveryObject)
  case class ScheduleCompensation(stack: mutable.Stack[RecoveryObject])
  case class AskRequest(action: Action)
  case class Result(res: String)
  case class TargetResult()
  case class UnFreeze(state: State)

}
```

Appendix C

C.1 The **`receive`** Method of an Orchestrator

```
def receive = {

    case ActorCompensated =>
      targetHandlerMap.entrySet().stream()
                              .filter(entry => entry.getValue == sender())
                              .findAny().ifPresent(entry => {
        val target = entry.getKey
        changeBehaviorAvailability(behaviorARoutees, entry.getValue, true)
        changeBehaviorAvailability(behaviorBRoutees, entry.getValue, true)
        changeBehaviorAvailability(behaviorCRoutees, entry.getValue, true)
        targetHandlerMap.remove(target)
      })

    case DeadLetter(msg, from, to) =>
      log.debug(DeadLetterMsg.msg, from, to, msg)

    case Terminated(ref) => {
      ref.path.name match {
        case name if name.startsWith(UniqueNames.BEHAVIOR_A) =>
          behaviorARoutees.entrySet().stream()
            .filter(entry => entry.getKey == ref)
            .findAny()
            .ifPresent(entry => if (entry.getValue.handlerActor.isDefined)
                              entry.getValue.handlerActor.get ! BehaviorTerminated(ref))
          behaviorARoutees.remove(ref)
        case name if name.startsWith(UniqueNames.BEHAVIOR_B) =>
          behaviorBRoutees.entrySet().stream()
            .filter(entry => entry.getKey == ref)
            .findAny()
            .ifPresent(entry => if (entry.getValue.handlerActor.isDefined)
                              entry.getValue.handlerActor.get ! BehaviorTerminated(ref))
          behaviorBRoutees.remove(ref)
        case name if name.startsWith(UniqueNames.BEHAVIOR_C) =>
          behaviorCRoutees.entrySet().stream()
            .filter(entry => entry.getKey == ref)
            .findAny()
            .ifPresent(entry => if (entry.getValue.handlerActor.isDefined)
                              entry.getValue.handlerActor.get ! BehaviorTerminated(ref))
          behaviorCRoutees.remove(ref)
      }
    }

    case BehaviorRegistration(behavior, currentState, initialState) =>
      behavior match {
        case UniqueNames.BEHAVIOR_A =>
          behaviorARoutees.put(sender(), BehaviorStatus(initialState, currentState))
          log.info(Registration.msg, sender().path.name, self.path.name)
          applyWatchAndSubscribe(sender())
        case UniqueNames.BEHAVIOR_B =>
          behaviorBRoutees.put(sender(), BehaviorStatus(initialState, currentState))
          log.info(Registration.msg, sender().path.name, self.path.name)
          applyWatchAndSubscribe(sender())
        case UniqueNames.BEHAVIOR_C =>
          behaviorCRoutees.put(sender(), BehaviorStatus(initialState, currentState))
          log.info(Registration.msg, sender().path.name, self.path.name)
          applyWatchAndSubscribe(sender())
      }

    case GotoException(behavior, exception) =>
      behavior ! exception

    case FreezeBehavior(behavior) =>
        behavior ! Freeze

    case UnFreezeBehavior(behavior, state) =>
      behavior ! UnFreeze(state)

    case AskForAvailableBehaviors =>
      val listBuffer = new ListBuffer[ActorRef]
      behaviorARoutees.entrySet().stream().forEach(elem => listBuffer += elem.getKey)
      behaviorBRoutees.entrySet().stream().forEach(elem => listBuffer += elem.getKey)
      behaviorCRoutees.entrySet().stream().forEach(elem => listBuffer += elem.getKey)
      sender() ! listBuffer

    case KillBehavior(behavior) =>
      behavior ! PoisonPill
```

```
case ScheduleCompensation(stack) =>
  targetHandlerMap.entrySet()
    .stream()
    .filter(entry => entry.getValue == sender())
    .findAny()
    .ifPresent(entry => {
    val target = entry.getKey
    targetHandlerMap.remove(target)
  })
  lazy val cancellable:Cancellable = system
                .scheduler
                .schedule(10 seconds, 10 seconds){
    compensate(stack, cancellable, sender())
  }
  cancellable

