UNIVERSITÀ DEGLI STUDI DELL'AQUILA
Department of Information Engineering, Computer Science and Mathematics

Doctoral Program in Information and Communication Technology
Curriculum of emerging computing models, software architectures,
and intelligent systems
XXXIII Cycle

# Advanced Model-Driven Techniques to Improve Non-Functional Properties of Software Systems

SSD INF/01 Informatica

Candidate
**Michele Tucci**

Advisor:
Romina Eramo

Co-Advisor:
Vittorio Cortellessa

Doctoral Program Supervisor:
Vittorio Cortellessa

A.Y. 2019/2020

# Abstract

Supporting changes in software models is becoming increasingly important. Some of these changes are induced by non-functional aspects that are nowadays analyzed throughout the entire software life-cycle. However, non-functional analysis is generally based on models, tools and data that are different from the ones usually adopted for software functional design. Therefore, it becomes crucial to develop methods to reduce the gap between design knowledge and non-functional analysis context. To this extent, in the last two decades, a number of approaches, mostly based on Model-Driven Engineering (MDE) like model transformation, have been introduced to generate non-functional analysis models from software models. However, when analysis models are modified to meet non-functional requirements, changes are not propagated back to the software model, whereas the automated identification and propagation of changes would better support a round-trip analysis process. Moreover, once the system is built and running in production, advanced techniques are required to continuously assess and improve non-functional properties. In the last few years, design-runtime interactions proved to be effective in addressing this problem on a number of practical scenarios.

In this PhD thesis we intend to bridge this gap by providing solutions to: (i) support complex change propagation and model synchronization scenarios by developing advanced model-driven techniques to enhance bidirectional model transformation and traceability management; (ii) generate availability analysis models from software models, and automatically propagate changes, driven by availability requirements satisfaction, from analysis models back to software models; (iii) relate design and runtime artifacts to detect potential performance problems and resolve them through the automated refactoring of models as well as of the running system; (iv) assist the detection of root causes of defects in safety-critical systems, especially when such defects only manifest after deployment.

The model-driven techniques developed in this thesis are specifically targeted at change propagation and traceability management problems. They are mainly implemented within the Janus Transformation Language (JTL), a constraint-based model transformation framework designed to support bidirectionality and change

propagation. The approaches we realized on the basis of these techniques are applied to different non-functional properties such as performance, availability, reliability and safety. We have been able to validate our approaches on several case studies and benchmarks from both literature and industry.

# List of publications

- Eramo, R., Pierantonio, A., and Tucci, M. (2018). Enhancing the JTL tool for bidirectional transformations. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018*, pages 36–41. ACM.
  https://doi.org/10.1145/3191697.3191720

- Eramo, R., Pierantonio, A., and Tucci, M. (2018). Improved traceability for bidirectional model transformations. In *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018*, volume 2245 of *CEUR Workshop Proceedings*, pages 306–315. CEUR-WS.org.
  http://ceur-ws.org/Vol-2245/mdetools_paper_1.pdf

- Tucci, M. (2018). Model-driven round-trip software dependability engineering. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, pages 186–191, ACM.
  https://doi.org/10.1145/3270112.3275337

- Cortellessa, V., Eramo, R., and Tucci, M. (2018). Availability-driven architectural change propagation through bidirectional model transformations between UML and Petri net models. *In IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018*, pages 125–134. IEEE Computer Society.
  https://doi.org/10.1109/ICSA.2018.00022

- Di Pompeo, D., Tucci, M., Celi, A., and Eramo, R. (2019). A Microservice Reference Case Study for Design-Runtime Interaction in MDE. In *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019), Eindhoven, The Netherlands, July 15 - 19, 2019*, volume 2405 of *CEUR Workshop Proceedings*, pages 23–32. CEUR-WS.org.
http://ceur-ws.org/Vol-2405/06_paper.pdf

- Arcelli, D., Cortellessa, V., Di Pompeo, D., Eramo, R., and Tucci, M. (2019). Exploiting architecture/runtime model-driven traceability for performance improvement. In *IEEE International Conference on Software Architecture, ICSA 2019, Hamburg, Germany, March 25-29, 2019*, pages 81–90. IEEE.
https://doi.org/10.1109/ICSA.2019.00017

- Eramo, R., de Kerchove, F. M., Colange, M., Tucci, M., Ouy, J., Brunelière, H., and Di Ruscio, D. (2019). Model-driven design-runtime interaction in safety critical system development: an experience report. In *Journal of Object Technology*, 18(2):1:1–22.
https://doi.org/10.5381/jot.2019.18.2.a1

- Cortellessa, V., Eramo, R., and Tucci, M. (2020). From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement. In *Journal of Information and Software Technology*, 127:106362.
https://doi.org/10.1016/j.infsof.2020.106362

# Table of contents

# List of figures

# List of tables

# Chapter 1

# Introduction

During the last two decades, a major challenge for researchers working on software modeling and evaluation has been the assessment of non-functional properties, such as performance, schedulability, dependability or security. This is mostly due to the increasingly demanding user requirements, on one side, and to the pervasiveness of software over heterogeneous platforms on another side. Attributes like performance and reliability are very complex to analyze, because the designer is usually able to assess them only late in the life cycle, when most of the system has been designed and implemented. Nevertheless, if non-functional requirements are not met, then a variety of negative consequences (such as user dissatisfaction, lost income, etc.) can impact on significant parts of a project [112]. This has led to a widespread use of quantitative models for the assessment of non-functional properties from the early phases of software design [38]. To this extent, a considerable number of approaches, mostly based on Model-Driven Engineering (MDE) [110] techniques like model transformation, has been proposed in the last decade to automatically generate non-functional models from software models [23, 39]. An increased number of companies are embedding MDE approaches in their processes, with the perceived benefit of enabling developers to work at a higher level of abstraction and to rely on automation throughout the development process. Nevertheless, MDE solutions need to be further developed to deal with real-life scenarios emerging from industrial projects.

The general process for a model-driven assessment of non-functional requirements on software is outlined in Figure 1.1. Starting from a software model $M_1$ (top-left corner) annotated with non-functional properties, an initial analysis model $AM_1$ is obtained in the first step. This first step is usually called the *forward path* from software models to analysis models. Although a designer could perform this step manually, we are only interested in the automated generation of analysis models. Model $AM_1$ is

then solved in order to obtain analysis indices $\mathrm{IND}_1$ (e.g., response time, probability of failure, etc.). These indices may not meet some non-functional requirements, thus leading the designer to perform changes ($\Delta_1$) to the initial analysis model ($\mathrm{AM}_1$) with the aim of improving the model. The refactored analysis model $\mathrm{AM}_2$ and analysis indices $\mathrm{IND}_2$ are obtained in this way. This process is repeated until a satisfactory analysis model $\mathrm{AM}_k$ is obtained when the corresponding analysis indices $\mathrm{IND}_k$ meet the requirements. However, a satisfactory analysis model has undergone several refactoring steps, hence it no longer corresponds to the initial software model.

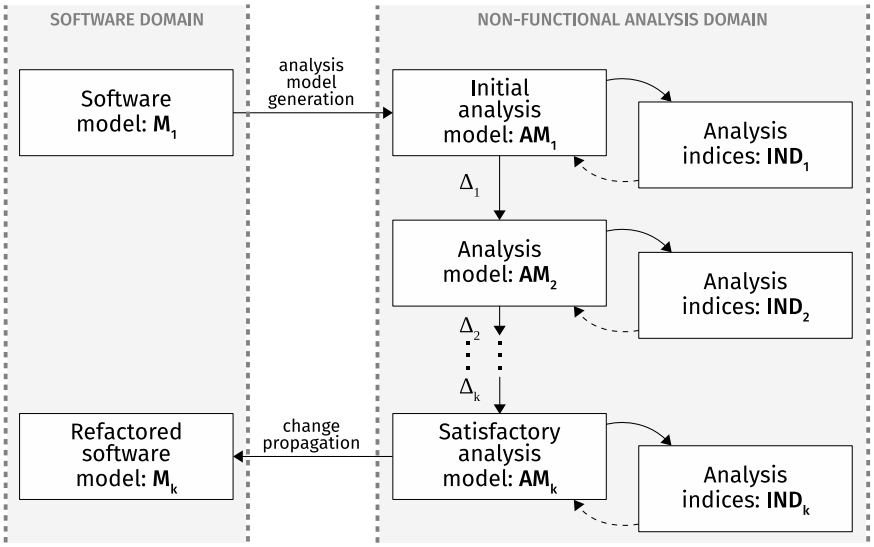



Fig. 1.1 Model-driven round-trip assessment of non-functional properties

Hence, the next task should be to somehow replicate in the software domain the changes performed in the non-functional analysis domain. This phase, usually called the *backward path*, corresponds to the change propagation step in Figure 1.1. Such a step is conducted by interpreting the feedback gained during the analysis to produce a refactored software model ($\mathrm{M}_k$). This informal process of deriving changes is inherently difficult and usually based on the designer expertise and domain knowledge.

An alternative to the realization of an automated *backward path* would be to never modify models in the non-functional analysis domain and apply modifications only on the software domain. A study by Arcelli et al. [10] compares these two alternative methods that derive software changes from the results of a model-based performance analysis. Both methods represent valid alternative processes depending on the context. However, a large variety of approaches have been proposed to implement the *forward path*, especially in the performance analysis domain [6, 61], whereas there is still lack of automated solutions for the *backward path*. For this reason, our intent is to mainly

contribute to the latter alternative. Moreover, introducing automation in this step goes beyond the mere advantage of achieving automated refactoring. An automated round-trip process allows performing the non-functional validation on the same models, languages, and tools from which analysis results are obtained, without the need for tentatively modifying the software model until the generated non-functional model meets the requirements. Finally, change propagation may as well serve as a mean to explore software design alternatives induced by the improvement of non-functional properties.

Although models provide an abstraction to reason about the properties of the system to be built, the knowledge already embedded in them can be further exploited beyond the early design phases of software development. When the implemented system is deployed and executed in the real world, some modeling assumption, like those about the execution environment, may prove to be inadequate, as they may reflect only partially what happens in reality. For instance, certain user behaviors that were assumed to be unusual may prove to be quite common or, as another example, resource demands of the actual implementation may differ from those estimated at design time. In addition, recent trends in software development indicate that modern software systems need to constantly evolve and adapt to changing environmental conditions. As a consequence, the assessment of non-functional properties should extend to *run time* and become an ongoing activity. This led to the need for monitoring facilities that can continuously detect changes in the environment. In this context, relating design knowledge of the system to the information that is constantly emerging from the run time phase can provide a more comprehensive view in support of non-functional assessment and improvement activities. However, to achieve this goal, advanced techniques are required to relate design and run time artifacts.

The specific challenges described so far may benefit from specialized solutions that should integrate with the model-based software engineering processes already in use. Providing automated solutions to these problems often requires to constantly propagate changes and synchronize modeling artifacts. In MDE, bidirectional model transformations are a key mechanism to propagate changes and maintain the consistency of related models. For these reasons, bidirectional transformations represent an ideal candidate to provide the foundation on which specialized solutions can be built.

In this PhD thesis, we intend to tackle these challenges by providing solutions to:

- support complex change propagation and model synchronization scenarios by developing advanced model-driven techniques to enhance bidirectional model transformation and traceability management;

- generate availability analysis models from software models, and automatically propagate changes, driven by availability requirements satisfaction, from analysis models back to software models;
- relate design and runtime artifacts to detect potential performance problems and resolve them through the automated refactoring of models as well as of the running system.
- assist the detection of root causes of defects in safety-critical systems, especially when such defects only manifest after deployment.

The model-driven techniques developed in this thesis are specifically targeted at change propagation and traceability management problems. They are mainly implemented within the Janus Transformation Language (JTL) [36, 49], a constraint-based model transformation framework designed to support bidirectionality and change propagation. The approaches we realized on the basis of these techniques are applied to different non-functional properties such as performance, availability, reliability and safety. We have been able to validate our approaches on several case studies and benchmarks from both literature and industry.

While providing model-based solutions for non-functional validation was the main objective of this PhD, we also wanted to investigate how performance are usually assessed during the life cycle of open source software projects. The latter are especially valuable because of their inherent propensity to share not only the source code but also the collaboration activities and the decision making process taking place during their life cycle. More specifically, in this context, we were interested in refactoring operations, intended as behavior preserving changes, that are usually performed to improve the code quality. Therefore, we conducted a study to estimate the impact of this kind of refactoring activities on the software execution time, which is measured through benchmarks already put in place by project developers. Consequently, we here also present the findings of this study, which is the result of an ongoing collaboration among our whole research group at University of L'Aquila, University of Molise, and the Software Institute of the Università della Svizzera Italiana.

The thesis is structured as follows:

**Chapter 2** describes the basic concepts used in the thesis. It introduces bidirectional model transformations, traceability management, as well as general notions in the model-based assessment of non-functional properties.

**Chapter 3** describes how bidirectional model transformations and advanced traceability management has been implemented in the JTL framework. It also provides usage examples of such techniques on two case studies from the literature.

**Chapter 4** introduces an approach for the improvement of availability through bidirectional model transformations. It shows how to generate Petri Nets from UML models, how to refactor the obtained Petri Nets to improve the steady state availability of the systems, and how to propagate such changes back to the UML model.

**Chapter 5** introduces an approach for continuous performance engineering in microservices-based systems. Starting from UML models and monitoring logs, the approach is able to detect performance antipatterns, recommend refactoring actions to remove them, and perform such refactorings on a running application in order to improve its performance.

**Chapter 6** introduces an approach, based on an industrial case study, for the assessment of safety properties. By relating log information with the formal method specifications of the system, the approach is able to support safety analysts in the root cause analysis of intermittent and recurring malfunctionings.

**Chapter 7** shortly presents an empirical study on the impact of maintainability refactoring on execution time.

**Chapter 8** outlines the thesis more general contributions and summarizes main results. I also discuss current limitations in the presented research and propose potential future work.

# Chapter 2

# Background

In this chapter, we introduce basic concepts that will be used throughout the thesis. First, we present the definitions of different types of non-functional properties, along with general ideas behind the assessment of such properties through the use of models. Furthermore, we provide the main notions in bidirectional model transformations and traceability management in order to establish the specific model-driven context in which this research work is conducted.

## 2.1 Non-functional validation of software

Non-functional validation consists of checking whether the system under development meets non-functional requirements at any stage of the software life cycle. Such validation is usually performed through the analysis of the produced software artifacts. Requirements play a key role in software validation, as they represent the target against which the advancement of development is measured. Non-functional requirements are usually expressed through software metrics that quantify certain system attributes that can be collected in two groups of attributes: dependability and performance.

Avizienis et al. [14] define and classify non-functional attributes related to dependability. The original definition of dependability is *the ability to deliver service that can justifiably be trusted*. While this definition stresses the need for justification of trust, an alternative definition states that the dependability of a system is *the ability to avoid failures that are more frequent and severe than acceptable*. In this case the definition provides the criterion for deciding if a service is dependable. Dependability encompasses five attributes [14]:

- reliability, the continuity of correct service,

- availability, the readiness for correct service,
- maintainability, the ability to undergo modifications and repairs,
- integrity, the absence of improper system alterations,
- safety, the absence of catastrophic consequences on the users and environment.

Performance is not included in the definition of dependability, although in some domains performance failures are as critical as other types of failures. As a general definition, performance measures how effective is a software system with respect to time constraints and allocation of resources. Traditional performance indices are *response time*, *throughput* and *utilization*. Response time is the end-to-end time that a task spends to traverse a certain path within the system. Throughput is the number of jobs that can be completed per unit of time by a certain part of the system. Utilization is the percentage of time that a certain part of the system is busy working.

Performance needs appropriate notations to be represented. These notations aim at capturing static and dynamic characteristics of the software system, as well as the parameters that represent the stochastic behavior of the system. The performance model parameters can be partitioned in the following main categories: (i) the operational profile, (ii) the workload, (iii) the resource demands. The operational profile is a collection of data that stochastically represent how the system is used by different types of users. In particular, the operational profile defines the relative frequencies with which different kind of users will interact with the system. The workload represents the intensity of system invocations from users. It can be characterized globally for any type of users or specifically for each type. The resource demands represent the amount of resources that a piece of software (at any level of abstraction) requires to complete its task. This parameter points out that a certain amount of information about the target hardware platform is necessary in order to model and analyze software performance.

Non-functional attributes can be quantified through metrics that represent the indices of quality of a software product. Specific notations (such as Bayesian Belief Networks, Fault Trees, Queueing Networks) have been introduced to evaluate these indices starting from their parameters. Such notations allow one to build non-functional models that express the relationships between software model characteristics and non-functional indices. However, the information required to build non-functional models does not completely reside in the software models that are produced for the functional specification and validation of the software. From software models is possible to derive knowledge about the structure and behavior of the software, but, in order to build non-functional models, quantified parameters, as the one we mentioned before, are also

required. This represented a significant gap between software development practices and non-functional validation.

In the last two decades, these reasons led to many approaches aimed at reducing this gap through the introduction of automation in non-functional validation activities. The majority of such approaches augments software models with the quantified parameters needed for the non-functional validation, and subsequently employs model-driven techniques, like model transformations, to automatically generate non-functional models from software models. Two attributes appear crucial to make any approach acceptable by the software community: transparency, that is trying to keep a minimal impact on the software notation and the software process adopted, and effectiveness, meaning that the techniques to annotate and transform software models into performance models should not exceed in complexity [39].

## 2.2 Model-driven techniques for non-functional validation

As said in the previous section, non-functional validation is usually performed on quantitative models. Such models significantly differ from the ones used to design the architecture of the system to be built, as they may retain different details, provide a different view point, or establish a different abstraction level. This has led to the proliferation of approaches to automatically generate non-functional models from software models. Among such approaches, those based on model-driven engineering have proved to be effective in tackling the problem by exploiting techniques like model transformations [23, 39]. For the purposes of this research work, we are going to focus on two specific model-driven engineering techniques: bidirectional model transformations and traceability management. The following introduces the general concepts behind these techniques.

### 2.2.1 Bidirectional model transformations

The main goal of model transformation is automating routine aspects of the software development and maintenance process in order to make it more efficient. For instance, a common scenario may be the restoring of consistency between two models that are independently altered during the software life-cycle. In such a case, the process of restoring consistency can be codified and therefore automated through a model transformation. The simplest case may be one in which only one of the two models is

ever modified by designers and the other is generated by a tool. One may view such a transformation as deterministic, in the sense that, in most model transformation formalisms, given an input model (*source*), the tool will always generate the same output model (*target*). This is a case where there is no need to check consistency because the output model can be considered just the result of the application of a function to an input model.

Arguably, a more typical situation is when both models are modified by designers, leading to round-trip scenarios where consistency must be assessed, and eventually restored, at every step. This represents an inherently difficult problem because there may be several ways to restore consistency and it is extremely challenging to formalize rules which capture which way is best in a deterministic way [115]. Bidirectional transformations are a way of dealing with such situations by specifying how consistency should be restored through the modification of one the involved models. In these cases, two models can be considered consistent if it is acceptable to all stakeholders to proceed with these models, without modifying either [115].

In general, if the sets of models are $M$ and $N$, a transformation can be defined as a function $f\colon M \to N$ where $m$ and $n$ are consistent iff $n = f(m)$. This consistency relation is a bijective relation if and only if $f$ is a bijective function, and in that case there exists the inverse function $f^{-1}\colon N \to M$. However, this represents a very restrictive case in which different formats for the same information must be synchronized. In practice, this is rarely the case as one notation may be an abstraction of the other or, more generally, the two models pertain to different abstractions which retain different information. Therefore, to be effective in practical scenarios, bidirectional transformation languages should provide ways of specifying partial and non-bijective relations [116].

### 2.2.2 Traceability

Traceability is the ability of establishing relations between two or more products of a development process, especially when those products have a predecessor-successor or master-subordinate relationship to one another [56]. Traceability mechanisms support the identification of the origin and rationale of software artifacts, as they help in understanding the relationships between the artifacts across software development stages. For instance, a Java class can be traced back to its class in a class diagram, but also to the requirement that motivates its presence in the system.

In the general meaning of the term, traceability approaches can be classified in requirements-driven, when traces are used to connect requirements specifications to

software artifacts, modelling approaches, that are interested in how metamodels, models or conceptual frameworks are involved in the tracing process, and model transformation approaches, where traces are generated to relate elements during the execution of a transformation. In this thesis, we are interested in the case where traceability is employed to connect elements of the source model to elements of the target model, that is often the case in model-driven engineering.

Since model-driven engineering supports automating the creation and the discovery of relations among models, model transformations can be considered as a compelling mechanism to generate trace links. As a consequence, there exist transformation languages that support the automatic creation and usage of traceability links between models. Such links may have the primary use of enhancing the execution of model transformation providing additional information about how the target elements are created. However, in some cases, traceability information can serve other purposes, for example, by re-using it to derive some quality attributes of the transformation or to support a performance analysis.

# Chapter 3

# Advanced Model-Driven Techniques for Non-Functional Validation

In Model Driven Engineering (MDE), bidirectional model transformations are considered a core ingredient for managing both consistency and synchronization of two or more related models. Their relevance has been advocated by a number of approaches that have been proposed in the last decade. Most of these approaches are characterized by a set of specific properties pertaining to a particular applicative domain [42, 65].

An aspect which is still largely underestimated is the multiplicity of solutions: if the forward mapping of a bidirectional transformation is non-bijective, e.g., it maps two distinct models to the same target model, then the corresponding backward mapping must be a *one-to-many* mapping (*non-determinism*) [49]. Consequently, since current languages are able to generate only one model, non-deterministic transformations involved in round-tripping can give rise to results which are somewhat unpredictable. In these cases, the solution is normally identified according to heuristics or by considering rules order [129, 114]. A typical example in which bidirectionality is needed is the *Collapse/Expand State Diagrams* benchmark defined in [41]: starting from a hierarchical state diagram (involving some one-level nesting), a flat view has to be provided. In this case, adding a transition between two flatten states cannot be back-propagated in a unique way, since the corresponding transition in the hierarchical state diagram can be added to any nested states as well as to the container state itself.