case TargetRequest(targetState: State, action: Action, restart) => {
  val entry = targetHandlerMap.entrySet().stream()
                                        .filter(entry => entry.getKey == sender())
                                        .findAny()
  if (entry.isPresent) {
    log.info(ForwardMessage.msg, entry.get().getValue.path.name)
    val actor = entry.get().getValue
    actor forward Request(targetState, action, restart)
  } else {
    become(ReceiveHandlerActorMessage)
    val behaviorAEntry = behaviorARoutees.entrySet().stream()
      .filter(elem => elem.getValue.currentState == elem.getValue.initialState)
      .filter(elem => elem.getValue.isAvailable).findAny()
    behaviorAEntry.ifPresent(e => e.getValue.isAvailable = false)

    val behaviorBEntry = behaviorBRoutees.entrySet().stream()
      .filter(elem => elem.getValue.currentState == elem.getValue.initialState)
      .filter(elem => elem.getValue.isAvailable).findAny()
    behaviorBEntry.ifPresent(e => e.getValue.isAvailable = false)

    val behaviorCEntry = behaviorCRoutees.entrySet().stream()
      .filter(elem => elem.getValue.currentState == elem.getValue.initialState)
      .filter(elem => elem.getValue.isAvailable).findAny()
    behaviorCEntry.ifPresent(e => e.getValue.isAvailable = false)

    val handlerActor = system.actorOf(HandlerActor.props(if (behaviorAEntry.isPresent)
                                                          Some(behaviorAEntry.get())
                                                         else None
      , if (behaviorBEntry.isPresent) Some(behaviorBEntry.get()) else None
      , if (behaviorCEntry.isPresent) Some(behaviorCEntry.get()) else None, self)
      , sender().path.name + UniqueNames.HANDLER)
    log.info(HandlerActorCreatedMessage.msg, handlerActor.path.name)
    context.watch(handlerActor)
    targetHandlerMap.put(sender(), handlerActor)
    behaviorAEntry.ifPresent(e => e.getValue.handlerActor = Some(handlerActor))
    behaviorBEntry.ifPresent(e => e.getValue.handlerActor = Some(handlerActor))
    behaviorCEntry.ifPresent(e => e.getValue.handlerActor = Some(handlerActor))
    handlerActor ! InitializeRequest(targetState, action, restart, sender())
  }
}

case CurrentState(actor:ActorRef, state: State) =>
  log.info(CurrentStateLog.msg, actor.path.name, state)
  actor.path.name match {
    case name if name.startsWith(UniqueNames.BEHAVIOR_A) =>
      behaviorARoutees.entrySet().stream()
                                .filter(entry => entry.getKey == actor)
                                .findAny()
                                .ifPresent(entry => entry.getValue.currentState = state)
    case name if name.startsWith(UniqueNames.BEHAVIOR_B) =>
      behaviorBRoutees.entrySet().stream()
                                .filter(entry => entry.getKey == actor)
                                .findAny()
                                .ifPresent(entry => entry.getValue.currentState = state)
    case name if name.startsWith(UniqueNames.BEHAVIOR_C) =>
      behaviorCRoutees.entrySet().stream()
                                .filter(entry => entry.getKey == actor)
                                .findAny()
                                .ifPresent(entry => entry.getValue.currentState = state)
  }

case Transition(actor:ActorRef, oldState: State, newState: State) =>
  become(ReceiveSuccessTransition)
  actor.path.name match {
    case name if name.startsWith(UniqueNames.BEHAVIOR_A) =>
      behaviorARoutees.entrySet().stream()
                                .filter(entry => entry.getKey == actor)
                                .findAny()
                                .ifPresent(entry => {
                                  if (newState != Freeze) {
                                    entry.getValue.currentState = newState
                                  entry.getValue.isUp = true
                                  } else if (newState == Freeze) {
                                        entry.getValue.isUp = false
                                      }
          if (entry.getValue.handlerActor.isDefined)
           entry.getValue.handlerActor.get ! BehaviorTransition(actor, oldState, newState)
        })
      case name if name.startsWith(UniqueNames.BEHAVIOR_B) =>
        behaviorBRoutees.entrySet().stream()
                                   .filter(entry => entry.getKey == actor)
                                   .findAny().ifPresent(entry => {
          if (newState != Freeze) {
            entry.getValue.currentState = newState
            entry.getValue.isUp = true
          } else if (newState == Freeze) {
            entry.getValue.isUp = false
          }
          if (entry.getValue.handlerActor.isDefined)
           entry.getValue.handlerActor.get ! BehaviorTransition(actor, oldState, newState)
        })
      case name if name.startsWith(UniqueNames.BEHAVIOR_C) =>
        behaviorCRoutees.entrySet().stream()
                                   .filter(entry => entry.getKey == actor)
                                   .findAny().ifPresent(entry => {
          if (newState != Freeze) {
            entry.getValue.currentState = newState
            entry.getValue.isUp = true
          } else if (newState == Freeze) {
            entry.getValue.isUp = false
          }
          if (entry.getValue.handlerActor.isDefined)
           entry.getValue.handlerActor.get ! BehaviorTransition(actor, oldState, newState)
        })
    }

  case event =>
    stash()
    log.debug(UnhandledEvent.msg, event)
}
```

Appendix D

D.1 Log Messages

```
sealed trait LogMessage {
    val msg: String
  }
  sealed trait TargetLogMessage {
    def content() : String
    val symbolStart: String = " -- START TARGET LOG -------------------------------------------"
    val symbolEnd: String = " -- END OF TARGET LOG -------------------------------------------"
    val msg: String = s"\n $symbolStart \n $content \n $symbolEnd"
  }
  sealed trait DispatcherLogMessage {
    def content() : String
    val symbolStart: String = " --------------- START DISPATCHER LOG --------------------------"
    val symbolEnd: String = " --------------- END OF DISPATCHER LOG --------------------------"
    val msg: String = s"\n $symbolStart \n $content \n $symbolEnd"
  }
  sealed trait BehaviorLogMessage {
    def content() : String
    val symbolStart: String = " ---------------------------- START BEHAVIOR LOG ---------------"
    val symbolEnd: String = " ---------------------------- END OF BEHAVIOR LOG ---------------"
    val msg: String = s"\n $symbolStart \n $content \n $symbolEnd"
  }
  sealed trait HandlerActorLogMessage {
    def content() : String
    val symbolStart: String = " ---------------------------------------- START HANDLER LOG ----"
    val symbolEnd: String = " ---------------------------------------- END OF HANDLER LOG ----"
    val msg: String = s"\n $symbolStart \n $content \n $symbolEnd"
  }
  sealed trait DebugLogMessage {
    def content() : String
    val symbolStart: String = " -- START DEBUG LOG --------------------------------------------"
    val symbolEnd: String = " -- END OF DEBUG LOG --------------------------------------------"
    val msg: String = s"\n $symbolStart \n $content \n $symbolEnd"
  }

  sealed trait Behavior
  case object A extends Behavior
  case object B extends Behavior
  case object C extends Behavior
  case object NoBehavior extends Behavior

  trait Registration {
    def getBehaviorName(): String
    def getInitialState(): Constants.State
  }
  trait BehaviorInitialization {
    def init(name: String, port: Int, logLevel: String = "INFO")
  }


  case object UnhandledEvent extends DebugLogMessage {
    override def content: String = "Unhandled event received with content: {}."
  }
  case object DeadLetterMsg extends DebugLogMessage {
    override def content: String = "Message from {} to {} does not deliver and the message is {}"
  }

  // Behavior Logging
  case object CompensatedBehavior extends BehaviorLogMessage {
    override def content: String  = "{} in {} Compensated."
  }
  case object Transition extends BehaviorLogMessage {
    override def content: String = "{} is going from {} to {}."
  }
  case object TerminatedMsg extends BehaviorLogMessage {
    override def content: String = "{} is terminated and controller generator is going to adjust."
  }
  case object SendRegistrationRequest extends BehaviorLogMessage {
    override def content: String = "{} is going to send a registration request."
  }
  case object FSMUnhandledEvent extends BehaviorLogMessage {
    override def content: String = "Unhandled Event \'{}\' received in state {} in {} FSMActor."
  }
  case object FreezeMsg extends BehaviorLogMessage {
    override def content: String = "{} is going to freeze."
  }
  case object UnFreezeMsg extends BehaviorLogMessage {
    override def content: String = "{} is going to Unfreeze."
  }
  case object ExceptionMsg extends BehaviorLogMessage {
    override def content: String = "{} thrown an exception \'{}\' in state {}."
  }
  // End of Behavior Logging