The Janus Transformation Language (JTL) [36, 49] is a constraint-based model transformation language specifically tailored to support bidirectionality and change propagation. Its relational semantics relies on a constraint solver to find a consistent choice for the other source when multiple choices are possible. Thus, the responsibility of choosing the right model among the generated ones is left to the designer.

The development of JTL started in 2010 with the major goal including the specification of bidirectional transformations within an engine based on Answer Set Programming (ASP) [54]. Its prototype version was presented in [36] as a set of independent Eclipse plugins for academic use. In the last few years, there has been a lot of progress in the Eclipse Modeling Framework (EMF)[1] and it is now established as the de facto standard for models representation and manipulation, offering stable and well-tested components. As a consequence, we wanted to re-engineer the existing JTL features and integrate them in a dedicated Eclipse product. In particular, the Eclipse Rich Client Platform (RCP)[2] has been used as a basis to create a feature-rich stand-alone application. In summary, the following improvements have been contributed by this thesis:

- the semantic anchoring between the JTL syntax and the ASP engine is completely restructured using ATL [68] and EMFText[3] technologies;
- the existing transformation engine has been modified to solve shortcomings that emerged from test cases;
- we introduced a new advanced traceability engine which was designed to better support partial and non-bijective transformations;
- the DLV solver system [85] has been updated and integrated in the overall environment to provide interoperability with EMF;
- usability, performance and multi-platform support have been also improved in the process.

This chapter presents the new tool for JTL[4] by using a running example as well as a benchmark from the literature to show traceability management in action.

## 3.1 The Janus Transformation Language

Within the JTL framework, the user can define models and metamodels by exploiting the EMF environment, whereas bidirectional transformations can be specified in a textual representation by using the JTL syntax.

Considering the *Collapse/Expand State Diagrams* benchmark [41], List. 3.1 shows a fragment of the HSM2SM bidirectional transformation, which relates hierarchical and flat state machines (refer to Appendix A for the entire transformation). To specify a

[1] https://www.eclipse.org/modeling/emf/
[2] https://wiki.eclipse.org/Rich_Client_Platform
[3] http://www.emftext.org
[4] http://jtl.di.univaq.it/

number of *relations* among elements of the two involved *domains*, JTL adopt a syntax inspired by QVT-R. In particular, the Line 1 of the listing declares the variable `hsm` that matches models conforming to the metamodel `HSM`, and the variable `sm` that matches models conforming to the metamodel `SM`. The relation `StateMachine2StateMachine` in Lines 2-11 relates state machines in the different metamodels. The *when* and *where* clauses specify pre- and post-conditions on the relation. In particular, the where clause in Lines 5-10 invokes a set of relations that are held if the calling relation is enforced. For instance, the relation `ownedState2ownedState`, declared in Lines 12-22, defines a correspondence between the reference `ownedState` of type `State` belonging to the hierarchical domain and the reference `ownedState` of type `State` belonging to the flat domain. Note that, the states have the same variable name (`s`) and that the relation `State2State(s,s)` is invoked in the where clause. It means that if a state machine have a reference to a state `s` in the source domain, a correspondent reference to a state `s` must be created in the target domain. The correspondent state `s` is created by enforcing the relation `State2State`. Similarly, the relation `CompositeState2State` relates composite states in the hierarchical domain to state in the flat domain, and vice-versa. Note that, the two relations, `State2State` and `CompositeState2State`, make the transformation *non-injective*. In fact, in the backward direction an object of type `State` may be equally mapped to objects of type `State` or `CompositeState`.

```
1 transformation HSM2SM(hsm : HSM, sm : SM) {
2   top relation StateMachine2StateMachine {
3     enforce domain hsm hm : HSM::StateMachine {  };
4     enforce domain sm m : SM::StateMachine {  };
5     where {
6       ownedState2ownedState(hm,m);
7       ownedCompositeState2ownedState(hm,m);
8       ownedTransition2ownedTransition(hm,m);
9       ...
10    }
11  }
12  relation ownedState2ownedState {
13    enforce domain hsm hm : HSM::StateMachine {
14      ownedState = s : HSM::State { }
15    };
16    enforce domain sm m : SM::StateMachine {
17      ownedState = s : SM::State { }
18    };
19    where {
20      State2State(s,s);
21    }
```

```
22  }
23  relation State2State {
24    varName : String;
25    enforce domain hsm hs : HSM::State {
26      name = varName
27    };
28    enforce domain sm s : SM::State {
29      name = varName
30    };
31  }
32  relation ownedCompositeState2ownedState {
33    ...
34  }
35  relation CompositeState2State {
36    varName : String;
37    enforce domain hsm hs : HSM::CompositeState {
38      name = varName
39    };
40    enforce domain sm s : SM::State {
41      name = varName
42    };
43  }
44  ...
45  relation ownedTransition2ownedTransition {
46    ...
47  }
48  relation Transition2Transition {
49    varTrigger : String;
50    varEffect : String;
51    enforce domain hsm ht : HSM::Transition {
52      trigger = varTrigger,
53      effect = varEffect
54    };
55    enforce domain sm t : SM::Transition {
56      trigger = varTrigger,
57      effect = varEffect
58    };
59    where {
60      TransitionSource2TransitionSource(ht, t);
61      TransitionTarget2TransitionTarget(ht, t);
62      TransitionSourceComposite2TransitionSource(ht, t);
63      TransitionTargetComposite2TransitionTarget(ht, t);
64      TransitionTargetIntoComposite2TransitionTargetComp(ht, t);
65    }
```

```
66  }
67  relation TransitionSource2TransitionSource {
68    enforce domain hsm ht : HSM::Transition {
69      source = s : HSM::State { }
70    };
71    enforce domain sm t : SM::Transition {
72      source = s : SM::State { }
73    };
74    when {
75      State2State(s, s);
76    }
77  }
78  relation TransitionTarget2TransitionTarget {
79    enforce domain hsm ht : HSM::Transition {
80      target = hs : HSM::State { }
81    };
82    enforce domain sm t : SM::Transition {
83      target = s : SM::State { }
84    };
85    when {
86      State2State(hs, s);
87    }
88  }
89  relation TransitionSourceComposite2TransitionSource {
90    ...
91  }
92  relation TransitionTargetComposite2TransitionTarget {
93    ...
94  }
95  relation TransitionTargetIntoComposite2TransitionTargetComp {
96    enforce domain hsm ht : HSM::Transition {
97      target = hs : HSM::State {
98        owningCompositeState = s:HSM::CompositeState { }
99      }
100   };
101   enforce domain sm t: SM::Transition {
102     target = s : SM::State { }
103   };
104   when {
105     CompositeState2State(s,s);
106   }
107 }
108 ...
```

```
109 }
```

Listing 3.1 A fragment of the HSM2SM transformation.

The `Transition2Transition` in Lines 48-66 relates transitions in the two different metamodels. The where clause invokes a list of relations that have the purpose of setting the correspondent source and target elements of the transitions. For instance, the relation `TransitionSource2TransitionSource` involves transitions of the two metamodels and relates their references `source` of type `State`. The relation is constrained by means of the when clause, implying that only states that have been generated from the relation `State2State` are considered (it excludes sub-states). Finally, the relation `TransitionTargetIntoComposite2TransitionTargetComposite` manages the case in which the target of a transition refers to a sub-state and it must be mapped in a correspondent transition, which targets the state that correspond to the composite state in which the sub-state belongs. Even in this case, the relations involving transitions may cause multiple solutions when the transformation is executed in backward direction. In fact, a transition that involves a state can be equally mapped in a transition that involves a state or a composite state.

The described relations are bidirectional, in fact both the involved domains are specified with the construct *enforce*.

## 3.2 JTL engine and semantics

The JTL engine is based on a relational and declarative approach implemented using ASP, that is a form of declarative programming oriented towards difficult (primarily NP-hard) search problems and based on the stable model (answer set) semantics of logic programming. JTL exploits the benefits of logic programming, enabling the specification of relations between source and target types by means of predicates, and intrinsically supports bidirectionality [42] in terms of unification-based matching, searching, and backtracking facilities. More precisely, model transformations specified in JTL are transformed into ASP programs (search problems), then an ASP solver is executed to find all the possible stable models, which are sets of atoms consistent with the considered rules and supported by a deductive process.

Figure 3.1 depicts the overall environment supporting the execution of JTL transformations. The *JTL engine* is written in the ASP language and makes use of the *DLV solver* to execute transformations in both forward and backward directions. The engine executes JTL transformations which have been written in a QVT-like syntax, and then automatically transformed into ASP programs (see the *map* arrows in Figure 3.1).

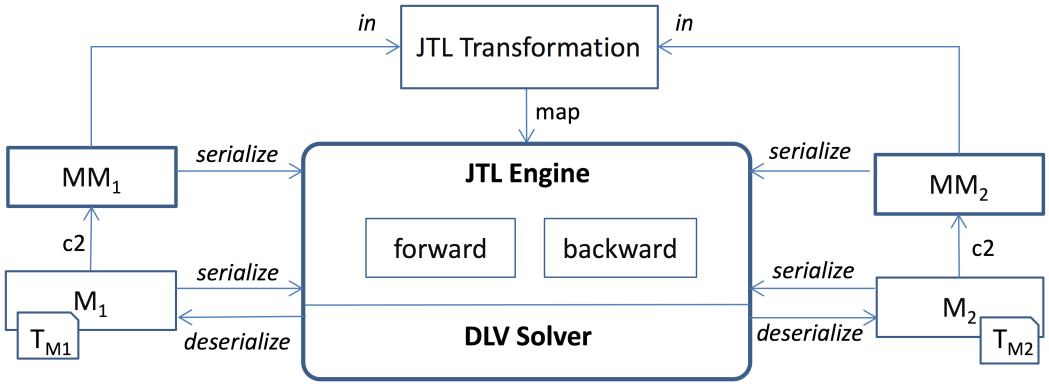



Fig. 3.1 Overview of the JTL engine.

Such semantic anchoring has been implemented in terms of an ATL transformation defined on the JTL and ASP metamodels. Moreover, the source and target metamodels of the considered transformation ($MM_1$, $MM_2$) are automatically encoded in ASP and managed by the engine during the execution of the considered transformation in order to generate the output models.

Starting from the encoding of the involved metamodels and the source model $M_1$[5], the representation of the target model ($M_2$) is generated according to the JTL specification (see the *serialize* and *deserialize* arrows). The execution of the backward direction is performed by giving as input the obtained $M_2$ and optionally the trace model $T_{M2}$. When the trace model $T_{M2}$ is provided, the correspondent $M_1$ is re-generated together with the related trace model $T_{M1}$, otherwise JTL will treat the execution as new one starting from $M_2$, and it will consequently generate all the solution candidates conforming to $MM_1$ ($M_1$ will be one of the candidates). Note that, trace models are generated during the execution so that each one refers to a specific execution and is related to a specific solution model. That is, in case of multiple solutions, multiple trace models (one for model) are generated.

### 3.2.1 Model transformation execution

When a transformation is executed, the JTL framework generates an ASP program from the transformation specification. Such program is composed of *i) rules*, which describe mappings and correspondences among element types of the source and target metamodels, and *ii) constraints*, which specify restrictions that must be satisfied on the given relations when executing the corresponding mappings.

[5]Note that, since JTL provides declarative specifications, the role of source and target models is determined at execution time

The transformation process can be summarized into the following steps:

1. given the input metamodels and the input model, the execution engine induces all the possible solution candidates (*answer sets*) according to the specified rules;

2. the set of candidates is refined by means of constraints.

The invertibility of transformations is obtained by means of trace information connecting source and target elements. Traceability can be considered as an intrinsic property of ASP, in fact each obtained answer set is composed of elements that are explicitly derived from elements of the initial stable model. In essence, tracing information describes how a target element has been generated starting from a source one.

Elements involved in non-bijective relations (that are source of non-determinism) can be maintained by means of tracing information. For instance, if we consider the backward execution of the HSM2SM transformation and the relations `State2State` and `CompositeState2State`, a state in the *sm* domain can be translated and linked in both a state or a composite state in the *hsm* domain. By exploiting traceability during the transformation process, the relations between models created by the transformation executions can be stored to permanently preserve mapping information.

Finally, all the source elements lost during the forward transformation execution (for example, due to the different expressive power of the metamodels) are stored in tracing information in order to be generated again in the backward transformation execution.

**Constraining the solution space.** The number of alternatives that we may obtain depends on the intrinsic characteristics of the transformation and on the model elements matched by the non-bijective rules. In other words, the number of alternatives depends on the degree of non-determinism of the model transformation. As said above, the execution engine may generate a large number of alternatives when only the information encoded in the transformation is used. The constraints play a key role in the transformation process. In particular, transformations can be mapped into logical rules which are constrained by context information consistently narrowing the solution space to only those models which are relevant.

The answer sets may be refined in subsequent steps: *i)* the answer set is filtered according to the constraints induced by the source metamodel, *ii)* the answer set is further reduced by considering the constraints induced by the tracing information and *iii)* additional user-defined constraints can be added to browse the solution space.

Furthermore, the use of constraints may reduce back-tracking because it allows for early detection of dead-ends. In practice, constraints can be written a posteriori and given as input of the transformation engine. The designer can decide to use such constraints for a specific execution or to integrate them with the transformation specification itself to avoid inconsistencies between executions.

Even if adding constraints can help to discard alternatives, non-determinism may generally cause a combinatorial explosion of solutions. In order to reduce the burden of managing a collection of models, multiple solutions can be represented in a intensional manner by adopting a model for uncertainty [49].

### 3.2.2 Execution on the HSM2SM benchmark

In this section we show the execution of the HSM2SM transformation within the JTL tool[6]. We considered the scenario presented in [36]: starting from the definition of the involved metamodels, the JTL transformation is specified as described in Listing 3.1 on page 15). By referring to Figure 3.1, the transformation, the source and target metamodels and the source model have been created and need to be translated in their ASP encoding in order to be executed from the JTL engine.

After this phase, the application of the HSM2SM transformation on the source model `HSM.xmi` generates the corresponding target model `SM.xmi`, as depicted in Figure 3.2. Together with the `.xmi` model, the engine generates the correspondent ASP encoding `SM.aspm` and the trace model `SM_trace.aspm`. The latter is composed of a set of elements encoded in ASP with the role maintaining relations between source and target elements of the execution and storing elements that are lost during the transformation or derive from non-injective mappings. In fact, by re-applying the transformation in the backward direction, it is possible to obtain again the `HSM` source model. The missing sub-states and the transitions involving them are restored by means of trace information.

Suppose that, in a refinement step, the obtained target model is manually modified and an updated model `SM'.xmi` is obtained as shown in the left part of Figure 3.3. In particular, the state `Begin Installation` is renamed in `Start Install` and a new transition `[alternative = try again]` between the state `Disk Error` and the state `Install Software` is added.

If the transformation HSM2SM is applied on `SM'.xmi`, we expect changes to be propagated to the source model. However, target changes may be propagated in a

[6]The HSM2SM execution is described as a tutorial at http://jtl.di.univaq.it/.


Fig. 3.2 Forward execution of the HSM2SM transformation.

number of different ways, thus making the application of the reverse transformation to propose more solutions. The new transition can have each one of the 3 nested states within `Install Software` as targets, as well as the super state itself. Analogously, the same transition can have each one of the 4 nested states within `Disk Error` as sources, as well as the super state itself. The resulting alternative models are 20.

Four of the 20 generated sources, namely `HSM'_1/2/3/4`, can be inspected in Figure 3.3: the change (1) has been propagated renaming the state to `Start Install`; the change (2) has been propagated by creating the new transition, that has as target the nested states within `Install Software` and the super state itself (see the properties `HSM'_1/2/3/4` in Figure 3.3). For example, as visible in the properties of the transition, `HSM'_1` represents the case in which the transition targets the composite state `Install Software`. This restriction of the solution space was possible through the addition of an *a posteriori* ASP constraint. The pseudocode of the constraint that eliminate the

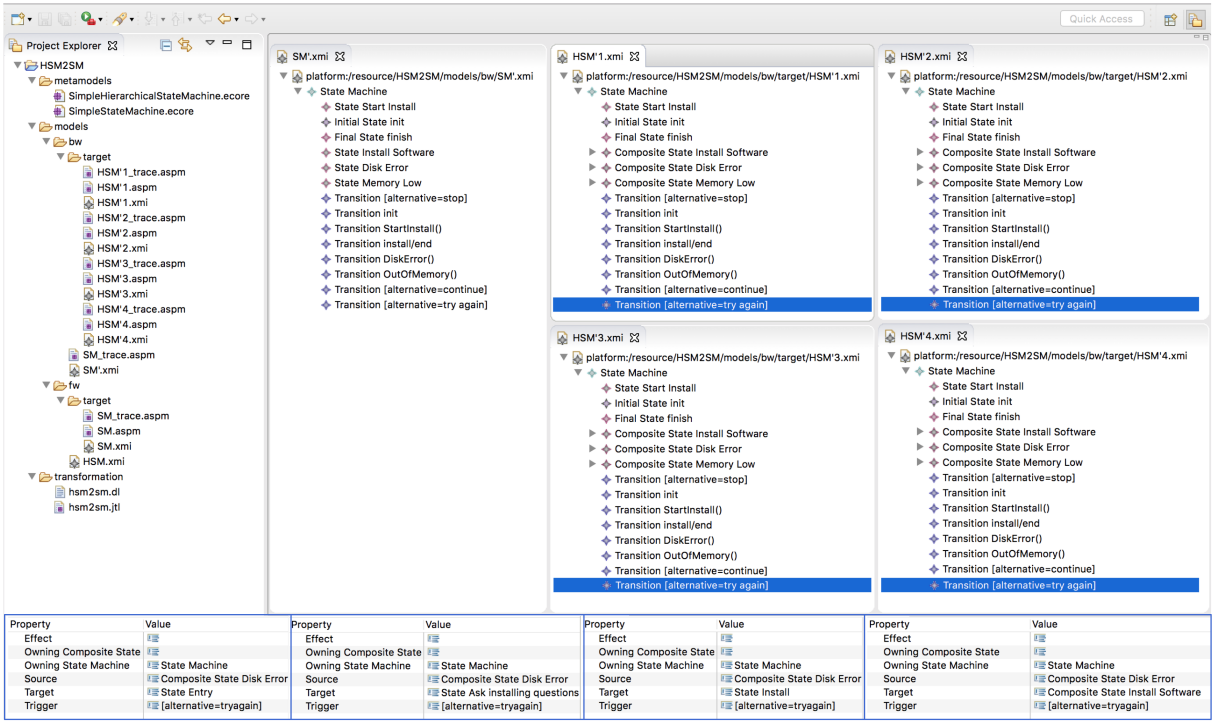



Fig. 3.3 Backward execution of the HSM2SM transformation.

transitions with source in the sub-states is the following:

$$
\begin{aligned}
:- & \mathit{sm.Transition}(T1).\mathit{source}(t1).\mathit{State}(S1),\\
& \mathit{hsm.Transition}(T1).\mathit{source}(t1).\mathit{State}(S2),\\
& \mathit{hsm.State}(S2).\mathit{owningCompositeState}.\mathit{CompositeState}(S1).
\end{aligned}
$$

It eliminates all the alternative models such that: a transition $T1$ that targets the state $S1$ exists in the *sm* domain, and also a transition $T1$ that targets the state $S2$ (that is a sub-state of $S1$) exists in the *hsm* domain.

If the transformation is applied on one of the derived `HSM'` models, the appropriate `SM'` models including all the changes are generated. However, this time the target will preserve information about the chosen `HSM'` source model, thus causing future applications of the backward transformation to generate only `HSM'`.

The specific role of traceability and how this was improved to deal with this kind of scenarios is discussed in details in the following sections.

## 3.3 Advanced traceability management

Current tools when dealing with non-injective mappings expose two different kinds of behaviour: either they make arbitrary choice of one consistent model based on heuristics, or they generate all consistent models and let the user pick one. In the latter case, invertibility might be jeopardised if the information about which model is singled out by the user is not made persistent while transforming it in both directions [127]. As to partial transformations, they typically do not cover all the (source) concepts with consequent information loss that may give place to unwanted behaviour when the transformation is reversed. As a matter of fact, practical model transformation engines frequently fail to restore consistency between models [116].

Complex round-tripping scenarios are very challenging and tools tend to respond with ad-hoc solutions compromising relevant properties, where better mechanisms to enforce *persistence* in bidirectional transformations are needed for storing important aspects about the transformation execution.

In previous works [36, 49], the traceability capabilities of JTL have been introduced as an implicit mechanism of the underlying engine. Such an implicit mechanism has been used to maintain a reference linkage between source and target elements and then discarded after the transformation execution. In the following sections, we present an advanced traceability approach aiming at: *i)* automatically storing expressive traceability information between source and target elements to prevent transformation *volatility* even in case of partial and non-injective mappings; *ii)* explicitly representing traceability as separate models; *iii)* reusing traceability in the transformation process to enable the invertibility of the transformation and provide support for managing models as well as their traceability information.

### 3.3.1 Motivating example

Recently, the need for testing and evaluating transformation tools led to the Transformation Tool Contest (TTC) workshop series. During the 2017 edition[7] (held within STAF Conferences[8]), three cases have been selected via single blind reviews in order to assess the expressiveness, the usability, and the performance of transformation tools. Among these, the Families to Persons Case [8] is especially interesting because of the challenges it might pose. The considered bidirectional scenario involves the metamodels Families and Persons (as described in [8]). A family register stores a collection of

[7] http://www.transformation-tool-contest.eu/2017/
[8] http://www.informatik.uni-marburg.de/staf2017/

families whose members are distinguished by their roles, i.e., mother, father, daughters, and sons. Besides that, a person register maintains a flat collection of persons who are either male or female. Families and persons models are kept in a consistent state by means of a bidirectional transformation (available in Appendix A), so that: i) mothers and daughters (fathers and sons) are mapped to females (males), and vice versa; and ii) the name of every person is composed of the family name and the member name. Furthermore, there may be multiple families with the same name and multiple persons with the same name and birthday.

The metamodels are clearly non-isomorphic since there are not one-to-one correspondences among the concepts in both metamodels; for instance, the birthday attribute in person does not have a counterpart in family members. Moreover, a person may correspond to a parent or a child, and persons may be grouped into families in different ways, which is clearly non-injective. As a consequence, depending on the transformation engine, models can be transformed in different ways, leading to unwanted behaviours over which designers may have little or no control, as shown in the rest of the section.

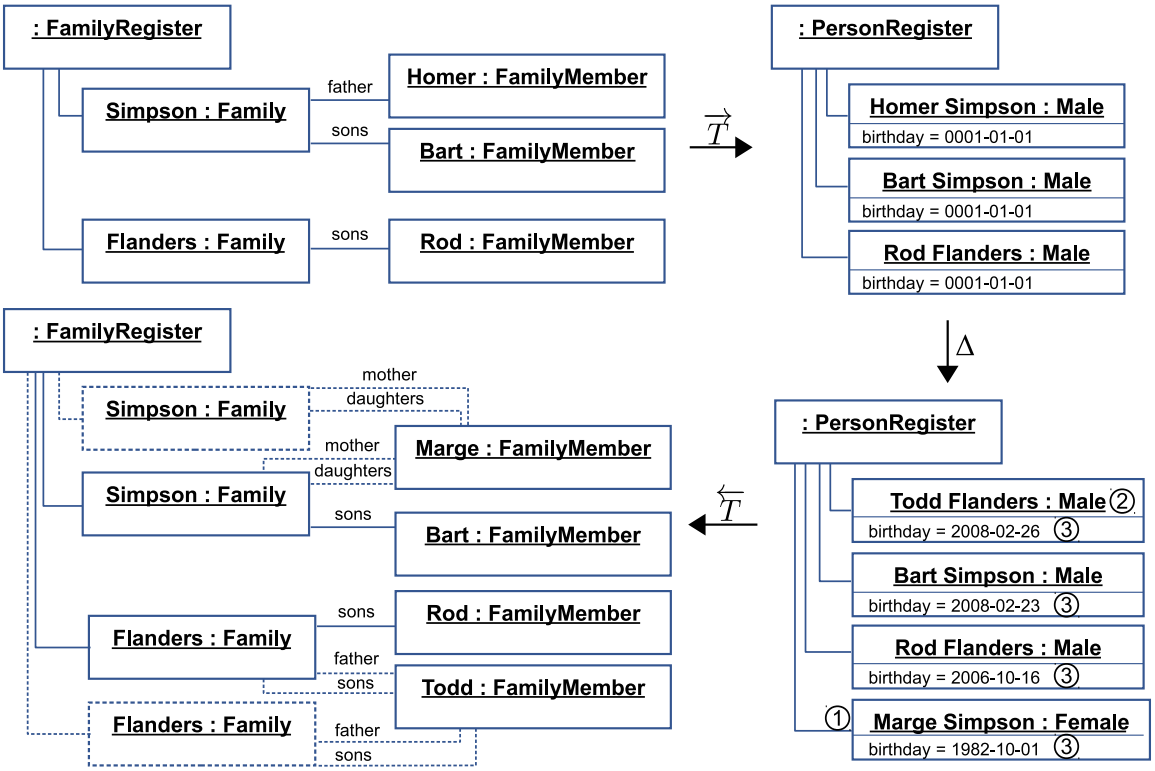



Fig. 3.4 Sample models for the Families to Persons case.

Figure 3.4 describes a round-trip scenario, where the `Families` model in the upper-left corner consists of the family `Simpson`, with a father `Homer` and a son `Bart`, and a family `Flanders`, with a son `Rod`. The corresponding `Persons` model is given in the upper-right corner: it consists of the corresponding male persons, i.e., `Homer Simpson`, `Bart Simpson` and `Rod Flanders`, whose `birthday`(s) are set to the metamodel default value `0001-01-01` as they do not have a counterpart in the source model. At this point, the `Persons` model is manually modified as follows (see the lower-right part of Figure 3.4): ① a new female `Marge Simpson` is added (initially its `birthday` is set to the default value); ② `Homer Simpson` is changed in `Todd Flanders`; ③ the `birthday`(s) of both `Flanders`' and `Simpson`'s are changed. Propagating such changes back to the source model is not univocal as more than one valid update policy is possible, as illustrated in the lower-left corner of the same figure where dashed lines denote alternative elements. While `Bart` and `Rod` are deterministically restored in their original families and role (`sons`), the change ① can be propagated in four different ways: `Marge` can be either `mother` or `daughters` in the existing `Simpson` family or in a new `Simpson` family. Due to the change ②, `Homer` is no longer a member of the `Simpson` family because his full name changed; thus, `Todd` can be equally mapped as a `father` or `sons` either in the existing or the new `Flanders` family. Finally, the change ③ can not be propagated to the source model since `birthday` is missing in the `Family` metamodel.

The non-injective and partial mappings described in the scenario are clearly challenging for most (if not all) existing transformation engines. In order to mitigate the problem of the uncertainty related to the above scenario, the TTC case authors have provided two configuration parameters controlling the backward transformation: one regulates whether a person is mapped to a parent or a child, while the other whether a person is added to an existing or a new family. However, the problem remains intrinsically difficult to solve and, despite the parameters, the execution of the TTC test cases on several tools highlighted some shortcomings as demonstrated by executing the TTC test suite on Medini QVT[9] and eMoflon[10]. Both tools make use of some traceability management to enable incremental updates controlled by the configuration parameters. While eMoflon correctly restores already existing elements, Medini is not able to restore the `Bart` and `Rod` original roles and families (see Figure 3.4). According to the configuration parameters settings `PREFER_CREATING_PARENT_TO_CHILD = true` and `PREFER_EXISTING_FAMILY_TO_NEW = true`, in eMoflon the new female `Marge` is added as `mother` in the existing `Simpson` family; while in Medini `Marge` is wrongly added in a

[9] http://projects.ikv.de/qvt
[10] http://www.moflon.org

new Family. As to the renaming (cfr. ②), Medini again exposes difficulties in restoring existing roles, whereas eMoflon is not able to configure the preference when both family and role changed. Finally, both approaches are *forgetful* when dealing with the modified birthdays in the target domain (cfr. ③) failing in re-establishing them during a forward execution. While placing reliance on tracing information is well-accepted in order to support round-tripping processes, the mechanisms adopted by these tools seem not accurate and do not help in achieving relevant transformation properties, such as correctness (in the sense of [114]), which is - at the end of the day - the reason because they are adopted. It is also worth noticing that the configuration parameters as a mean for controlling transformation behaviour seem problematic because they are subjectively decided and not part of the transformation specification.

Despite the described scenario is not complex, it demonstrates that transformation tools are far from being reliable up to the point that new research directions (e.g., [116]) are exploring sub-optimal characterisations to cope with tool difficulties. In particular, invertibility in bidirectional transformations can be supported by managing traceability information of the transformation executions, which can be exploited later on to trace back the target models. In fact, each generated element can be linked with the corresponding source and contribute to the resolution of some problem. For instance, in Figure 3.4 the correspondence between a person and its source role and family is necessary to restore Bart and Rod in their original families as sons. However, no trace links can be maintained for the new elements that cause the generation of multiple alternatives. Finally, information not considered by the partial mapping should be stored so that is not lost in the backward transformation. For instance, the updated birthday dates should be stored in order to be mapped in next alignments. Supporting traceability allows transformations to be persistent and overcome these difficulties.

### 3.3.2 A metamodel for traceability

In this section, we introduce a metamodel to specify links between model elements by using expressive traceability relationships for enabling a correct behaviour in bidirectional transformation executions.

The Trace Metamodel is depicted in Figure 3.5. For each transformation execution (both in forward and backward directions), a trace model (`TraceModel`) is generated; it relates a model belonging to a *left domain* to a model belonging to a *right domain*.

The metamodel supports the definition of a set of trace links between left and right elements and the transformation rule that enforced their mapping. A trace link is characterized by an `id`, a rule name (`trule`), a `name` (i.e., concatenation of `trule`

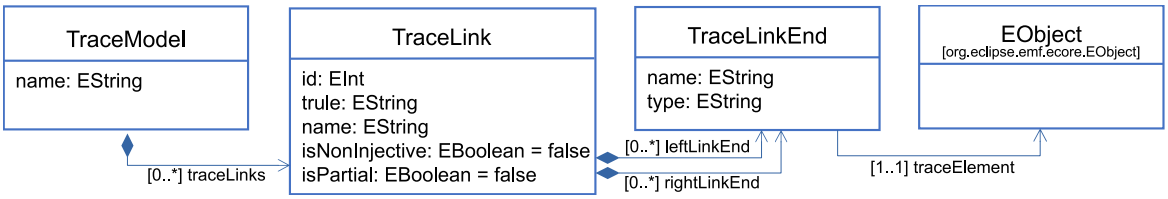



Fig. 3.5 The Trace Metamodel.

and `id`), a boolean `isNonInjective` (that is true in the case of non-injective traces) and a boolean `isPartial` (that is true in the case of partial traces). Furthermore, it relates one or more elements belonging to the left domain (`leftLinkEnd`) and the correspondent one or more elements belonging to the right domain (`rightLinkEnd`). Such references target elements of type `TraceLinkEnd`, that have a `name` and a `type`. Each `TraceLinkEnd` represents a specific object in the left or right domain (`EObject`).

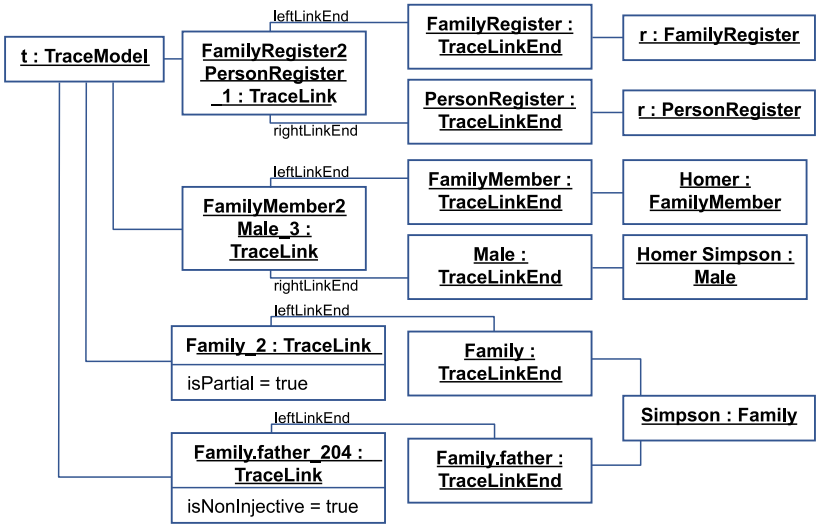



Fig. 3.6 A sample of trace model.

A sample of trace model is depicted in Figure 3.6. It represents a fragment of the trace model generated by the forward execution of the Families to Person case as described in Figure 3.4. The model in figure contains the following trace links:

- `FamilyRegister2PersonRegister_1` relates objects `:FamilyRegister` and `:PersonRegister` via the rule named `FamilyRegister2PersonRegister`
- `FamilyMember2Male_3` relates the object `Homer` of type `FamilyMember` to the object `Homer` of type `Male` via the rule named `FamilyMember2Male`

- `Family_2` is a partial link that stores the object `Simpson` of type `Family` that does not have a counterpart in the Persons domain
- `Family.father_204` is a non-injective link that stores the reference `father` of the object `Simpson` of type `Family` that is involved in a non-injective mapping.

This representation is used within the JTL framework to keep tracing information during the execution of transformations.

### 3.3.3 Including traceability in the bidirectional transformation process

The overall architecture of the JTL environment, upgraded with traceability capability, is reported in Figure 3.7. The engine executes JTL transformations which have been written in a QVT-like syntax, and then automatically transformed into ASP programs. Source and target metamodels of the considered transformation ($MM_1$, $MM_2$) are also encoded in ASP and managed by the engine during the transformation process. For each execution, a trace model relating source and target model elements is generated and optionally given as additional input of the next execution ($T_{M1M2}$, $T_{M2M1}$). When a trace model is provided as additional input of a transformation, the execution will automatically consider traceability information during the mapping process. Trace models conforming to the metamodel $TTM$ are automatically mapped in/from ASP.

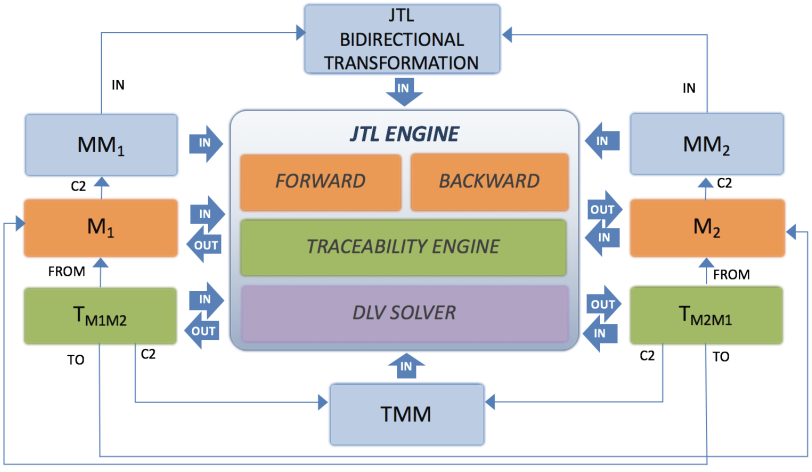



Fig. 3.7 Overview of the JTL framework with traceability.

The traceability management process is composed of the following phases:

- *Automatic generation:* The traceability mechanism exploits the intrinsic characteristic of the ASP-based engine to derive relevant details about the linkage between source and target model elements at execution-time (including the applied transformation rules). This step relies on the *Traceability Engine*;
- *Explicit representation and storing:* Trace links are extrapolated during the transformation execution and made explicit by the framework. Meaning that trace models are maintained within EMF as models conforming to the Trace Metamodel and serialized in XMI;
- *Re-use:* Trace models can be re-used during the transformation execution. In particular, a trace model can be given as input of the transformation execution in order to (re-)establish consistency, manage ambiguities and guarantee the correctness of the transformation. In particular, the *Traceability Engine* is able to include traceability information in the execution process.

In the next section, the JTL traceability management is shown in practice by applying it on the *Families to Persons* benchmark presented in Section 3.3.1.

### 3.3.4 Round-trip on the Families to Persons benchmark

According to the scenario described in Section 3.3.1, the input model conforming to the *Families* metamodel is reported in its Ecore representation in the left-hand side of Figure 3.8. The application of the JTL *Families2Persons* bidirectional transformation[11] on `sampleFamily` generates the corresponding `samplePerson` model, depicted in the right-hand side of Figure 3.8. We reported the generated trace model `sampleFamily2samplePerson` in the middle of the same figure, where dashed arrows connect trace elements to the elements they refer to in source and target models.

The generated trace model is composed of:

a) trace links `FamilyRegister2PersonRegister_1`, `FamilyMember2Male_3`, `FamilyMember2Male_4`, and `FamilyMember2Male_6`, that store how family elements (left `TraceLinkEnd`s) have been mapped to person elements (right `TraceLinkEnd`s);

b) partial trace links `Family_families_102`, `Family_families_104`, `Family_2`, `Family_5`, that represent elements or structural features not covered by the transformation;

c) non-injective trace links `FamilyMember_father_204`, `FamilyMember_sons_206`, `FamilyMember_sons_504` that store elements involved in non-injective mappings.

[11] The interested reader can access the implementation at http://jtl.di.univaq.it/

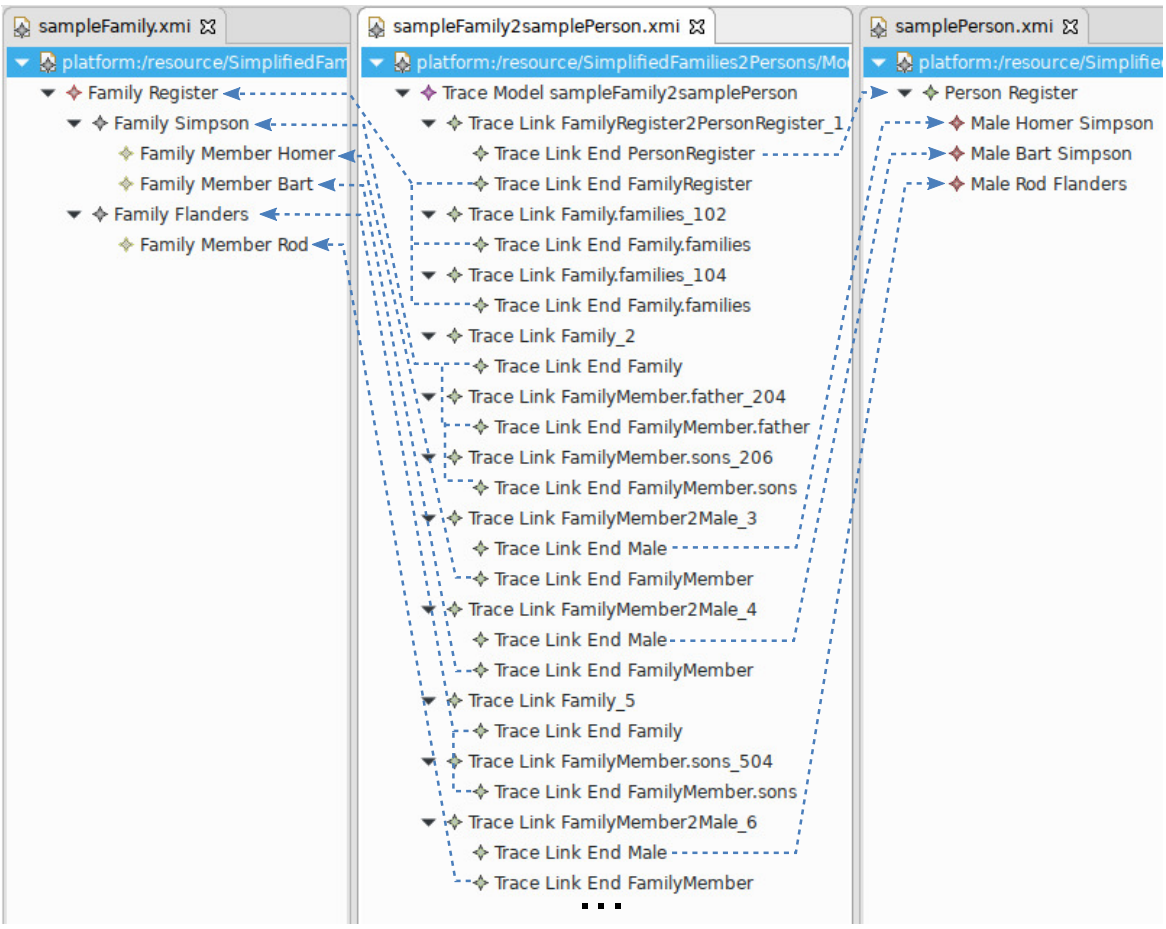



Fig. 3.8 The forward execution of the Families to Persons case.

Note that, by re-applying the transformation in the backward direction it is possible to obtain again the `sampleFamily` source model. The missing information about the original families and role of the members are restored by means of trace information by providing as additional input the trace model `sampleFamily2samplePerson` obtained in the previous forward execution.

After a refinement step, the target model including the changes described in Section 3.3.1 is shown in the left part of Figure 3.9. The modified model `samplePerson'` and the trace model `sampleFamily2samplePerson` (obtained in the previous execution) are given as input of the backward transformation execution. Due to the non-injectivity of the transformation, 16 alternative models, namely `sampleFamily'{1,2,..,16}` are generated. All the alternative models correctly restore families and roles of the members as in the original source model. In particular, the new female Marge Simpson can be propagated both as a mother or a daughter in the existing Simpson family as well as in a new one. Analogously, the changed member `Todd Flanders` can be propagated in

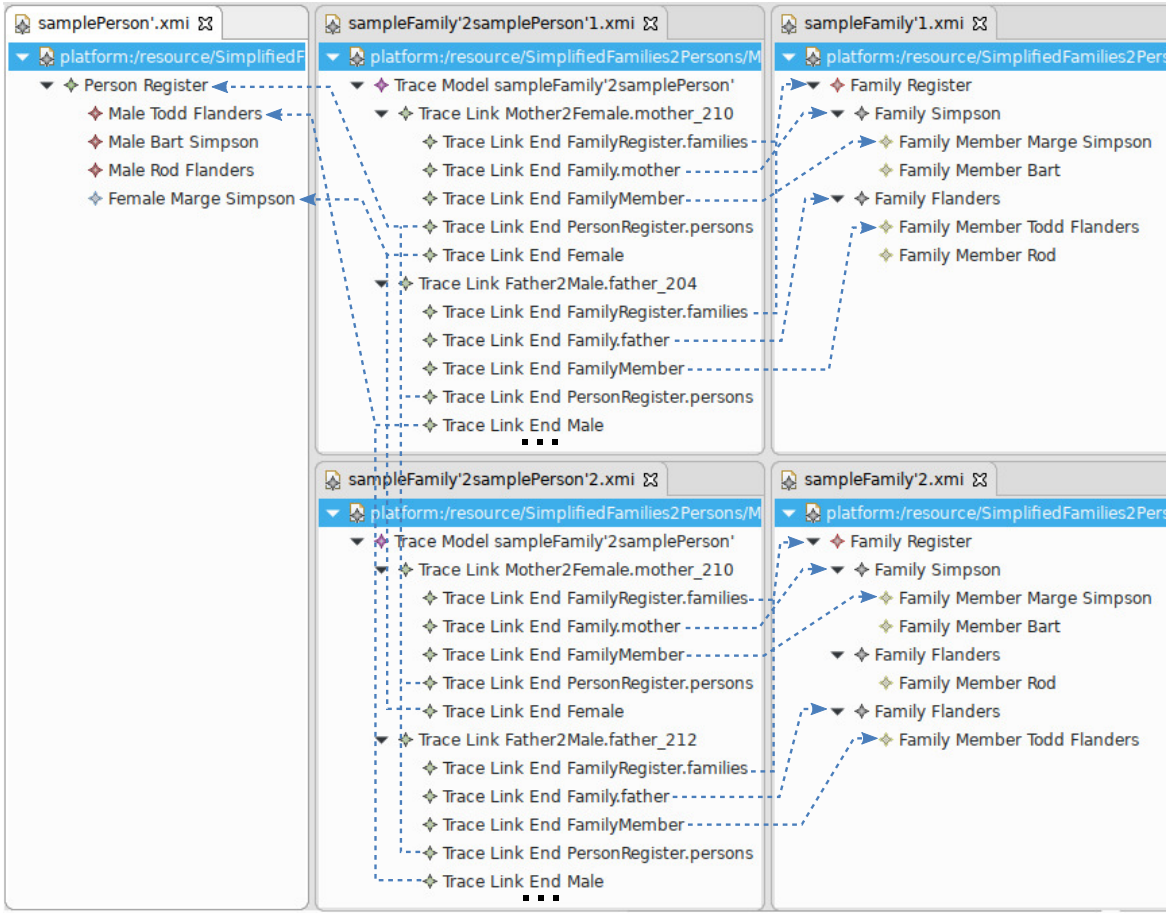



Fig. 3.9 The backward execution of the Families to Persons case.

four different ways, by adding Todd to the existing or new family either as a father or as a son. Finally, for each generated model, a correspondent trace model is generated.