  // Behavior B Logging
  case object B1CarInfoB2  extends LogMessage {
    override val msg: String = "Car information request has been done, state moving from B1 to B2"
  }
  case object B2DriverInfoB1 extends LogMessage {
    override val msg: String = "Driver information request has been done, state moving from B2 to B1"
  }
  case object B2TrucksB3 extends LogMessage {
    override val msg: String = "Trucks request has been done, state moving from B2 to B3"
  }
  case object B2TrucksB1 extends LogMessage {
    override val msg: String = "Trucks request has been done, state moving from B2 to B1"
  }
  case object B3CarInfoB4 extends LogMessage {
    override val msg: String = "Car information request has been done, state moving from B3 to B4"
  }
  case object B3IssueTicketB1 extends LogMessage {
    override val msg: String = "Ticket issue request has been done, state moving from B3 to B1"
  }
  case object B4DriverInfoB3 extends LogMessage {
    override val msg: String = "Driver information request has been done, state moving from B4 to B3"
  }
  // End of Behavior B Logging


  // Behavior A Logging
  case object A1DriverInfoA2 extends LogMessage {
    override val msg: String = "Driver information request has been done, state moving from A1 to A2"
  }
  case object A1AmbulanceA1 extends LogMessage {
    override val msg: String = "Ambulance request has been done, state moving from A1 to A1"
  }
  case object A1IssueTicketA1 extends LogMessage {
    override val msg: String = "Ticket issue request has been done, state moving from A1 to A1"
  }
  case object A2AmbulanceA1 extends LogMessage {
    override val msg: String = "Ambulance request has been done, state moving from A2 to A1"
  }
  case object A2IssueTicketA2 extends LogMessage {
    override val msg: String = "Ticket issue request has been done, state moving from A2 to A2"
  }
  // End of Behavior A Logging


  // Behavior C Logging
  case object C1IssueTicketC2 extends LogMessage {
    override val msg: String = "Ticket issue request has been done, state moving from C1 to C2"
  }
  case object C2TrucksC1 extends LogMessage {
    override val msg: String = "Trucks request has been done, state moving from C2 to C1"
  }
  case object C2CarInfoC2 extends LogMessage {
    override val msg: String = "Car information request has been done, state moving from C2 to C2"
  }
  // End of Behavior C Logging

  // Handler Logging
  case object BehaviorResult extends HandlerActorLogMessage {
    override def content: String = "{} and result is sending to {}"
  }
  case object SuccessfullyCompensated extends HandlerActorLogMessage {
    override def content: String = "{} is Successfully Compensated."
  }
  case object UnableToCompensateMsg extends HandlerActorLogMessage {
    override def content: String = "{} is unable to compensate please contact us to do it manually."
  }
  case object NotCompensatableMsg extends HandlerActorLogMessage {
    override def content: String = "{} with {} property is not compensatable."
  }
  case object CompensationOnScheduleMsg extends HandlerActorLogMessage {
    override def content: String = "{} is not available and compensation message is scheduled to send later."
  }
  case object CurrentStates extends HandlerActorLogMessage {
    override def content: String = "Current states are {}, {}, {}."
  }
  case object EmptyCandidateControllers extends HandlerActorLogMessage {
    override def content: String = "There are no Candidate Controllers for {} and the recovery stack is {}"
  }
  case object CandidateControllers extends HandlerActorLogMessage {
    override def content: String = "Candidate Controllers for {} are {}"
  }
  case object SelectedBehavior extends HandlerActorLogMessage {
    override def content: String = "{} is selected for {}"
  }
```

```
case object CompensateMsg extends HandlerActorLogMessage {
  override def content: String = "{} is going to compensate."
}
// End of Handler Logging