Figure 3.9 (middle and right-hand side) shows two of the 16 generated models along with a fragment of their traces. In both family models, Marge Simpson is created as mother of the existing Simpson family, as stored in the trace link `Mother2Female.mother_210` in both the trace models. Moreover, in the model `sampleFamily'1` Todd Flanders is created as father of the existing Flanders family, while in the model `sampleFamily'2` he is created as father of a new family, as stored in the trace links `Father2Male.father_204` and `Father2Male.father_212` in the first and second trace models, respectively. Note that, by selecting one of the alternative models and re-applying the transformation in the forward direction it is possible to obtain again the `samplePerson'` model. The missing information about the birthday attributes are restored by means of trace information, that is giving as input the trace model `sampleFamily'12samplePerson'` obtained in the previous forward execution.

## 3.4 Related work

**Bidirectional transformations.** Over the last decade, a number of bidirectional approaches and tools have been proposed. Concerning the multiplicity of solutions, most of the existing languages are deterministic, i.e., they produce one model at time.

The QVT-R bidirectional transformation language [114] does not support non-bijective transformations. A formal discussion about non-deterministic transformations is provided by Abou-Saleh et al. [1]. Despite the heavy emphasis placed on model transformation by the OMG's, tool support for bidirectional transformations expressed in QVT-R remains limited [115, 86]. Medini QVT is an Eclipse plugin for a subset of the QVT-R language. Its semantics admittedly disregards the semantics from the QVT standard (it does not have a checkonly mode for instance).

Within the available tools for Triple Graph Grammars (TGGs), eMoflon received wide acceptance and it is one of the most prominent TGG platforms. It features both batch and incremental model transformations [80]. Moreover, eMoflon proposes to manage non-determinism by allowing designers to make decisions as early. In particular, it provides a Java-based API via which the designer can choose which matches she prefers/prioritises as soon as there are multiple choices available in the transformation process.

The BiFluX language [129] represents an attempt to make bidirectional transformation deterministic though intentional updates. However, the problem that a transformation cannot be tested for non-determinism at static-time reduces its effectiveness. Recently, BiGUL [73] has been proposed as a revision of the core of BiFlux, designed to be closer to practical programming languages. GRoundTram (Graph Roundtrip Transformation for Models) [64] is an integrated framework for developing well-behaved bidirectional model transformations based on the functional programming language OCaml. It is a compositional, functional and algebraic approach based on graph algebra and structural recursion.

Similarly to JTL, Macedo and Cunha [86] propose an approach able to enumerate the possible solutions of a non-deterministic specification. The proposed QVT-R tool is based on a SAT solver (Alloy). In contrast with JTL, it supports metamodels enriched with OCL constraints to limit the possible edits and control non-determinism. Moreover, the enforcement semantics is based on the principle of least change. Tool support is available, but development seems to be discontinued.

**Traceability for model transformations.** The technological importance of traceability is well recognized in model transformations. Most existing transformation

approaches adopt an internal traceability model that can be inspected at execution time; for instance, to resolve dependencies between the rules or to check if a target elements was already created for a given source element. Widely used unidirectional languages, like ATL [67] and ETL [75], automatically create traceability links during the transformation execution. The traceability approach is specific to each transformation language. Both in ATL and ETL, relations have to be bijective and traceability is used to resolve dependencies between the rules.

In bidirectional transformations, the state may be stored in explicit or implicit way. Boronat et al. [26] propose an approach to traceability within MOMENT, a model management framework, where model operators are defined independently of metamodels. In this case, traceability is employed as a mechanism to follow transformations refinement steps. Among the proposed operators, the Match operator permits the inference of traceability links between two models. The same authors, also propose ModelGen [27], an operator that uses the term rewriting system Maude as transformation engine and provides support for traceability. ModelGen also provides support for QVT-R in an EMF environment. Medini QVT uses a trace model to facilitate model transformations and conformance checking, and to additionally enable incremental updates during further executions. The traceability metamodel is deduced from the transformation specification. Even if in QVT-R relations may not be bijective, only bijective relations are addressed in Medini QVT. Thus, trace data are not used for the resolution of ambiguities. In TGG [111], the correspondence graph ensures traceability between source and target elements, and traceability can be used to check the correspondence between two models and synchronize them incrementally. eMoflon provides a Java-based API via which the designer can choose the priority matches as soon as there are multiple choices available in the transformation process. Tracing information are not used to support partial and non-injective cases.

Traceability is an intrinsic property in lens-based approaches [53, 64, 99]. In fact, each element of the abstract structure is related with a correspondent element in the concrete structure. In particular, the put function updates the concrete structures restoring any information discarded during the forward transformation. When a change is performed, the transformation engine propagates the change towards the other direction. In Boomerang[12] and BiFlux[13], traceability information is not explicitly visualizable or stored in any file. In GRoundTram[14], trace information can be generated

[12] http://www.seas.upenn.edu/harmony/
[13] http://www.prg.nii.ac.jp/projects/BiFluX/
[14] http://www.biglab.org/

during the transformation execution as a graph or textual file. However, in such functional approaches no support to partial and non-injective transformations is given.

# Chapter 4

# Software Availability Assessment and Improvement

In this chapter we present a model-driven approach that works on the forward and backward path of a round-trip software process to support designers in improving the availability of their software architecture. In particular, we introduce JASA (JTL-based framework for Availability analysis of Software Architecture), which makes use of bidirectional model transformations to map architectural models and availability models in both forward and backward directions. By working with UML models annotated with availability parameters and Generalized Stochastic Petri Nets (GSPN), JASA is able both to derive an analysis model from a software architecture and, after the analysis, to propagate back on the software architecture the changes made on the analysis model. In addition, these changes can be based on well-known fault tolerance patterns that we have preventively modeled in GSPN to be easily applied to the model under analysis.

In a previous work [37], we presented a bidirectional model transformation between UML State Machines (SMs), annotated with availability parameters, and GSPN. Such transformation was aimed both to derive a GSPN availability model from a SM-based software architecture and, after the analysis, to propagate back on the UML model the changes made on the GSPN model. This chapter is an extension of our previous paper stemming from the realization that only considering SMs was restricting. In fact, a deeper semantic comprehension of an UML model can be achieved if the dynamic behavior is modeled by using the Sequence Diagrams (SDs) in addition to SMs [98]. Additionally, SMs describe the behavior of an object (that could be the instance of a particular component/class) depending on what state it is currently in, whereas SDs show the execution of use cases and the behavior of involved objects in

terms of their interactions. Such modeling extension of behavioral aspects of software architectures impacts on the accuracy of availability analysis and on the introduction of well-known fault-tolerance refactoring techniques (e.g. error masking). Moreover, the back propagation of GSPN changes into UML models is improved by considering the interactions among components.

In this chapter, we also introduce a catalog of refactoring patterns with the aim to drive the designers in their process. In particular, the patterns for fault tolerance presented in [109] have been considered to generate the corresponding patterns in GSPN, that will be propagated in UML through the bidirectional model transformations defined in JASA. The overall approach has been realized as a dedicated framework implemented within Eclipse.

Finally, our approach has been evaluated on an Environmental Control System example application in order to address these points:

i) generation of analyzable availability models from software architecture models;

ii) back generation of valid software architecture models from availability models;

iii) ability to improve the availability of software architecture models.

## 4.1 Background

In the following, we describe the background of this research work and its contributions in terms of non-functional analysis and refactoring process leveraged for the definition of JASA.

### 4.1.1 Round-trip non-functional analysis process

In order to validate non-functional requirements on a software architecture, some approaches, mostly based on model transformations, have been proposed in the last decades to generate *non-functional models* from *software architectural descriptions* [39, 23]. This generation step is also called *forward path*, and it is represented by the topmost steps of Figure 4.1. However, the solution of generated models does not necessarily produce indices that satisfy the requirements, thus an iterative process is often required to refactor the generated model on the basis of solution results. This process (hopefully) ends up when satisfactory indices are produced, and it is represented by the rightmost step of Figure 4.1.

Thereafter, changes applied to non-functional models, for the sake of requirement satisfaction, have to be propagated back to the software architecture, and this is represented by the bottom-most step of Figure 4.1, also called *backward path*. However, analysis results do not straightforwardly suggest what changes have to be made on the software architecture, hence this propagation is often based on the ability of experts that interpret the results. This clear lack of automation in the backward path represents a heavy limitation towards the construction of a round-trip process for non-functional validation of a software architecture. In this chapter, we consider this general round-trip non-functional analysis process in the availability analysis context.

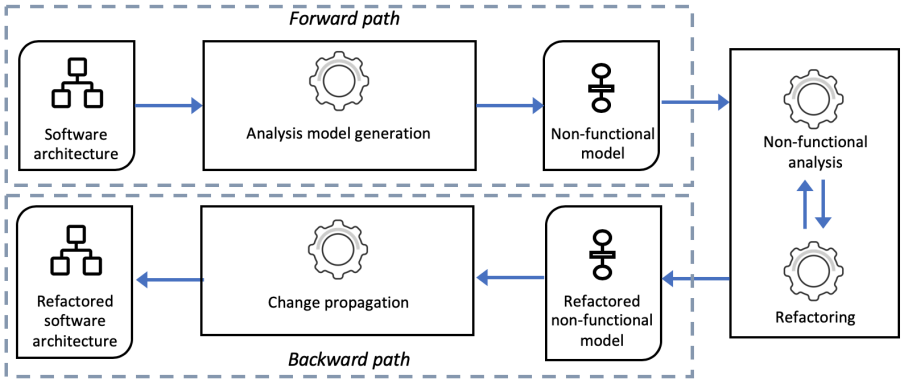



Fig. 4.1 Round-trip non-functional analysis process.

### 4.1.2 Model-based availability analysis

Availability can be defined as the system readiness to provide correct service. It corresponds to the probability that the system is working within its specifications at a given instant [14]. In particular, the steady state availability can be expressed as the ratio between the value of MTTF (Mean Time To Failure) and the sum of MTTF and MTTR (Mean Time To Repair) values.

Stochastic Petri Nets (SPN) are a well-established formalism for modeling systems availability [23]. In this chapter, we consider an extension of SPN, called Generalized Stochastic Petri Nets (GSPN) [87]. Transitions defined in GSPN can be either immediate, when firings take no time, or timed, when associated delays are exponentially distributed. Immediate transitions fire with priority over timed transitions, and different priority levels can be defined over them. A weight is also associated to each immediate transition. When two or more immediate transitions are in conflict (e.g., because they have the same priority), the selection of the one that fires first is made using the associated weights. The delay associated with a timed transition is a random

variable, distributed as a negative exponential, with a defined rate. When two or more timed transitions are in conflict, the selection of the one that fires first is made according to the race policy.

In this chapter, availability analysis is conducted on a GSPN derived from a software architecture modeled in UML [97]. Since UML does not natively provide support for availability modeling, we rely on the "Dependability Analysis and Modeling" (DAM) profile [22] to enhance UML models with availability annotations. DAM was designed on top of the standard MARTE profile [96], which extends UML to annotate models with schedulability and performance analysis information. Despite the ability to annotate behavioral models with availability properties, UML-DAM lacks the execution semantics to be formally analyzed. This is the reason why DAM-annotated UML models need to be transformed (e.g., in GSPN) for the sake of analysis.

### 4.1.3 Fault tolerance refactoring techniques

Nowadays, software has strong influence on system availability. Since defects inherently occur in software design and coding for several reasons (e.g., software complexity, changing requirements, time pressures), software fault tolerance is even more important.

Among the well-known fault tolerance refactoring techniques that may improve the software availability, we consider the techniques that deal with error masking [109], i.e.: *Passive Replication*, *Semi-Passive Replication*, *Active Replication* and *Semi-Active Replication*.

Error masking techniques aim at isolating the subsystem in which an error is detected by relying on some form of redundancy to resume the processing that the system was performing when the error occurred. Replicas of system components and checkpoints can be employed, even in combination, to implement such techniques. *Passive Replication* and *Semi-Passive Replication* patterns provide error masking by saving the state of a component (checkpoint) before it receives the input, so that, if an error occurs during processing, an identical replica of the component can be activated to restart the processing from the saved checkpoint. While in the *Passive Replication* pattern the checkpoint is stored in a separate *Storage* component, *Semi-Passive Replication* requires that the checkpoint is directly stored in the replica. On the other hand, *Active Replication* and *Semi-Active Replication* patterns require a group of replicas to be always active during input processing. In the *Active Replication* pattern, the replicas provide the output to a *Comparator* component that performs a majority vote before forwarding it to the rest of the system. In contrast, in the *Semi-Active Replication* pattern, a replica provides the output to the system only when

an error occurs in the original component. The patterns mentioned above, as well as refactoring inspired by them, will be presented in detail in Section 4.2.2.

## 4.2 Approach

In this section we introduce JASA, a model-driven framework for supporting the round-trip availability analysis process and software architectural refactoring. The approach aims at supporting designers in their availability analysis process that involves the back propagation of results as refactoring actions on the software architecture. In particular, JASA leverages the interplay of UML and GSPN and provides automation for their mapping by means of a bidirectional model transformation mechanism. In addition, JASA provides a set of refactoring actions that can be used by the designer to improve the availability of the system.

In the following, we introduce the used technologies, we present the process underlying our approach, and we provide a catalog of availability patterns that can be applied on GSPN models. Then, we describe the implementation of the approach based on bidirectional model transformations. The complete implementation of JASA is available online[1].

### 4.2.1 From availability assessment to architecture improvements

JASA has been implemented on top of JTL and mainly exploits the Eclipse Modeling Framework (EMF). As a consequence, the environment supports any language defined as a metamodel conforming to Ecore (i.e., the EMF metamodel). In this chapter, we focus on GSPN-based analysis models, whereas, the software architecture is modeled by means of UML. In particular, for the behavioral aspects, State Machines (SM) and Sequence Diagrams (SD) annotated via DAM are considered, whereas for the static aspects, Component Diagrams (CD) are considered. In the rest of the chapter, we use $\text{UML}^{\text{JASA}}$ to refer to the considered UML diagrams, that are $\text{UML}^{\text{SM}}$, $\text{UML}^{\text{SD}}$ and $\text{UML}^{\text{CD}}$.

The JASA overall approach is reported in Figure 4.2. As said, the *Bidirectional engine* relies on JTL to enable the execution of bidirectional model transformations in both forward and backward directions. The $\text{UML}^{\text{JASA}}$-GSPN bidirectional transformation maps UML models to GSPN and vice versa. In particular, in order to execute the

[1] https://github.com/SEALABQualityGroup/JASA

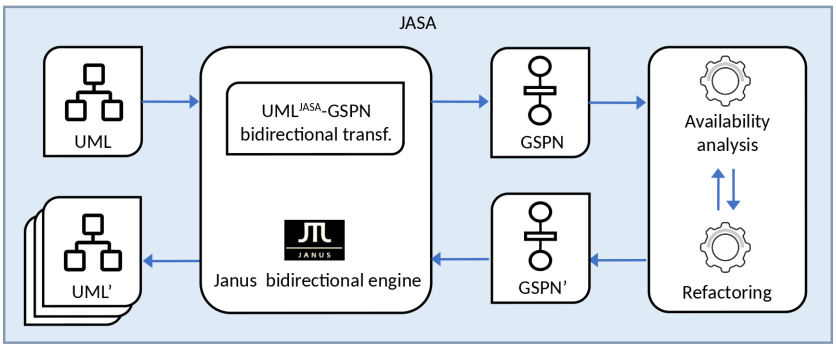



Fig. 4.2 The JASA overall approach.

transformation in the forward direction, a DAM-annotated UML model is taken as input to the engine, and the correspondent GSPN model is produced as output. The generated GSPN model is solved in order to obtain a set of indices that have to be interpreted (see *Availability analysis* in the figure). Thereafter, the GSPN model is iteratively modified until availability requirements are satisfied (see *Refactoring* in the figure). In order to propagate changes applied to the GSPN model back to the UML model, the bidirectional transformation is executed in the backward direction. In our case, the engine takes as input the modified GSPN model and produces as output a DAM-annotated UML model representing the software architecture that embeds the changes made on the GSPN to solve arisen availability problems.

### 4.2.2 A catalog of availability patterns

In this section, we present a set of patterns that can be used to improve the availability of a system. These patterns employ error masking techniques, based on replicas and checkpoints, that can be applied to a system designed by means of a GSPN. For each pattern, we show how a GSPN can be refactored to mask errors coming from a component without altering its original functionality.

Although, at the current stage, the refactoring activity is performed manually, the GSPN refactoring patterns we introduce in this section are designed to support automation. In particular, each pattern is equipped with anchor points that are used to properly insert it on a specific point of a GSPN modeling the original behaviour of a software component. Potentially, an automated tool can take as inputs the original GSPN, the pattern to be inserted and the specific point where it has to be applied, and it returns the GSPN refactored with the pattern.

Once applied on the GSPN model, the refactoring patterns will be propagated backwards through the transformation that will be presented in Section 4.2.3.

### Passive Replication

In this pattern, an error is masked by saving the state of a system component (a checkpoint) before it starts processing the input. If an error is detected, a backup replica of the same component is activated and the checkpoint restored. Hence, the backup will restart the processing of the input from the last state in which the system was behaving correctly. A Component Diagram of this pattern is reported in Figure 4.3a, where components represent roles in the pattern and interfaces are used to depict the actions that are required for coordination. Figure 4.3b shows the implementation of this pattern in GSPN. For the sake of presentation, three vertical dots are used to visually compress sequences of places and transitions without branching points, whereas surrounding boxes are used to highlight the roles of GSPN subnets in the pattern. Grey boxes (e.g., *Primary Behavior* and *Backup Behavior* in the figure) are introduced to show where the original behavior of components will fit into the pattern.

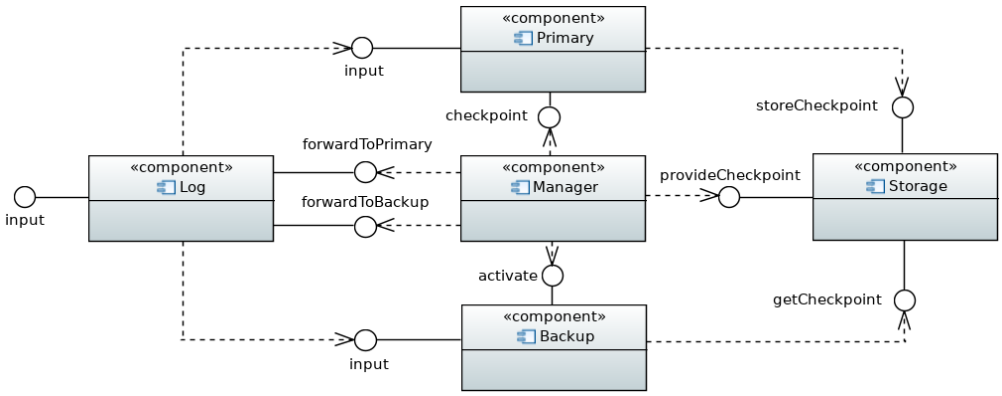
In order to implement the Passive Replication pattern, the following additional components should be added to the system:

- The *Backup*, that is identical to the original (*Primary*) component of which we want to mask errors. This replica is not started during error-free executions;
- The *Log*, which is able to record and forward inputs to the *Primary* as well as send the recorded inputs again to the *Backup* in case of error;
- The *Storage*, that is responsible for storing checkpoints and sending them to *Backup* upon request from the *Manager*. We assume that the *Storage* is not subject to errors;
- The *Manager*, that has the tasks of (i) asking the *Primary* to save a checkpoint, (ii) activating the backup in the presence of errors and (iii) requesting from the *Storage* to provide the last saved checkpoint.

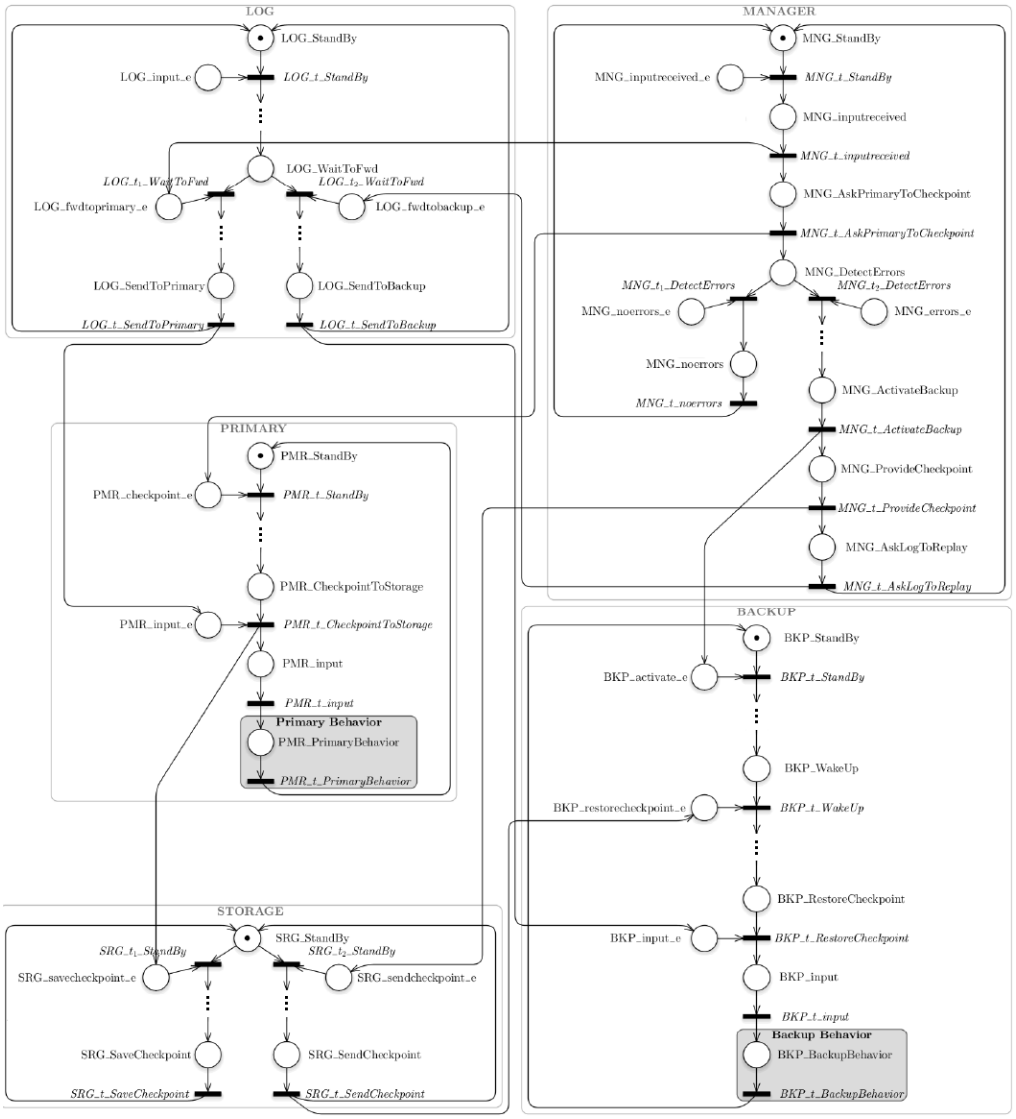
*Primary*, *Backup*, and *Manager* must be deployed to different units of failure.

### Semi-Passive Replication

The *Semi-Passive Replication* pattern is able to mask errors in a similar way to the *Passive Replication* pattern, but without requiring a storage dedicated to checkpoints. The Component Diagram of this pattern is shown in Figure 4.4a. The *Primary* component saves the checkpoint by sending it directly to the *Backup*. *Log* and *Manager* components are still required to implement the pattern. The *Log* stores and forward




(a) Component Diagram of the *Passive Replication* pattern.




(b) GSPN of the *Passive Replication* pattern.

Fig. 4.3 The *Passive Replication* pattern.

the input to the *Primary* which, before processing it, sends a checkpoint to the *Backup*. When an error occurs, the *Manager* activates the *Backup* and asks the *Log* to forward the input to it. The *Backup* restores its state using the saved checkpoint before starting to process the input. *Primary* and *Backup* must be deployed to different units of failure.

### Active Replication

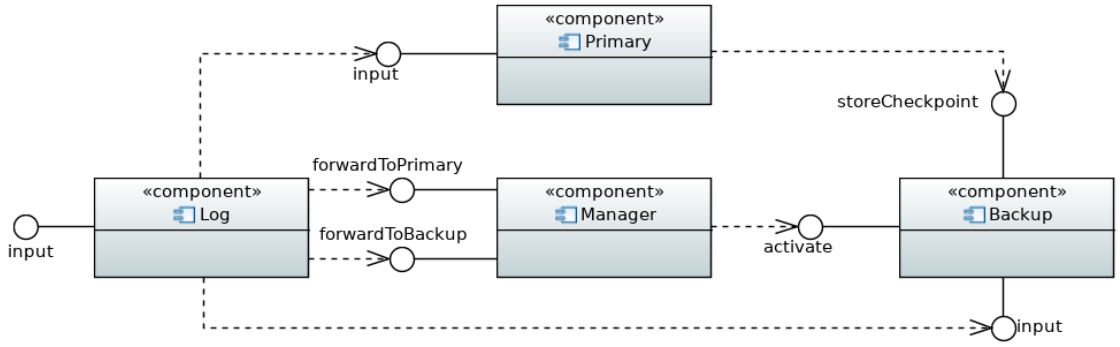
The *Active Replication* pattern is considered the most effective error masking technique but also the most expensive. This pattern employs a group of replicas actively receiving and processing every input intended for the component of which we want to mask errors. According to this pattern, we need to introduce two new components:

- The *Distributor*, which receives the input intended for the original component and forward it to all the replicas in the group;
- The *Comparator*, that is responsible for comparing the output computed by the replicas and deciding what will be the final output of the system. Since, in this case, the goal is to tolerate only crash failures, the *Comparator* passes the first response to arrive from the replicas and discards the rest.

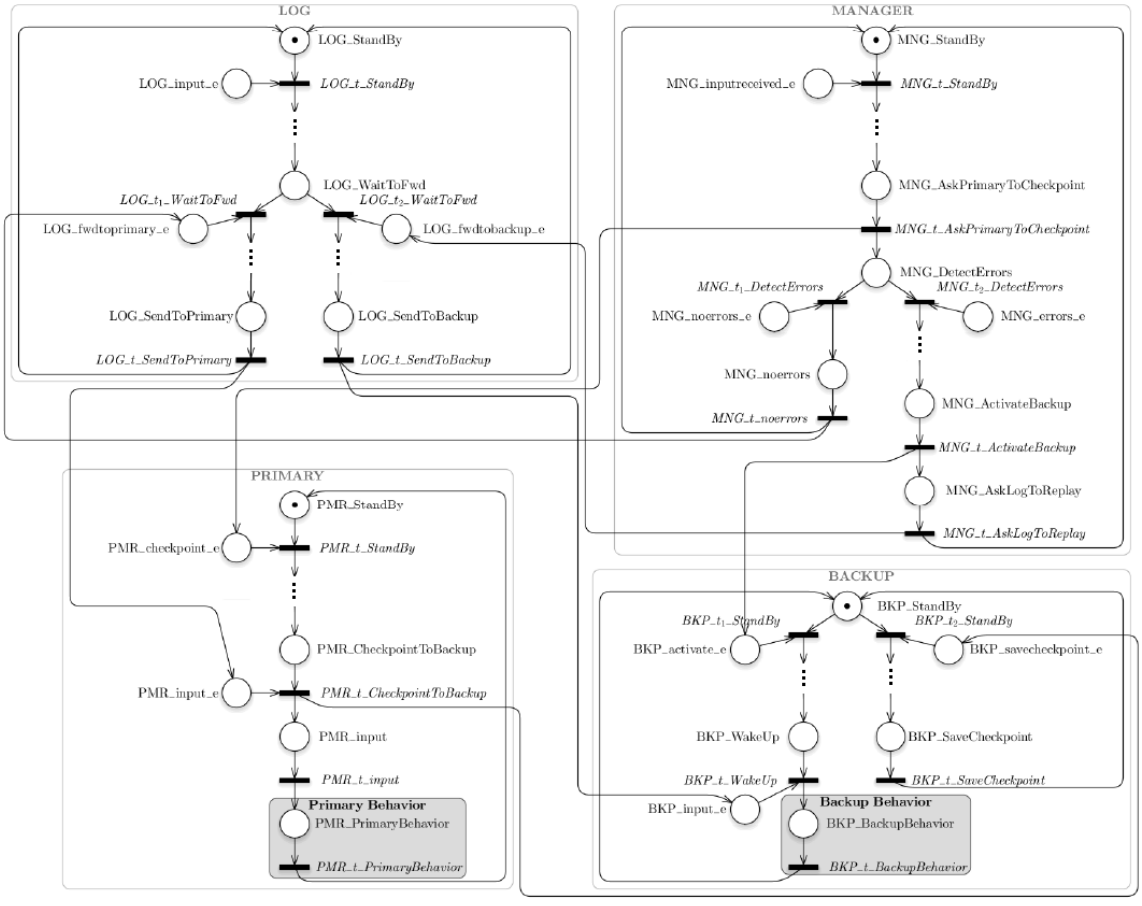
Figure 4.5a shows the Component Diagram of this pattern when the group of replicas is composed by (i) a *Primary* component, representing the original component of which we want to mask errors, and (ii) a *Backup* component, that is an identical replica of *Primary*. For the reason that all the replicas are continuously active, the structure of this pattern does not contain a different path that the system will follow in case of errors. *Primary*, *Backup*, and *Comparator* components must be mapped to different units of failure.

### Semi-Active Replication

Similarly to the previous pattern, the *Semi-Active Replication* pattern employs a group of replicas that are always active. However, unlike the *Active Replication* pattern, only one replica will deliver the output. Figure 4.6a reports the Component Diagram of this pattern with the group of replicas composed by *Primary* and *Backup*. A *Distributor* component is still needed to forward the input to all the replicas. In an error-free execution, the *Primary* component directly delivers the output to the environment and then reports to the *Backup* that no errors occurred so that the output computed by the *Backup* can be discarded. If an error occurs on the *Primary*, then the *Backup*

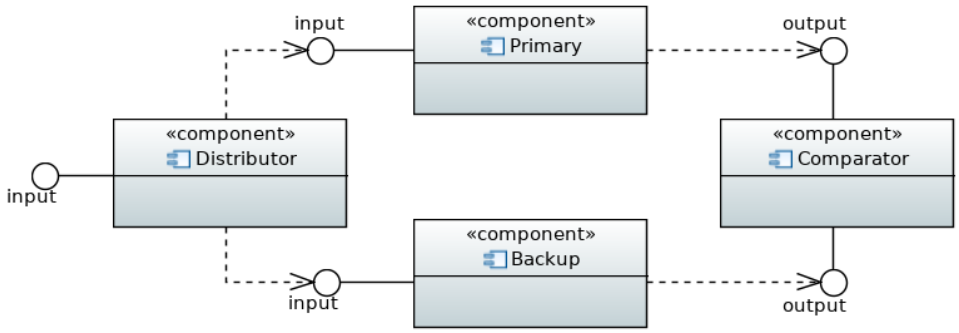



(a) Component Diagram of the *Semi-Passive Replication* pattern.

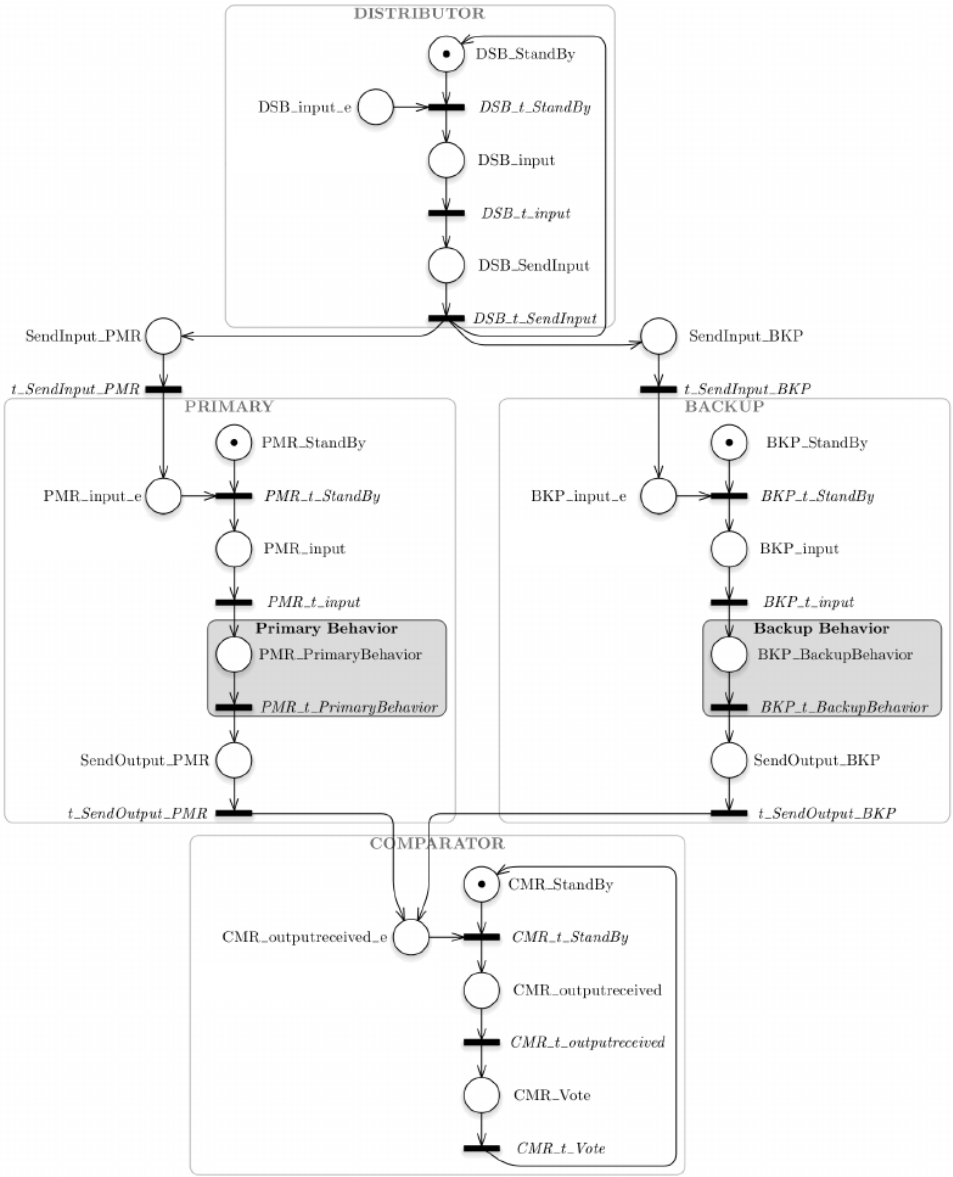



(b) GSPN of the *Semi-Passive Replication* pattern.

Fig. 4.4 The *Semi-Passive Replication* pattern.

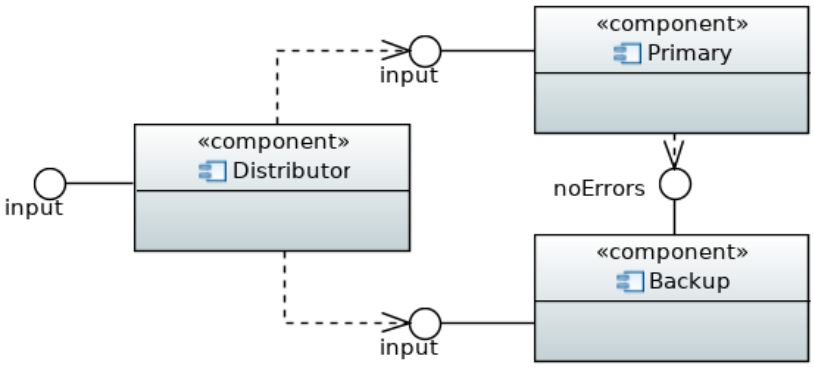



(a) Component Diagram of the *Active Replication* pattern.

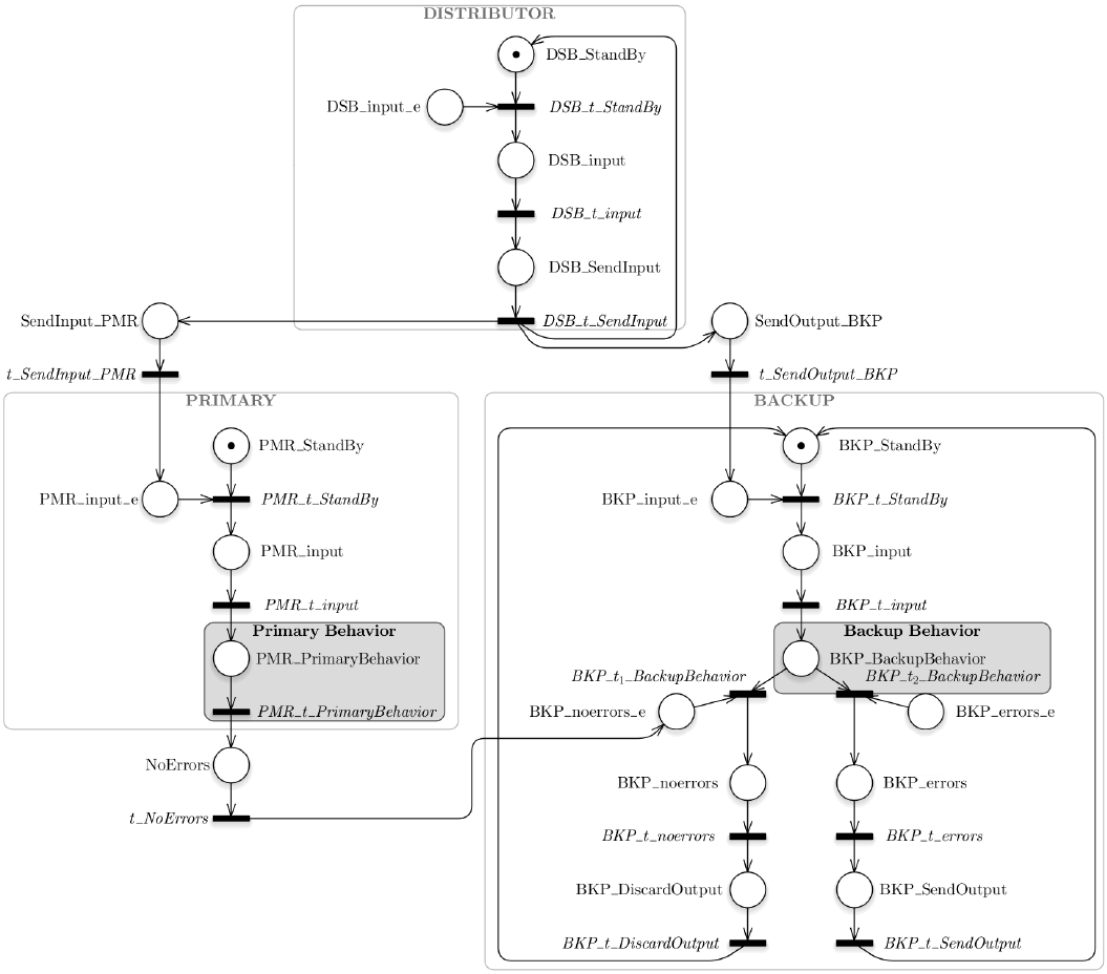



(b) GSPN of the *Active Replication* pattern.

Fig. 4.5 The *Active Replication* pattern.




(a) Component Diagram of the *Semi-Active Replication* pattern.




(b) GSPN of the *Semi-Active Replication* pattern.

Fig. 4.6 The *Semi-Active Replication* pattern.

takes over the responsibility to deliver the output. *Primary*, *Backup* must be deployed to different units of failure.

### 4.2.3 The UML$^{JASA}$-GSPN bidirectional transformation

The implementation of the UML$^{JASA}$-GSPN bidirectional transformation includes the definitions of the following tasks:

- Mapping UML$^{SM}$ to GSPN models and vice versa (UML$^{SM}$-GSPN),
- Composing GSPN subnets by mapping UML$^{SD}$ to GSPN models and vice versa (UML$^{SD}$-GSPN),
- Updating the static view of the architecture (UML$^{CD}$).

We remark that the considered UML diagrams are linked together in accordance with the UML specifications. As a consequence, the coordination between execution semantics of related machines is realized by considering the relationships between transitions and operations. More in detail, each transition has a reference to an event that, in turn, refers to an operation already defined in the Component diagram. For instance, in the State Machine diagrams in Figure 4.9 on page 64, the *getTemperature-Data* transitions in *TemperatureSensor* and *GreenhouseController* refer to the very same homonymous operation.

Specifically, with regard to the elements involved in the transformation, a single `State Machine` is defined for each `Component`, and `Transition` elements in the State Machine Diagram are linked to `Operation` elements in the Component Diagram by means of the `trigger.event.operation` reference. The same elements of `Operation` type are also linked to `Message` elements in Sequence Diagrams through the `signature` reference. Additionally, Sequence Diagrams are linked to Component Diagrams through the `Lifeline` elements that refer to `Component` elements by using the `represents.type` reference.

In the rest of this section, we present a detailed discussion of each of the above mentioned tasks. The complete implementation of the UML$^{JASA}$-GSPN bidirectional transformation is available in Appendix A.

#### UML$^{SM}$-GSPN

The first part of the transformation maps UML$^{SM}$ and GSPN; it is characterized by a one-to-pattern element mapping, meaning that a UML$^{SM}$ element is mapped to a

pattern of GSPN elements. In particular, starting from UML$^{SM}$ the corresponding patterns in GSPN are generated and vice versa. Such implementation considers the formal definition of the unidirectional translation of UML$^{SM}$ in GSPN provided in [21]. Starting from the latter, the relationships between UML$^{SM}$ and GSPN are deduced and then completed in order to define the bidirectional mapping between the notations. The complexity of the latter task is high because the unique bidirectional transformation has to guarantee the syntactic and semantic consistency of source and target models in both directions.