// Dispatcher Logging
case object Registration extends DispatcherLogMessage {
  override def content: String = "{} is Registered successfully in {}"
}
case object CompensationMsg extends DispatcherLogMessage {
  override def content: String = "{} compensation message from {} to {} is going to send to {}."
}
case object RetryScheduledCompensation extends DispatcherLogMessage {
  override def content: String = "{} Not found, retry later."
}
case object SucceedCompensationMsg extends DispatcherLogMessage {
  override def content: String = "{} successfully compensated."
}
case object ForwardMessage extends DispatcherLogMessage {
  override def content: String = "{} has been chosen to forward the request...."
}
case object HandlerActorCreatedMessage extends DispatcherLogMessage {
  override def content: String = "{} is created to handle the request...."
}
case object CurrentStateLog extends DispatcherLogMessage {
  override def content: String = "{} is in state {}"
}
case object HandleStashMessage extends DispatcherLogMessage {
  override def content: String = "Received {} and stash it while waiting for a message from actor in {}."
}
case object HandlerActorMessage extends DispatcherLogMessage {
  override def content: String = "Handler Actor created successfully."
}
// End of Dispatcher Logging

// Logging
case object PreStartMessage extends LogMessage {
  override val msg: String = "{} preStart Hook...."
}
case object PostRestart extends LogMessage {
  override val msg: String = "{} postRestart Hook...."
}
case object PreRestart extends LogMessage {
  override val msg: String = "{} preRestart Hook.... due to {}"
}
case object PostStop extends LogMessage {
  override val msg: String = "{} postStop Hook...."
}

// Target Logging
case object TargetRequest extends TargetLogMessage  {
  override def content: String = "{} is sending a request for {}."
}
case object TargetRegret extends TargetLogMessage {
  override def content: String = "Unfortunately the request could not realized by behaviors, try again later..."
}
// End of Target Logging
```

Appendix E

E.1 Log Messages for the Normal Execution

```
[info]   -- START TARGET LOG -------------------------------------------
[info]  T1 is sending a request for car information.
[info]   -- END OF TARGET LOG -------------------------------------------

[info]   --------------- START DISPATCHER LOG --------------------------
[info]  targethandlerActor is created to handle the request....
[info]   --------------- END OF DISPATCHER LOG --------------------------

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  Current states are A1, B1, C1.
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  Candidate Controllers for CarInfo are List(Node(T1,A1,B1,C1,CarInfo,B,true,Compensatable))
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  B is selected for CarInfo
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   --------------- START DISPATCHER LOG --------------------------
[info]  Handler Actor created successfully.
[info]   --------------- END OF DISPATCHER LOG --------------------------

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  Car information request has been done, states moving from B1 to B2 and result is sending to target
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------- START BEHAVIOR LOG ---------------
[info]  behaviorBTest is going from B1 to B2.
[info]   ---------------------------- END OF BEHAVIOR LOG ---------------
```

```
[info]   -- START TARGET LOG -------------------------------------------
[info]  T2 is sending a request for trucks.
[info]   -- END OF TARGET LOG -------------------------------------------

[info]   --------------- START DISPATCHER LOG --------------------------
[info]  targethandlerActor has been chosen to forward the request....
[info]   --------------- END OF DISPATCHER LOG --------------------------

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  Current states are A1, B2, C1.
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  Candidate Controllers for Trucks are List(Node(T2,A1,B2,C1,Trucks,B,true,Compensatable))
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  B is selected for Trucks
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  Trucks request has been done, states moving from B2 to B3 and result is sending to target
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------- START BEHAVIOR LOG ---------------
[info]  behaviorBTest is going from B2 to B3.
[info]   ---------------------------- END OF BEHAVIOR LOG ---------------
```

```
[info]   -- START TARGET LOG --------------------------------------------
[info]  T4 is sending a request for ambulance.
[info]   -- END OF TARGET LOG --------------------------------------------

[info]   --------------- START DISPATCHER LOG --------------------------
[info]  targethandlerActor has been chosen to forward the request....
[info]   --------------- END OF DISPATCHER LOG --------------------------

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  Current states are A1, B3, C1.
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  Candidate Controllers for Ambulance are List(Node(T4,A1,B3,C1,Ambulance,A,true,Compensatable))
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  A is selected for Ambulance
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  Ambulance request has been done, states moving from A1 to A1 and result is sending to target
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------- START BEHAVIOR LOG ---------------
[info]  behaviorATest is going from A1 to A1.
[info]   ---------------------------- END OF BEHAVIOR LOG ---------------



[info]   -- START TARGET LOG --------------------------------------------
[info]  T5 is sending a request for issue the ticket.
[info]   -- END OF TARGET LOG --------------------------------------------