For the sake of detailed illustration, a fragment of the UML$^{SM}$-GSPN bidirectional transformation implemented via JTL is depicted in Listing 4.1. In the following listings, three dots are used in place of repetitive sections of code.

```
1 transformation UMLGSPN (uml:umlsm, pn:ptnet) {
2  ...
3  top relation StateMachine2PetriNet {
4   name: String;
5   enforce domain uml statemachine:umlsm::StateMachine {
6    name=name,
7    region=r:umlsm::Region {}
8   };
9   enforce domain pn petrinet:ptnet::PetriNet {
10   id=name,
11   pages=p:ptnet::Page {}
12  };
13  where {
14   State2Pattern(r, p);
15   StateActivity2Pattern(r, p);
16   Transition2Pattern(r, p);
17  }
18 }
19
20 relation State2Pattern {
21  enforce domain uml r:umlsm::Region {
22   subvertex=s:umlsm::State {}
23  };
24  enforce domain pn p:ptnet::Page {
25   objects=s:ptnet::Place {}
26  };
27  enforce domain pn p:ptnet::Page {
28   objects=s1:ptnet::Transition {
29    transitionKind="immediate"
30   }
```

```
31   };
32   enforce domain pn p:ptnet::Page {
33    objects=s2:ptnet::Arc {
34     source=s:ptnet::Place {}
35     target=s1:ptnet::Transition {}
36    }
37   };
38   where {
39    s.doActivity.oclIsUndefined()
40   }
41  }
42
43  relation StateActivity2Pattern {
44   enforce domain uml r:umlsm::Region {
45    subvertex=s:umlsm::State {
46     doActivity=a:umlsm::Activity {}
47    }
48   };
49   enforce domain pn p:ptnet::Page { ... };
50   enforce domain pn p:ptnet::Page { ... };
51   enforce domain pn p:ptnet::Page { ... };
52   enforce domain pn p:ptnet::Page {
53    objects=s3:ptnet::Place {}
54   };
55   enforce domain pn p:ptnet::Page {
56    objects=s4:ptnet::Arc {
57     source=s1:ptnet::Transition {}
58     target=s3:ptnet::Place {}
59    }
60   };
61   enforce domain pn p:ptnet::Page {
62    objects=s5:ptnet::Transition {
63     transitionKind="exponential"
64    }
65   };
66   enforce domain pn p:ptnet::Page {
67    objects=s6:ptnet::Arc {
68     source=s3:ptnet::Place {}
69     target=s5:ptnet::Transition {}
70    }
71   };
72  }
73
74  relation Transition2Pattern {
```

```
75  enforce domain uml region:umlsm::Region {
76   transition=t:umlsm::Transition {
77    source=s:umlsm::State {}
78   }
79  };
80  enforce domain pn page:ptnet::Page {
81   objects=t:ptnet::Place {}
82  };
83  enforce domain pn page:ptnet::Page {
84   objects=t1:ptnet::Transition {
85    transitionKind="immediate"
86   }
87  };
88  enforce domain pn page:ptnet::Page {
89   objects=t2:ptnet::Arc {
90    source=t:ptnet::Place {}
91    target=t1:ptnet::Transition {}
92   }
93  };
94  enforce domain pn page:ptnet::Page {
95   objects=t3:ptnet::Arc {
96    source=t:ptnet::Place {}
97    target=s1:ptnet::Transition {}
98   }
99  };
100 enforce domain pn page:ptnet::Page {
101  objects=t4:ptnet::Place {}
102 };
103 enforce domain pn page:ptnet::Page {
104  objects=t5:ptnet::Arc {
105   source=s1:ptnet::Transition {}
106   target=t4:ptnet::Place {}
107  }
108 };
109 enforce domain pn page:ptnet::Page {
110  objects=t6:ptnet::Transition {
111  transitionKind="immediate"
112  }
113 };
114 enforce domain pn page:ptnet::Page {
115  objects=t7:ptnet::Arc {
116   source=t4:ptnet::Place {}
117   target=t6:ptnet::Transition {}
118  }
```

```
119   };
120  }
121
122  relation TransitionDaStep2Pattern {
123   name:String;
124   prob:String;
125   enforce domain uml region:umlsm::Region {
126    transition=t:umlsm::DaStep {
127     name=name,
128     occurrenceProb=prob,
129     source=s:umlsm::State {}
130    }
131   };
132   enforce domain pn page:ptnet::Page { ... };
133   enforce domain pn page:ptnet::Page {
134    objects=t1:ptnet::Transition {
135     id=name,
136     weight=prob
137     transitionKind="immediate"
138    }
139   };
140   enforce domain pn page:ptnet::Page { ... };
141   enforce domain pn page:ptnet::Page { ... };
142   enforce domain pn page:ptnet::Page { ... };
143   enforce domain pn page:ptnet::Page { ... };
144   enforce domain pn page:ptnet::Page { ... };
145   enforce domain pn page:ptnet::Page { ... };
146  }
147  ...
```

Listing 4.1 A fragment of the UML$^{JASA}$-GSPN bidirectional transformation.

As said, the transformation is specified by means of a set of *relations* among elements of the two involved *domains*; they represent the transformation rules that can be executed in both directions. The first line of the listing declares the variable `uml` that matches models conforming to the UML$^{SM}$ metamodel and the variable `pn` that matches models conforming to the GSPN metamodel (based on the standard Petri Net Markup Language (PNML) [126]). The main relations specified in the transformation are described as follows:

- `StateMachine2PetriNet` (lines 3-18) generates a container element of type `PetriNet` with attribute `id` from an element of type `StateMachine` with attribute `name`, and

vice versa in the opposite direction. Moreover, the correspondence between the reference `region` of type `Region` and the reference `pages` of type `Page` is defined.

- `State2Pattern` (lines 20-41) maps simple states to a specific pattern. Since a single element in the UML$^{SM}$ domain induces the creation of a list of elements in the GSPN domain, the relation enforces multiple patterns. In particular, for each UML$^{SM}$ `State` in a `Region` (see the reference `subvertex`), the following GSPN elements (see the references `objects`) are created: an element `s` of type `Place`, an element `s1` of type `Transition` (of kind "immediate", marking an immediate GSPN transition), and an element `s2` of type `Arc` that links `s` and `s1`. In the opposite direction, for each occurrence of the described GSPN pattern a correspondent `State` is generated;

- `StateActivity2Pattern` (lines 43-72) considers states that involve elements of type `Activity` and add a pattern of elements to the base pattern defined for simple states. In particular, the following elements are added: `s3` of type `Place`, `s4` of type `Arc` that links the previously created transition `s2` and the place `s4`, `s5` of type `Transition` (of kind "exponential", marking an exponential GSPN transition), and `s6` of type `Arc` that links `s4` and `s5`. Note that the generated GPSN transition will retain the default rate value of 1 as it is the case for all the generated timed transitions. When the actual value of delays is known to the designer, it can be set using the `rate` property of `Transition`. In the opposite direction, for each occurrence of the described GSPN pattern a correspondent `State` is generated;

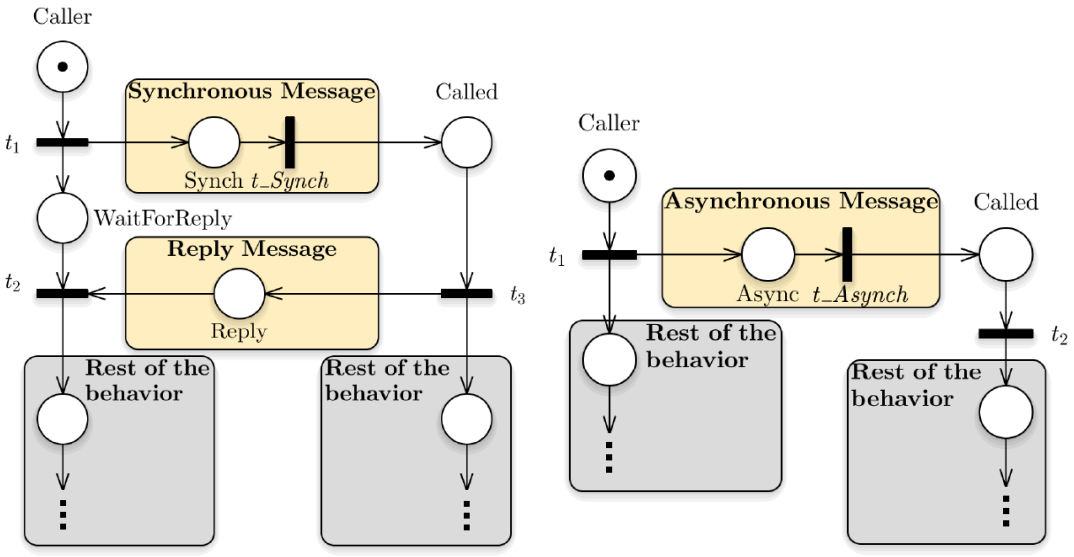
- `Transition2Pattern` (lines 74-120) relates transitions to a specific pattern. In particular, for each UML$^{SM}$ `Transition` in a `Region` (see the reference `transition`), the following GSPN elements (see the references `objects`) are created: an element `t` of type `Place`, an element `t1` of type `Transition` (of kind "immediate"), an element `t2` of type `Arc` that links `t` and `t1`, an element `t3` of type `Arc` that links `t` and the transition `s1` (created from a simple state), an element `t4` of type `Place`, an element `t5` of type `Arc` that links `s1` and `t4`, an element `t6` of type `Transition` (of kind "immediate"), and an element `t7` of type `Arc` that links `t4` and `t6`. In the opposite direction, for each occurrence of the described GSPN pattern a correspondent `Transition` is generated;

- `TransitionDaStep2Pattern` (lines 122-146) relates UML$^{SM}$ transitions annotated with the stereotype `DaStep` from the profile DAM and GSPN transitions (of kind

"immediate"). Moreover, the value of the attribute `occurrenceProb` is mapped to attribute `weight`, and vice versa.

### GSPN subnets composition

The UML$^{SM}$-GSPN transformation in the previous section generates a separate GSPN for each UML$^{SM}$. The set of GSPNs obtained in this way does not represent the entire system as their behavior is not properly connected. These GSPNs can be considered to be subnets of the final system. It is therefore necessary to compose such subnets by connecting them, so that the resulting GSPN represents a system scenario. In this approach, we derive the composition of GSPN subnets from messages exchanged in UML$^{SD}$.




(a) A synchronous message in GSPN (b) An asynchronous message in GSPN

Fig. 4.7 GSPN subnets composition

Specifically, we need to consider two cases: when messages represent a *synchronous* or *asynchronous* call. In case of a synchronous call, as depicted in Figure 4.7a, we need to connect the GSPN immediate transition of the state in which the caller component is currently positioned to the GSPN place of the state in which the called component is positioned when receiving the message. The *reply* message resulting from a synchronous call connects the last GSPN immediate transition representing the end of the called component behavior to the GSPN immediate transition on which a token was waiting for the reply message. In the asynchronous case, shown in Figure 4.7b, the call is

represented similarly to the synchronous case with the important distinction that no token will wait for a reply message as none is expected.

The mapping of UML$^{SD}$ to GSPN is characterized by a one-to-pattern element mapping, meaning that a UML$^{SD}$ element is mapped to a pattern of GSPN elements. In particular, starting from UML$^{SD}$, the corresponding GSPN is generated and vice versa. Such implementation considers the formal definition of the unidirectional translation of UML$^{SD}$ in GSPN provided in [20]. With respect to the latter, we only consider *instantaneous* messages (non delayed).

For the sake of detailed illustration, a fragment of the UML$^{SD}$-GSPN bidirectional transformation implemented via JTL is depicted in Listing 4.2.

```
...
 relation MessageSynch2Pattern {
  enforce domain uml m:umlsm::Message {
   messageSort="synchCall"
  };
  enforce domain pn p:ptnet::Page {
   objects=s:ptnet::Place {}
  };
  enforce domain pn p:ptnet::Page {
   objects=s1:ptnet::Transition {
    transitionKind="immediate"
   }
  };
  enforce domain pn p:ptnet::Page {
   objects=s2:ptnet::Arc {
    source=s:ptnet::Place {}
    target=s1:ptnet::Transition {}
   }
  };
  enforce domain pn p:ptnet::Page {
   objects=s3:ptnet::Arc {
    source=c:ptnet::Transition {}
    target=s:ptnet::Place {}
   }
  };
  enforce domain pn p:ptnet::Page {
   objects=s4:ptnet::Arc {
    source=s1:ptnet::Transition {}
    target=r:ptnet::Place {}
   }
  };
```

```
32   when {
33    State2Pattern(c, c);
34    State2Pattern(r, r);
35   }
36  }
37
38  relation MessageAsynch2Pattern {
39   enforce domain uml m:umlsm::Message {
40    messageSort="asynchCall"
41   };
42   enforce domain pn p:ptnet::Page {  ...  };
43   enforce domain pn p:ptnet::Page {  ...  };
44   enforce domain pn p:ptnet::Page {  ...  };
45   enforce domain pn p:ptnet::Page {  ...  };
46   enforce domain pn p:ptnet::Page {  ...  };
47   when {  ... }
48  }
49
50  relation MessageReply2Pattern {
51   enforce domain uml m:umlsm::Message {
52    messageSort="reply"
53   };
54   enforce domain pn p:ptnet::Page {
55    objects=s:ptnet::Place {}
56   };
57   enforce domain pn p:ptnet::Page {
58    objects=s1:ptnet::Arc {
59     source=r:ptnet::Transition {}
60     target=s:ptnet::Place {}
61    }
62   };
63   enforce domain pn p:ptnet::Page {
64    objects=s2:ptnet::Arc {
65     source=s1:ptnet::Transition {}
66     target=c:ptnet::Place {}
67    }
68   };
69   when {
70    State2Pattern(r, r);
71    State2Pattern(c, c);
72   }
73  }
74 ...
```

Listing 4.2 A fragment of the UML$^{JASA}$-GSPN bidirectional transformation.

The main relations specified in the transformation are described as follows:

- `MessageSynch2Pattern` (lines 2-36) maps messages to a specific pattern. Since a single element in the UML$^{SD}$ domain induces the creation of a list of elements in the GSPN domain, the relation enforces multiple patterns. In particular, for each UML$^{SD}$ `Message` generated with a synchronous type of communication action (messageSort = "synchCall", marking a synchronous message), the following GSPN elements (see the references `objects`) are created: an element `s` of type `Place`, an element `s1` of type `Transition` (of kind "immediate"), an element `s2` of type `Arc` that links `s` and `s1`, an element `s3` of type `Arc` that links `c` (that represent the caller transition) and `s`, and an element `s4` of type `Arc` that links `s1` () and `r` (that represent the receiver place). The elements `c` and `r` are mapped by calling the relation `State2Pattern` (from Listing 4.1) in the `when` clause. In the opposite direction, for each occurrence of the described GSPN pattern a correspondent synchronous `Message` is generated;
- `MessageAsynch2Pattern` (lines 38-48) maps UML$^{SD}$ `Message` generated with an asynchronous type of communication action (messageSort = "asynchCall", marking an asynchronous message) to a specific pattern, similarly to the previous relation; In the opposite direction, for each occurrence of the described GSPN pattern a correspondent asynchronous `Message` is generated;
- `MessageReply2Pattern` (lines 50-73) considers UML$^{SD}$ reply messages (messageSort = "reply", marking a reply message) and generate a pattern of elements in the GSPN domain. In particular, the following elements are added: `s` of type `Place`, `s1` of type `Arc` that links the receiver transition `r` and the place `s`, and `s2` of type `Arc` that links the transition `s1` and the caller place `c`. The elements `c` and `r` are mapped by calling the relation `State2Pattern` (from Listing 4.1) in the `when` clause. In the opposite direction, for each occurrence of the described GSPN pattern a correspondent reply `Message` is generated;

### Static view (UML$^{CD}$) update

After the refactoring and analysis steps are performed on the GSPN, the execution in backward direction of the transformation propagates the changes from GSPN to UML. This back propagation also affects the static view of the system, that is represented by a UML Component Diagram (UML$^{CD}$). For example, when a replica of a sensor is created in GSPN, the corresponding new component should be automatically generated

in the UML$^{CD}$. In order to achieve this, we introduce additional relations updating the static view of the system, as reported in the Listing 4.3.

```
...
 relation Component2Page {
  name:String;
  enforce domain uml c:umlsm::Component {
   name=name
  };
  checkonly domain pn p:ptnet::Page {
   id=name
  };
 }

 relation Interface2Pattern {
  enforce domain uml i:umlsm::Interface {
   ownedOperation=o:umlsm::Operation {}
  };
  enforce domain uml receiver:umlsm::Component {
   interfaceRealization=ir:umlsm::InterfaceRealization {
    supplier=i:umlsm::Interface {},
    contract=i:umlsm::Interface {}
   }
  };
  enforce domain uml caller:umlsm::Component {
   packagedElement=d:umlsm::Dependency {
    client=caller:umlsm::Component {},
    supplier=i:umlsm::Interface {}
   }
  };
  enforce domain pn p:ptnet::Page {
   objects=s:ptnet::Place {}
  };
  enforce domain pn p:ptnet::Page {
   objects=s1:ptnet::Transition {
    transitionKind="immediate"
   }
  };
  enforce domain pn p:ptnet::Page {
   objects=s2:ptnet::Arc {
    source=s:ptnet::Place {}
    target=s1:ptnet::Transition {}
   }
  };
```

```
42  enforce domain pn p1:ptnet::Page {
43   objects=s3:ptnet::Arc {
44    source=c:ptnet::Transition {}
45    target=s:ptnet::Place {}
46   }
47  };
48  enforce domain pn p2:ptnet::Page {
49   objects=s4:ptnet::Arc {
50    source=s1:ptnet::Transition {}
51    target=r:ptnet::Place {}
52   }
53  };
54  when {
55   Component2Page(caller, p1);
56   Component2Page(receiver, p3);
57  }
58  where {
59   s3.source.containerPage.id <>
60      s3.target.containerPage.id;
61   s4.source.containerPage.id <>
62      s4.target.containerPage.id;
63  }
64 ...
```

Listing 4.3 A fragment of the UML$^{JASA}$-GSPN bidirectional transformation.

The main relations specified in the transformation are described as follows:

- `Component2Page` (lines 2-10) maps a UML `Component` to a GSPN `Page`. Following the design assumption that a State Machine is created to describe the behaviour of a Component, this relation creates a correspondence between a `Component` in UML and a GSPN subnet enclosed in a `Page` that contains the behaviour defined in a State Machine. When executed in the backward direction, this relation generates a new `Component` for each new `Page` added by the refactoring in GSPN.
- `Interface2Pattern` (lines 12-63) maps a UML pattern composed of an `Interface`, its realization and usage to a GSPN pattern defining a call operation between components. Specifically, the UML pattern is composed of an `Interface`, its `ownedOperation` and two `Components`, one receiving the call (`receiver`) and the other performing it (`caller`). On the GSPN side, the relation matches the pattern corresponding to a call operation between components (the pattern matches both synchronous and asynchronous calls as they are differentiated only by the presence of a reply message). The `when` clause is used to ensure that the matched components have been mapped to different pages.

In order to guarantee that the matched call is happening between two components, the `where` clause contains two constraints requiring that `source` and `target` references of the `Arcs` `s3` and `s4` point to different `Page`s.

Next section shows how the approach is applied to an example scenario.

## 4.3 Evaluation

In this section, we present the approach in practice with the aim of illustrating the JASA process and how it can be replicated by potential researchers and practitioners that would like to follow the same process on their own architecture.

The experiment is conducted by applying the approach to the Environmental Control System (ECS) system example (as described in Section 4.3.1). First, the system has been modelled by means of UML annotated with the MARTE DAM profile (as described in Section 4.1). Then, in order to be used in the EMF environment, the involved models have been specified in their Ecore format. The approach has been executed within the JTL framework; in particular, the UML$^{\text{JASA}}$-GSPN bidirectional transformation has been run in the forward direction to generate the GSPN models (as described in Section 4.3.2); after performing the GSPN analysis, a set of refactoring actions have been performed on the GSPNs on the basis of the obtained results (as described in Section 4.3.3). Finally, the UML$^{\text{JASA}}$-GSPN bidirectional transformation has been run in the backward direction to propagate the changes and generate the updated UML architecture (as described in Section 4.3.4).

We also discuss the evaluation we have performed with the aim of answering the following research questions:

*RQ1*: Does the approach generate an analyzable availability model from a software architecture model?

*RQ2*: Does the approach generate a valid software architecture model back from an availability model?

*RQ3*: Does the approach help to identify the fault tolerance patterns that better improve the system availability?

In order to assess the approach according to the research questions, several measurements and properties have been considered for each step of our evaluation. The results of the performed experiments are discussed in the context of each research question on the basis of the selected evaluation criteria.

### 4.3.1 Environmental Control System (ECS) modeling

The approach presented in the previous sections has been applied to a software system for the environmental control of a botanical garden. The Environmental Control System (ECS) is responsible for the automated management of the artificial habitat preserved in greenhouses. A network of sensors periodically checks air temperature, air humidity and soil humidity inside greenhouses. When sensors detect values exceeding the thresholds defined for a given greenhouse, the system automatically restores the environment conditions activating irrigation and air conditioning systems as required.

ECS consists of seven software components: *GreenhouseController* is responsible for checking environment conditions; *TemperatureSensor*, *AirHumiditySensor* and *SoilHumiditySensor* respectively measure air temperature, air humidity and soil humidity; *Database* is queried to retrieve the thresholds defined for each monitored condition; *AirConditioner* can raise or decrease the air temperature inside a greenhouse; *MobileApp* notifies the user about certain events such as conditions exceeding the defined thresholds.

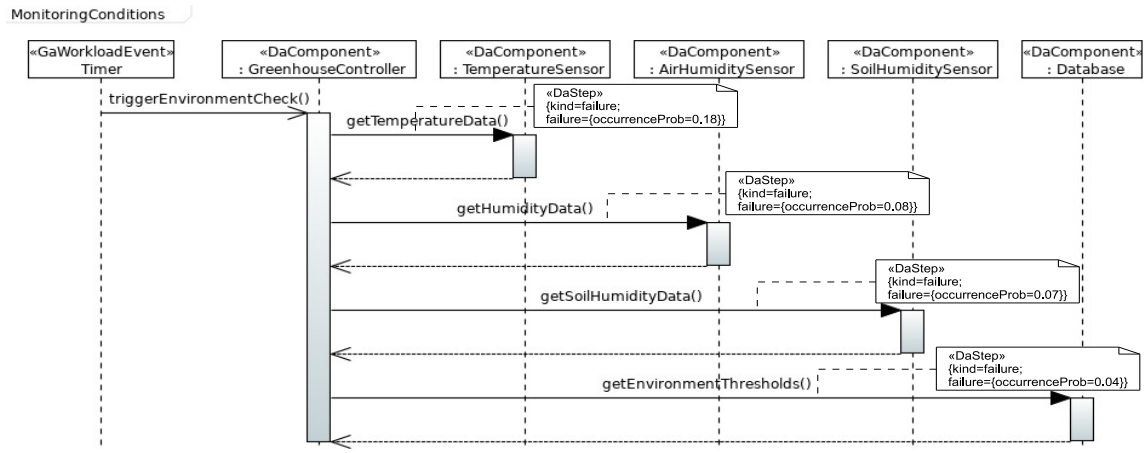
We consider three use case scenarios of ECS, for which we provide the respective UML Sequence Diagrams: *Monitoring Conditions*, in Figure 4.8a, in which a timer periodically activates a procedure to check environment conditions, *Remote Monitoring*, in Figure 4.8b, in which the air humidity is continuously monitored and the *GreenhouseController* notifies the user when the value exceeds the corresponding threshold, and *Managing Temperature*, in Figure 4.8c, that defines the procedure for the activation of the air conditioner when required. We assume that the complexity of a message parameters and return types, as well as the width of their ranges, do not affect the behaviour following that message.