[info]   --------------- START DISPATCHER LOG --------------------------
[info]  targethandlerActor has been chosen to forward the request....
[info]   --------------- END OF DISPATCHER LOG --------------------------

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  Current states are A1, B3, C1.
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  Candidate Controllers for TicketIssue are List(Node(T5,A1,B3,C1,TicketIssue,B,true,Compensatable)
                                                     , Node(T5,A1,B3,C1,TicketIssue,C,true,Compensatable))
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  B is selected for TicketIssue
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------------------- START HANDLER LOG ----
[info]  Ticket issue request has been done, states moving from B3 to B1 and result is sending to target
[info]   ---------------------------------------- END OF HANDLER LOG ----

[info]   ---------------------------- START BEHAVIOR LOG ---------------
[info]  behaviorBTest is going from B3 to B1.
[info]   ---------------------------- END OF BEHAVIOR LOG ---------------
```

Bibliography

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[2] R. Angarita, Y. Cardinale, and M. Rukoz. Faceta: Backward and forward recovery for execution of transactional composite WS. In E. Simperl, B. Norton, D. Mladenic, E. Della Valle, I. Fundulaki, A. Passant, and R. Troncy, editors, *The Semantic Web: ESWC 2012 Satellite Events*, volume 7540 of *LNCS*, pages 343–357, Berlin, Heidelberg, 2015. Springer.

[3] J. Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010.

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

[5] A. Bala and I. Chana. Fault tolerance-challenges, techniques and implementation in cloud computing. *International Journal of Computer Science Issues*, 9(1):288–293, 2012.

[6] M. Barati and R. St-Denis. Behavior composition meets supervisory control. In *2015 IEEE International Conference on Systems, Man, and Cybernetics*, pages 115–120. IEEE, 2015.

[7] D. Berardi, F. Cheikh, G. De Giacomo, and F. Patrizi. Automatic service composition via simulation. *International Journal of Foundations of Computer Science*, 19(2):429–452, 2008.

[8] B. Bonet and H. Geffner. Planning and control in artificial intelligence: A unifying perspective. *Applied Intelligence*, 14(3):237–252, 2001.

[9] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209, 1993.

[10] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, New York, NY, 2011.

[11] Y. Cardinale, J. El Haddad, M. Manouvrier, and M. Rukoz. CPN-TWS: a coloured petri-net approach for transactional-QoS driven web service composition. *International Journal of Web and Grid Services*, 7(1):91–115, 2011.

[12] Y. Cardinale, J. El Haddad, M. Manouvrier, and M. Rukoz. Transactional-aware web service composition: a survey. In *Handbook of Research on Service-Oriented Systems and Non-Functional Properties: Future Directions*, pages 116–141. 2011.

[13] Y. Cardinale and M. Rukoz. Fault tolerant execution of transactional composite web services: An approach. In *Proceedings of the 5th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 158–164, Wilmington, DE, 2011.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, Cambridge, MA, 1999.

[15] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[16] Y. S. Dai, M. Xie, K. L. Poh, and G. Q. Liu. A study of service reliability and availability for distributed systems. *Reliability Engineering & System Safety*, 79(1):103–112, 2003.

[17] G. De Giacomo, M. Mecella, and F. Patrizi. Automated service composition based on behaviors: The roman model. In A. Bouguettaya, Q. Z. Sheng, and F. Daniel, editors, *Web Services Foundations*, pages 189–214, New York, NY, 2014. Springer.

[18] G. De Giacomo, F. Patrizi, and S. Sardina. Behavior composition in the presence of failure. In G. Brewka and J. Lang, editors, *KR'08 Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning*, pages 640–650, Sydney, Australia, 2008.

[19] G. De Giacomo, F. Patrizi, and S. Sardina. Automatic behavior composition synthesis. *Artificial Intelligence*, 196:106–142, 2013.

[20] G. De Giacomo and S. Sardina. Automatic synthesis of new behaviors from a library of available behaviors. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, volume 7, pages 1866–1871, Hyderabad, India, 2007.