Moreover, the internal behavior of each software component is described by a State Machine that is consistent with the interactions defined in the Sequence Diagrams. The resulting State Machine diagram is shown in Figure 4.9.

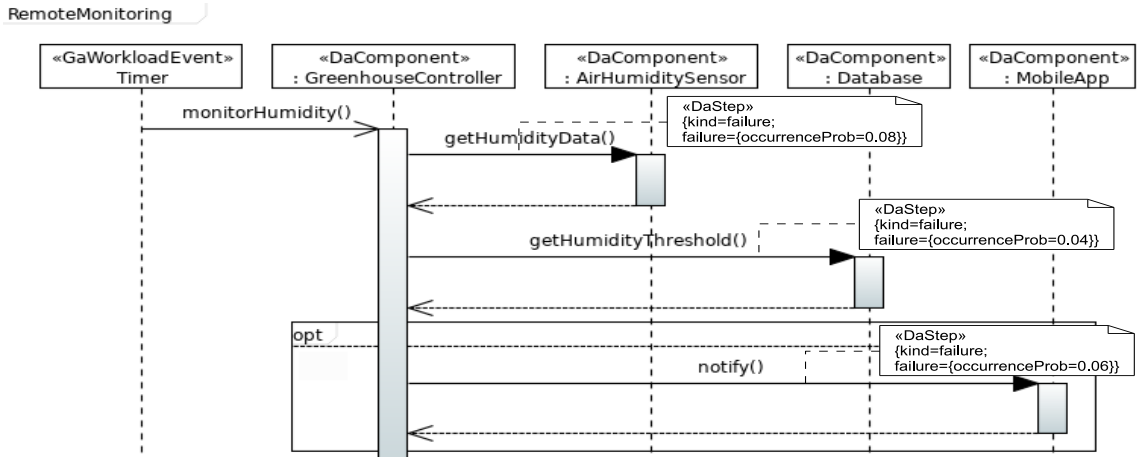
UML *Transition* and *Message* elements that may fail are annotated with the *DaStep* stereotype from DAM, as depicted in Figures 4.9 and 4.8, respectively. This stereotype is used here to define system failure modes and the probabilities of failures occurring in a scenario, as follows: attribute *kind* is set to *failure*, as a consequence, the attribute *failure* can be used to set the failure probability as the *occurrenceProb* real value.

The considered models, specified in UML, are available online[2].

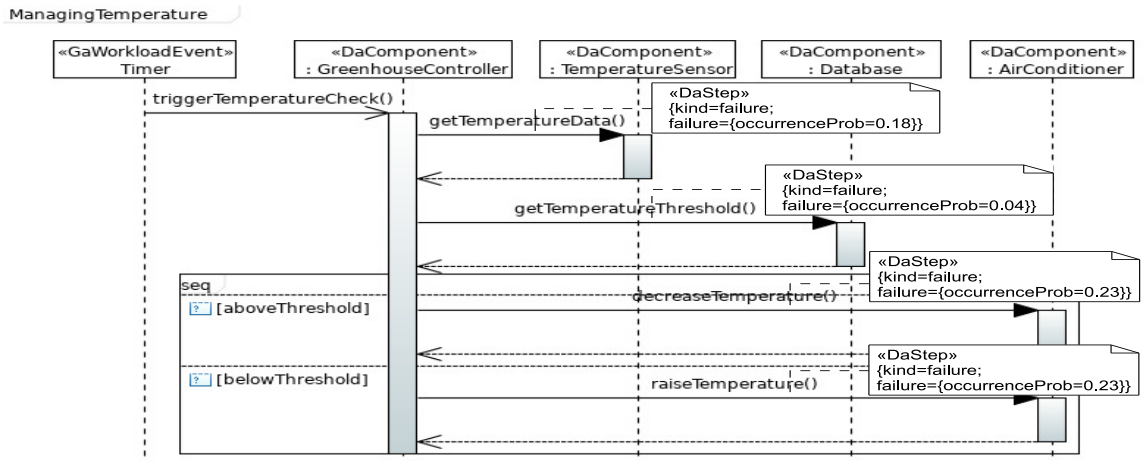
[2] https://github.com/SEALABQualityGroup/JASA/tree/master/UML




(a) The *Monitoring Conditions* scenario.




(b) The *Remote Monitoring* scenario.




(c) The *Managing Temperature* scenario.

Fig. 4.8 UML Sequence Diagrams of the ECS scenarios.

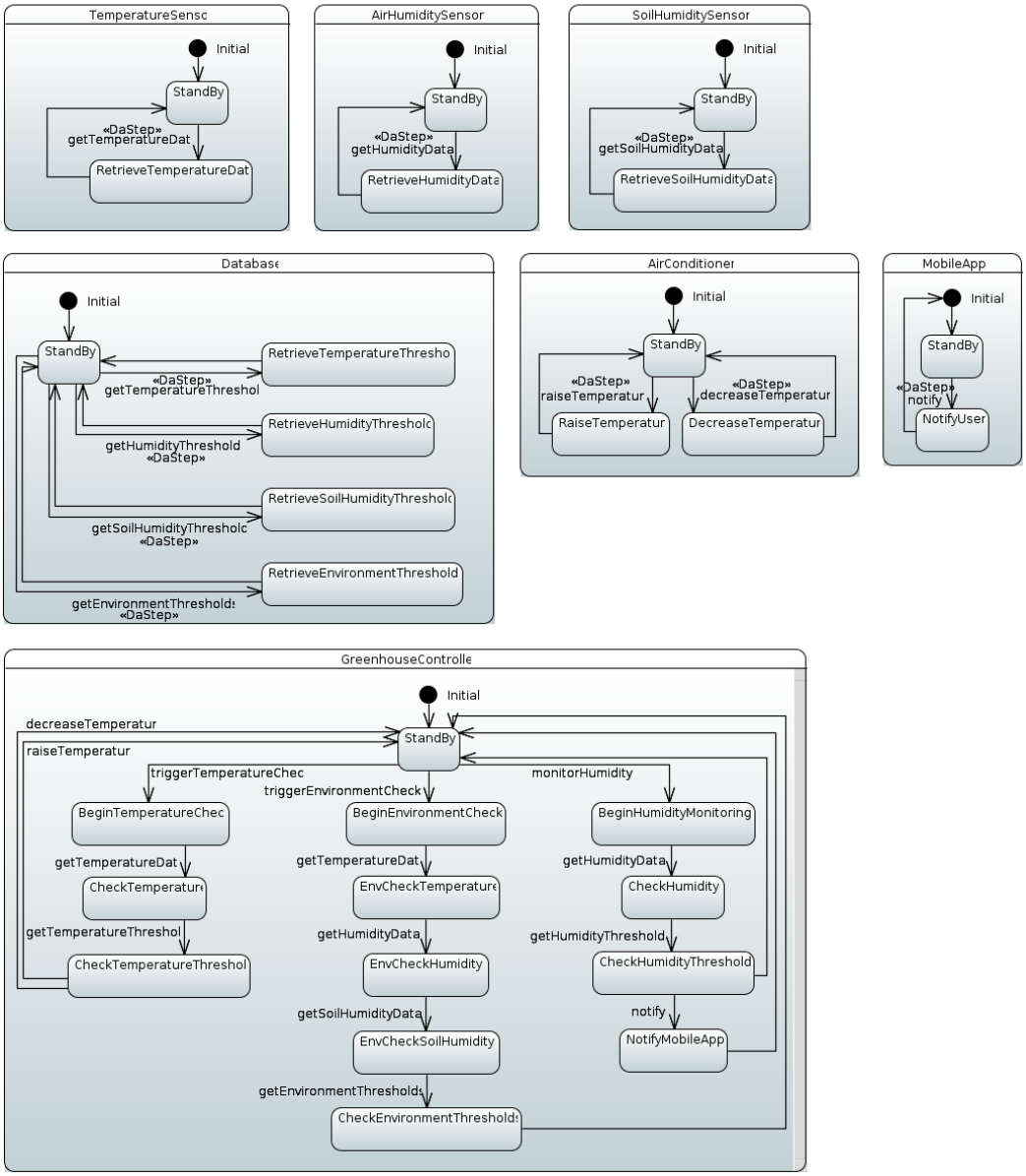



Fig. 4.9 UML State Machine Diagram of the ECS components.

### 4.3.2 Analysis model generation

The first operational step of our approach consists in the execution (in the forward direction) of the transformation presented in 4.2.3 (UML$^{JASA}$-GSPN) within the JTL framework. For each scenario, from a Sequence Diagram and the set of involved State Machines, this execution generates a GSPN. The transformation UML$^{SM}$-GSPN in Section 4.2.3 creates a GSPN subnet for each State Machine. As an example, Figure 4.10 shows a fragment of the GSPN obtained for the *Managing Temperature* scenario (Figure 4.8c). The GSPN subnets visible in the figure are generated from the *TemperatureSensor*, *AirConditioner* and *GreenhouseController* State Machines in Figure 4.9, where colours are used to outline the subnets generated from the corresponding State Machines. Such subnets are connected on the basis of the transformation UML$^{SD}$-GSPN in Section 4.2.3.

In general, the composition of subnets obtained from this step is based on interactions among components, as appearing in Sequence Diagrams. In particular, synchronous and asynchronous messages are mapped to the corresponding patterns presented in Section 4.2.3[3].

### 4.3.3 Analysis results and refactoring

In this section, first we play with a simple case for checking whether patterns induce differences in the system availability. Thereafter, we apply patterns to components on the basis of current practices and component role, as it will be explained in detail at the end of the section. The result of patterns application is a unique static architecture that subsumes different SD, hence different availability results for different scenarios (in terms of operational profile and workload).

As a first step, we consider the GSPN obtained from the execution in the forward direction of the transformation to perform a steady state availability analysis. Given an initial marking of a GSPN, and provided that every place of the net is bounded, the reachability set is the set of all the markings reachable by sequences of transition firings from the initial one. The reachability graph associated to a GSPN is a directed graph whose nodes are the markings in the reachability set and each arc, connecting a marking $M$ to a $M'$ one, represents the firing of a transition enabled in $M$ and leading to $M'$.

[3]The GSPNs generated for each scenario are available at https://github.com/SEALABQualityGroup/JASA/tree/master/GSPN.

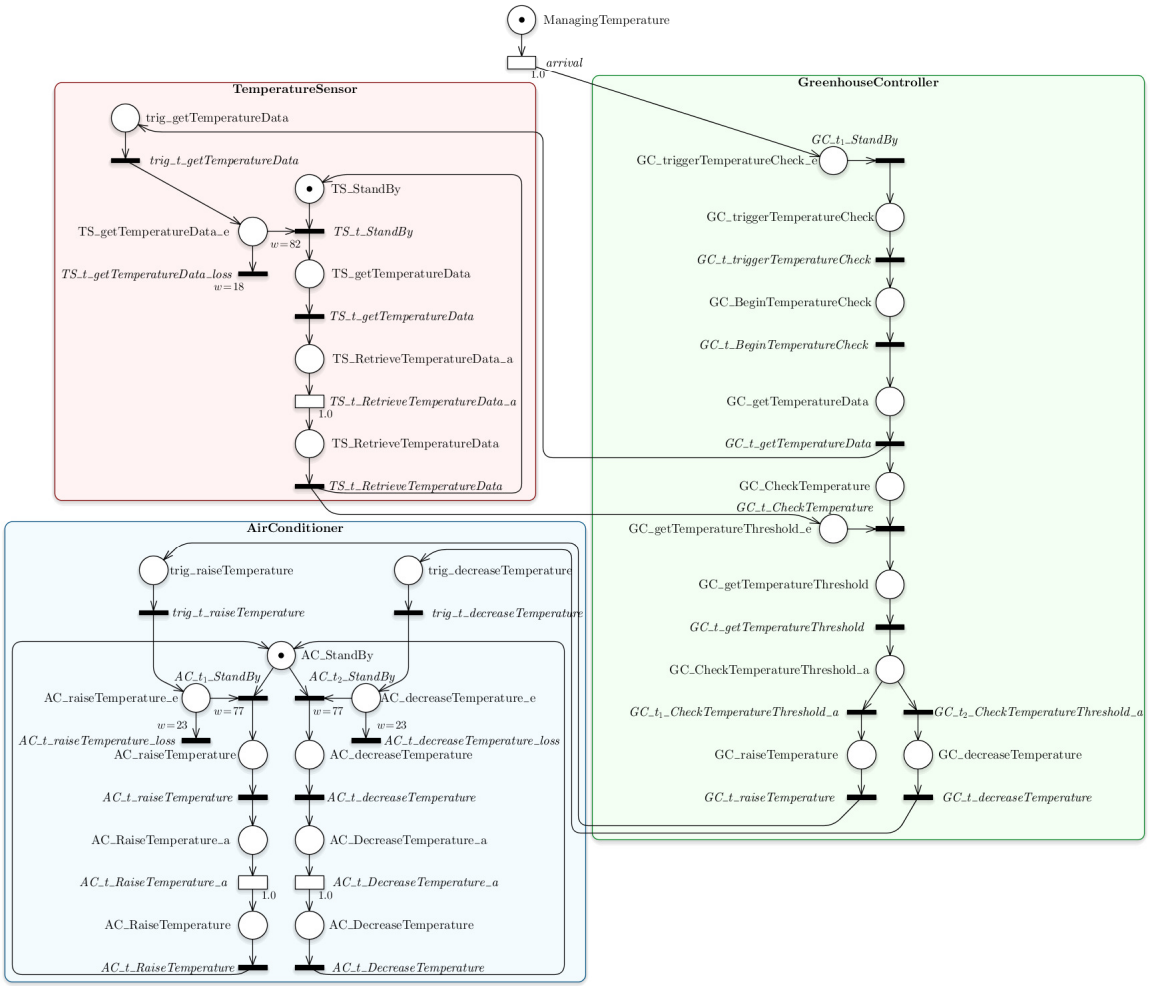



Fig. 4.10 Fragment of the GSPN generated for the *Managing Temperature* scenario.

In general, availability metrics of an GSPN model can be defined as reward functions on the reachability graph [60]. Let $M^0$ be the initial marking, and $r_M = \{1 \text{ if } M \in O,\ 0 \text{ if } M \in F\}$ be a state reward function that partitions the set of reachable markings $RS(M^0)$ into two sets: $O$, the set of operational system states, and $F$, the set of system failure states. The probability of the system being in marking $M$ at time instant $t$ can be expressed as $\sigma_M(t) = Pr\{X(t) = M\}$. Steady state probability can be computed as $\sigma_M = lim_{t\to\infty}\sigma_M(t)$, and it represents the probability of the system being in marking $M$ at any time instant $t > 0$. The steady state availability of the GSPN is then defined taking into account the reward function and the steady state probabilities of individual markings introduced before, as follows: $A_\infty = \sum_{M\in RS(M_0)} r_M\sigma_M = \sum_{M\in O}\sigma_M$. The value of $A_\infty$ is to be interpreted as the percentage of time the system is not in a failure state after running for a sufficiently long time.

System failure mode needs to be defined in order to discern operational states from failure ones, and to exclusively assign the related markings to one among the $O$ and $F$ subsets of reachable markings. The system is considered to be in a failure state when any of the state transitions annotated by the *DaStep* stereotype fails during execution. As a consequence, in the GSPN obtained from the previous step, we define as failure states the markings reached from firing all the transitions having the *_loss* suffix, as they represent the occurrence of a failure. When we will apply fault tolerance refactorings, some of the errors represented by *_loss* transitions will be masked. Therefore, we will not consider such transitions as leading to a failure state.

The GreatSPN solver [35] is used to derive the reachability graph of markings in the net and to compute the corresponding values of $\sigma_M$. In the initial marking of the net, a token appears in the `StandBy` place of each component subnet, so that the component is ready to serve incoming requests. Immediate transitions representing failures are marked with weights derived from the failure probabilities [4]. Since we assume that the operations belonging to the same component fail with the same probability, we report in Table 4.1 the initial failure probabilities of every component in ECS.

The steady state availability index can be computed by considering both the aforementioned initial marking of the GSPN and the failure probabilities. We used GreatSPN to simulate the GSPNs with 10000 iterations. Iterations that fire a failure transition will cause the GSPN to halt and consequently decrease the availability of the system. The resulting indices for the three scenarios we considered are reported in Table 4.2.

[4]Note that, for presentational purposes, the failure probabilities are intentionally chosen higher than those usually observed in reality. Very small values would hardly show any noticeable change in availability with $10^{-6}$ precision.

| | |
|---|---|
| TemperatureSensor | 0.18 |
| HumiditySensor | 0.08 |
| SoilHumiditySensor | 0.07 |
| Database | 0.04 |
| MobileApp | 0.06 |
| AirConditioner | 0.23 |

Table 4.1 Initial failure probabilities of components in ECS

| | |
|---|---|
| Monitoring Conditions | 0.985392 |
| Remote Monitoring | 0.991672 |
| Managing Temperature | 0.977984 |

Table 4.2 Steady state availability of execution scenarios

In order to establish the effectiveness of the fault tolerance patterns presented in Section 4.1.3, we apply each of them on the *TemperatureSensor* component in the *Managing Temperature* scenario. The steady state availability resulting in each case is reported in Table 4.3. The results show, as expected, that the application of the fault tolerance patterns increased the overall availability of the scenario, with the particular observation that *Active Replication* and *Passive Replication* induce the best improvements. Note that even a change in the second decimal digit of availability metric is already considered relevant, since high availability systems usually require to be available up to the 99.999% of the running time (this requirement is usually referred to as *five nines*) [119].

| | |
|---|---|
| Initial (no refactoring) | 0.977984 |
| Semi-Active Replication | 0.985605 |
| Active Replication | 0.988511 |
| Semi-Passive Replication | 0.98026 |
| Passive Replication | 0.989855 |

Table 4.3 Steady state availability of the *Managing Temperature* scenario after the application of fault tolerance patterns on *TemperatureSensor*

In order to further improve the system availability, we apply the *Semi-Active Replication* pattern to all the sensors components in the example application, as this pattern has proved effective in the deployment of sensors in high availability contexts [104]. Since the *Active Replication* pattern is widely used in practice to deploy high availability databases [72], we apply it to the *Database* component in each scenario. The results obtained from this refactoring are discussed in Section 4.3.7. An additional reason for the application of the *Active* and *Semi-Active Replication* patterns over their *Passive* and *Semi-Passive* counterparts resides in the stateless

nature of the functionalities provided by the sensors and the database in the example application we are considering. Indeed, since the *Passive* and *Semi-Passive Replication* patterns accomplish error masking by saving the current state of a component through checkpoints, their application to stateless operations would only increase error masking complexity and cost without providing additional benefits over the *Active* and *Semi-Active Replication* patterns.

### 4.3.4 Change propagation

After the analysis and refactoring step, the UML$^{JASA}$-GSPN bidirectional transformation is applied in backward direction on the refactored GSPN model. In particular, the refactored UML Sequence and State Machine Diagrams are generated for each scenario. These new diagrams contain the changes applied to the GSPN during the refactoring step and propagated back by the execution of the transformation.

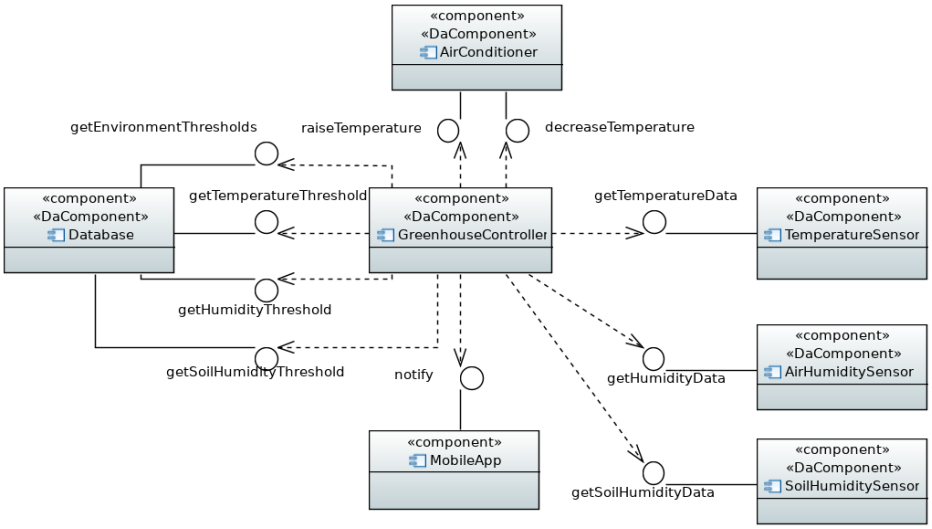
Moreover, the back propagation of changes generates additional software components. The updated Component Diagram is reported in Figure 4.11b. In particular:

- *Monitoring Conditions*: the components *TS_Distributor*, *TemperatureSensor1*, *HS_Distributor*, *AirHumiditySensor1*, *SHS_Distributor*, and *SoilHumiditySensor1* have been introduced by the application of the *Semi-Active Replication* pattern on *TemperatureSensor*, *AirHumiditySensor*, and *SoilHumiditySensor*, while the components *DB_Distributor*, *DB_Comparator*, and *Database1* have been introduced by the application of the *Active Replication* pattern on *Database*;

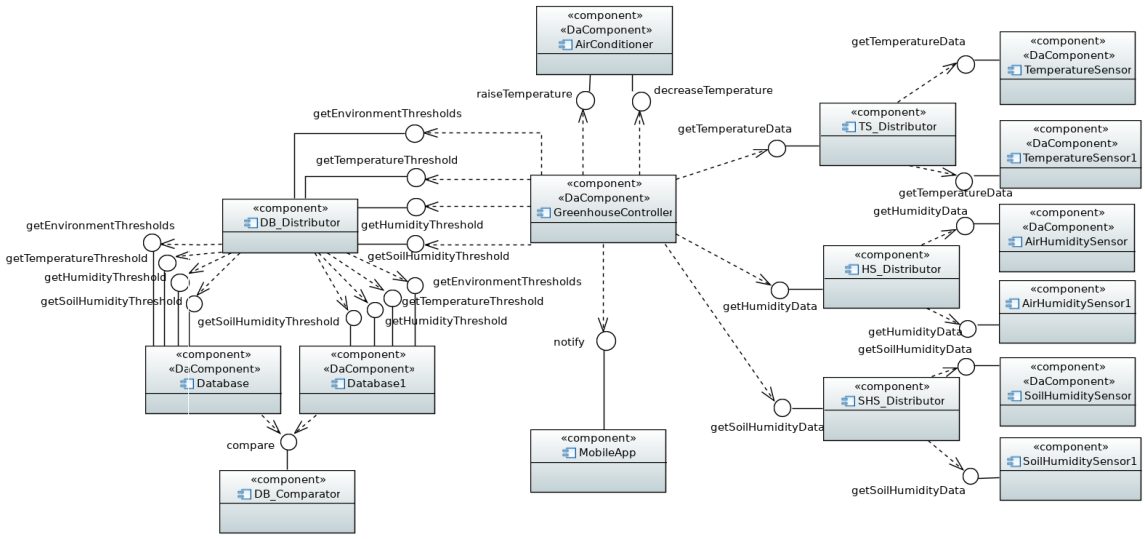
- *Remote Monitoring*: the components *HS_Distributor*, *AirHumiditySensor1*, have been introduced by the application of the *Semi-Active Replication* pattern on *AirHumiditySensor*, while the components *DB_Distributor*, *DB_Comparator*, and *Database1* have been introduced by the application of the *Active Replication* pattern on *Database*;

- *Managing Temperature*: the components *TS_Distributor*, *TemperatureSensor1*, have been introduced by the application of the *Semi-Active Replication* pattern on *TemperatureSensor*, while the components *DB_Distributor*, *DB_Comparator*, and *Database1* have been introduced by the application of the *Active Replication* pattern on *Database*;

As a consequence of the back propagation, nine new state machines have been generated by enforcing the `StateMachine2PetriNet` relation and its triggered relations. The state machines corresponding to the original components are instead restored




(a) Initial Component Diagram of ECS.