[21] V. Dialani, S. Miles, L. Moreau, D. De Roure, and M. Luck. Transparent fault tolerance for web services based architectures. In B. Monien and R. Feldmann, editors, *Euro-Par 2002 Parallel Processing*, volume 2400 of *LNCS*, pages 889–898, Berlin, Heidelberg, Germany, 2002. Springer.

[22] Y. M. Essa. A survey of cloud computing fault tolerance: Techniques and implementation. *International Journal of Computer Applications*, 138(13):34–38, 2016.

[23] P. Haller. On the integration of the actor model in mainstream technologies: The scala perspective. In *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, pages 1–6, New York, NY, 2012.

[24] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 453–462, Washington, DC, 1995. IEEE.

[25] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, San Francisco, CA, 1973.

[26] H. Jamjoom, D. Williams, and U. Sharma. Don't call them middleboxes, call them middlepipes. In *Proceedings of the 3rd workshop on Hot topics in software defined networking*, pages 19–24, New York, NY, 2014.

[27] A. Liu, Q. Li, L. Huang, and M. Xiao. Facts: A framework for fault-tolerant composition of transactional web services. *IEEE Transactions on Services Computing*, 3(1):46–59, 2010.

[28] Y. Lustig and M. Y. Vardi. Synthesis from component libraries. *International Journal on Software Tools for Technology Transfer*, 15(5–6):603–618, 2013.

[29] N. Menadjelia. Towards a formal study of automatic failure recovery in protocol-based web service composition. *Service Oriented Computing and Applications*, 10(2):173–184, 2016.

[30] R. Milner. An algebraic definition of simulation between programs. In D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, San Francisco, CA, 1971.

[31] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. In E. A. Emerson and K. S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *LNCS*, pages 364–380, Berlin, Heidelberg, 2006. Springer.

[32] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *LNCS*, pages 184–195, Berlin, Heidelberg, 1996. Springer.

[33] F. Rabbi, H. Wang, and W. MacCaull. Compensable workflow nets. In J. S. Dong and H. Zhu, editors, *Formal Methods and Software Engineering*, volume 6447 of *LNCS*, pages 122–137, Berlin, Heidelberg, 2010. Springer.

[34] G. Randelli, L. Marchetti, F. A. Marino, and L. Iocchi. Multi-agent behavior composition through adaptable software architectures and tangible interfaces. In J. Ruiz del Solar, E. Chown, and P. G. Plöger, editors, *RoboCup 2010: Robot Soccer World Cup XIV*, volume 6556 of *LNCS*, pages 278–290, Berlin, Heidelberg, 2011. Springer.

[35] A. S. Rao. Agentspeak(l): BDI agents speak out in a logical computable language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away*, volume 1038 of *LNCS*, pages 42–55, Berlin, Heidelberg, 1996. Springer.

[36] C. M. F Rubira, R. de Lemos, G.R.M. Ferreira, and F. Castor Filho. Exception handling in the development of dependable component-based systems. *Software: Practice and Experience*, 35(3):195–236, 2004.

[37] F. Saglietti. Qualitative and quantitative analysis of software fault tolerance. *IFAC Proceedings Volumes*, 26(2):657–660, 1993.

[38] J. G. Shanahan and L. Dai. Large scale distributed data science using Apache Spark. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2323–2324, New York, NY, 2015.

[39] M. Ter Beek, A. Bucchiarone, and S. Gnesi. Formal methods for service composition. *Annals of Mathematics, Computing & Teleinformatics*, 1(5):1–10, 2007.

[40] M. Thurau. Akka framework. *University of Lübeck*, 2012.

[41] Y. Wang, Y. Fan, and A. Jiang. A paired-net based compensation mechanism for verifying web composition transactions. In *Proceedings of the 2010 4th Inter-*

national Conference on New Trends in Information Science and Service Science, pages 1–6, 2010.

[42] W. Yu. Fault handling and recovery in decentralized services orchestration. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, pages 98–105, New York, NY, 2010.

[43] Y. Zhang, Z. Zheng, and M. R. Lyu. BFTCloud: A byzantine fault tolerance framework for voluntary-resource cloud computing. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 444–451, 2011.

[44] Z. Zhao, J. Wei, L. Lin, and X. Ding. A concurrency control mechanism for composite service supporting user-defined relaxed atomicity. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 275–278, Washington, DC.

[45] Z. Zheng and M. R. Lyu. A QoS-aware fault tolerant middleware for dependable service composition. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 239–248, 2009.