(b) Refactored Component Diagram of ECS.

Fig. 4.11 Component Diagrams before and after the change propagation.

without any modification. In addition, each state machine corresponding to replicas in the *Semi-Active Replication* pattern (i.e., all sensors' replicas) includes a new `discardOutput` transition that represents the case in which no failure occurs in the original component and, as a consequence, the data computed by the replica must be discarded. As an example, the UML State Machines generated (*TS_Distributor* and *TemperatureSensor1*) and restored (*TemperatureSensor*) from the *Semi-Active Replication* pattern on the *TemperatureSensor* component are included in Figure 4.12.

The refactored UML Sequence Diagrams for the scenarios *Monitoring Conditions*, *Remote Monitoring*, and *Managing Temperature* are shown in Figures 4.13, 4.14, and 4.15, respectively. In such diagrams, the application of the *Semi-Active Replication* pattern can be noticed by the presence of the `discardOutput` message that is sent from each sensor component (e.g., *TemperatureSensor*, *AirHumiditySensor*, *SoilHumidity-Sensor*) to its corresponding replica. Moreover, alternative fragments are created to model the two cases in which a failure may or may not occur. Lifelines for the newly created distributor and comparator components are included as well.

Finally, the obtained model is consistent with respect to the consistency relation defined in the transformation, and it is compliant with the source metamodel.

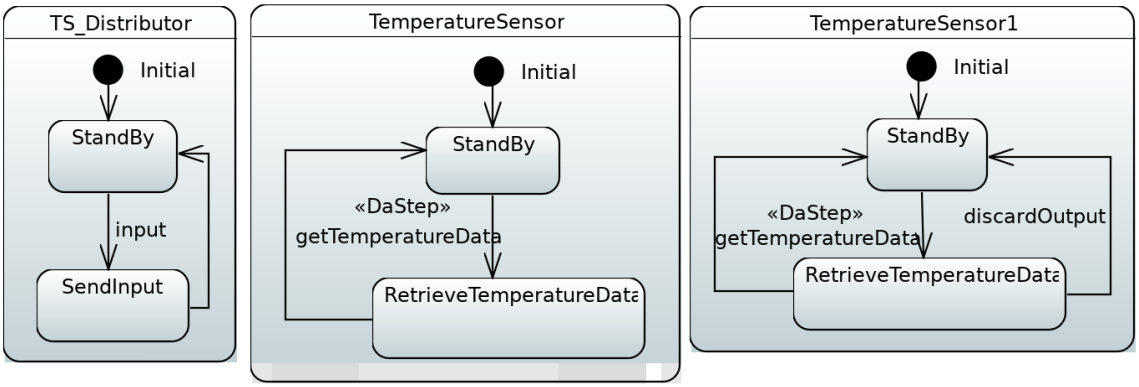



Fig. 4.12 UML State Machines generated from the back propagation of the *Semi-Active Replication* pattern on *TemperatureSensor*.

### 4.3.5 RQ1: Analizability of the generated analysis models

In order to answer this research question, we have observed the results obtained by transforming the UML models in the corresponding GPSN models, as well as by applying the refactoring actions. For evaluating if the considered GSPN models can support our analysis, we refer to a set of basic behavioral properties (as introduced in [93]) discussed as following.

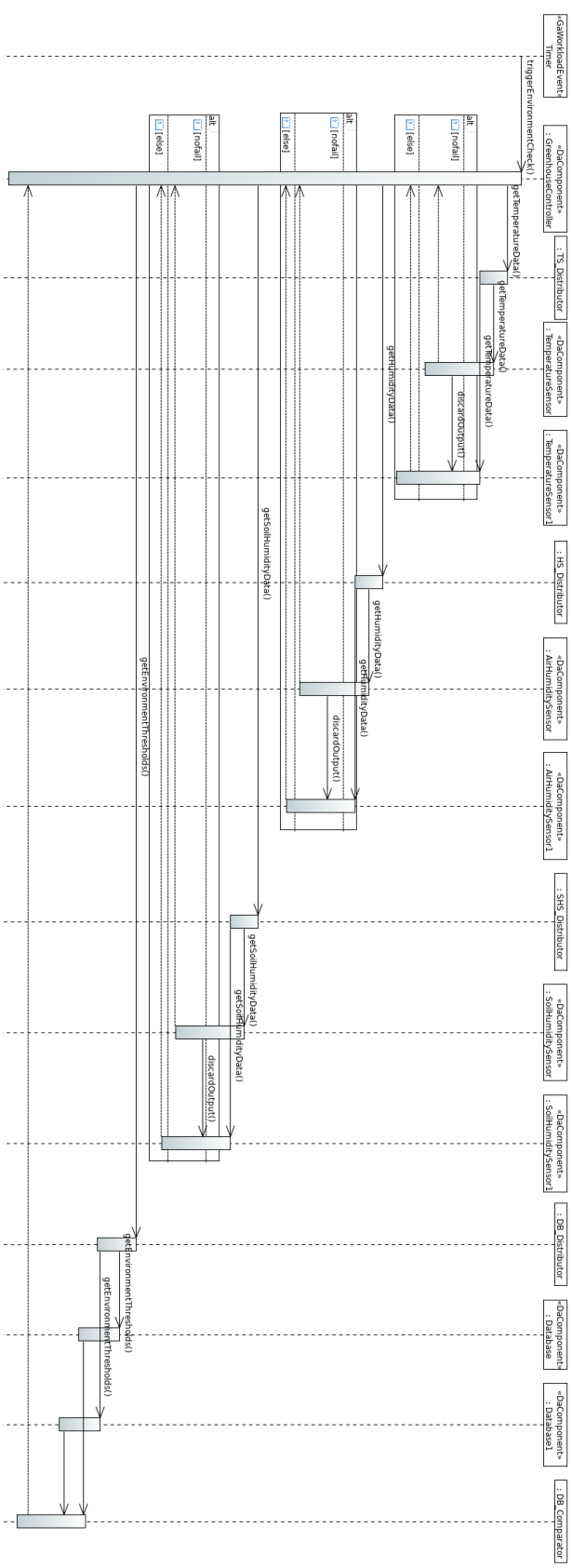



Fig. 4.13 UML Sequence Diagram of the *Monitoring Conditions* scenario.

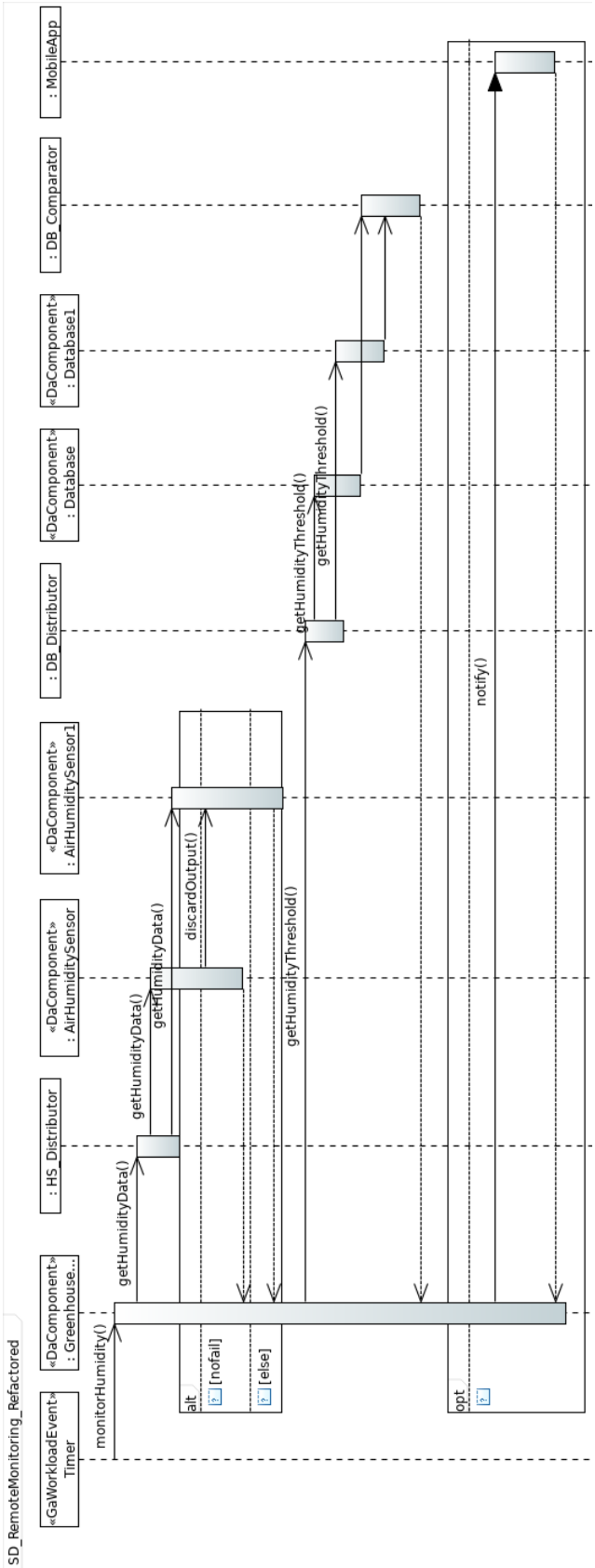



Fig. 4.14 UML Sequence Diagram of the *Remote Monitoring* scenario.

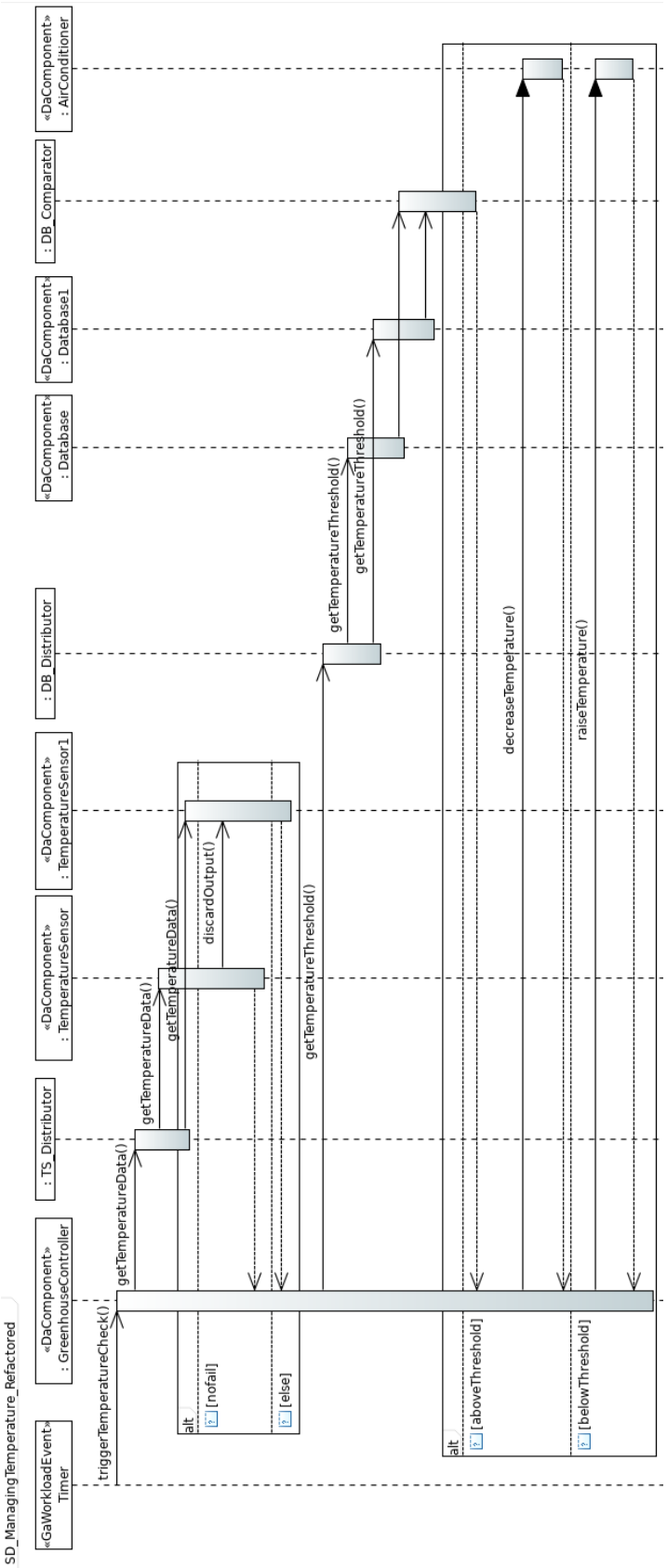



Fig. 4.15 UML Sequence Diagram of the *Managing Temperature* scenario.

*Reachability:*

In order to decide if the considered GSPN is reachable, we have to establish if any state of the modeled system is reachable from the initial state through a finite sequence of transitions. Formally, it is the problem of finding if any given marking $M$ is contained in the set of markings reachable from the initial marking $M^0$. This property is required since the availability metrics we considered are defined as reward functions on the reachability graph associated to the GSPN, as described in 4.3.3.

We verified the reachability of our GSPN models by using the GreatSPN tool, that is able to compute reachability graphs where every marking in the net is reachable from $M_0$. In our experiment, we can observe that all the reachability graphs have been successfully created. In Table 4.4, we report the cardinality of the reachability set $RS(M_0)$ for each scenario. In particular, the *Initial* values refer to the GPSN models obtained by applying the UML$^{\text{JASA}}$-GSPN bidirectional transformation, whereas the *Refactored* values refer to the GSPN models after the refactoring described in Section 4.3.3. The new elements introduced by the refactoring of the GSPN caused an increase in the cardinality of the reachability sets because they originated new markings. Since we were able to compute finite reachability sets, we can assert that the application of the transformation in forward direction and of the refactoring patterns have generated reachable GSPN models.

| | Initial | Refactored |
|---|---|---|
| Monitoring Conditions | 73 | 152 |
| Remote Monitoring | 66 | 105 |
| Managing Temperature | 77 | 116 |

Table 4.4 The cardinality of the reachability set of the GSPNs

*Boundedness*:

A GSPN model is said to be bounded or safe if the number of tokens in each place does not exceed a fixed number for any marking reachable from the initial marking $M_0$. This property is required for the steady state availability analysis as bounded GSPNs are isomorphic to finite Markov Chains [88].

By considering that (i) a GSPN is bounded if and only if its reachability graph is finite [102], and (ii) we showed in Table 4.4 that finite reachability sets can be computed before and after the refactoring, we can assert that all the GSPNs (i.e., initial and refactored ones) are bounded.

More generally, our transformation is designed so that the generated GSPNs cannot contain transitions without input places. This property is a necessary condition for

boundedness. Moreover, none of the proposed refactorings introduces this type of transitions.

*Liveness*:

This property is closely related to the complete absence of deadlocks. A GSPN is said to be live if, for any reachable marking, it is possible to ultimately fire any transition of the net through some further firing sequence.

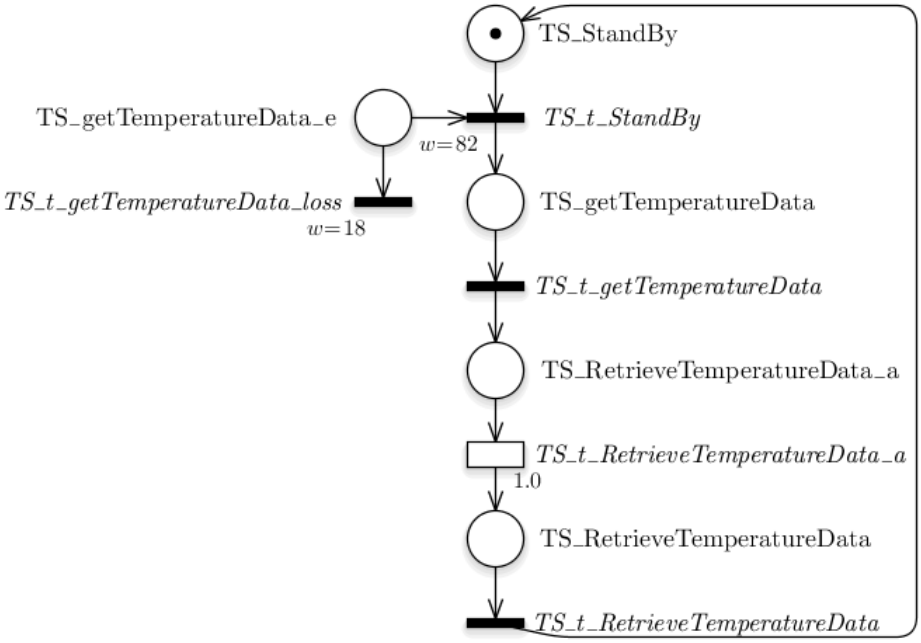



Fig. 4.16 GSPN subnet of the *Temperature Sensor*.

In our experiment, we can observe that all the GSPNs (both initial and refactored) are live, because from any reachable state it is possible to enable any transition by a firing sequence. In particular, the transitions modeling failures are *L1-live*, as they can be fired at least once in some firing sequence starting from the initial marking $M_0$. Note that, in the GSPNs we generated, any failure transition can be fired exactly once per solver iteration. After the firing of the first failure transition, the GSPN will stop, therefore modeling a crash failure of the system. As an example, the transition *TS_t_getTemperatureData_loss* in Figure 4.16 is a *L1-live* one, as it can potentially fire only once, when at least one token is in the place *TS_getTemperatureData_e*. All the other transitions in the GSPN are *L3-live*, since they can fire infinitely (as long as no failures occur), as well as all transitions in Figure 4.16 except *TS_t_getTemperatureData_loss*. In general, liveness of obtained GSPNs can be checked by the GreatSPN solver that we have adopted.

### 4.3.6 RQ2: Validity of the refactored architecture

In order to answer this research question, we have observed the results obtained by transforming the refactored GSPN back to the UML software architecture. To evaluate if the refactored architectural model is a still valid software architecture, we considered a set of properties that are commonly used in the analysis of software architectures [118].

*Correctness*:

It is an external property of an architectural model and ensures that it fully realizes the system specification. In order to evaluate the correctness of a refactored UML model resulting from the application of the approach, we need to consider the following aspects:

- We assume that the initial software architectural model is correct (i.e., it realizes the system specification).
- The refactoring applied on the GSPN model obtained from the forward application of the implemented UML$^{JASA}$-GSPN bidirectional transformation does not break the conformance to the system requirements. In fact, the adopted fault tolerance patterns make use of replicas and checkpoints techniques to provide error masking, thus without altering the original functionalities of the refactored component (as detailed in Section 4.2.2).
- The UML$^{JASA}$-GSPN bidirectional transformation is able to generate consistent solutions with respect to the relations specified in the transformation itself. In other words, the backward application of the transformation propagates changes by correctly mapping the refactoring patterns on GPSN in refactoring patterns in UML (i.e., without altering the original functionalities of the system). For instance, when a replica is introduced in the GSPN (e.g., Semi-Active and Active Replication pattern in Section 4.3), an additional state machine that contains the same states and transitions of the original component is introduced in the UML model, as well as additional messages from/to the replicated component.
- Finally, this aspect is strictly related to the correctness of bidirectional transformations. Formally, a bidirectional transformation $T$ between two classes of models, $M$ and $N$, is characterized by two unidirectional transformations: $\overrightarrow{T} : M \times N \rightarrow N$ and $\overleftarrow{T} : M \times N \rightarrow M$. $T$ is said to be *correct* if for any pair of models $m \in M$ and $n \in N$, $T(m, \overrightarrow{T}(m,n))$ and $T(\overleftarrow{T}(m,n), n)$ [114]. The capability of the JTL framework to correctly execute the transformation is discussed in [36, 49]. As a proof of concept, by running our transformation on forward

and backward directions without any change on the example application, the transformation generated the same pair of models.

*Completeness*:

This property is verified whether all necessary architectural elements are defined and whether all design decisions are made. In order to evaluate the completeness of the refactored UML model, let us to consider the following aspects:

- We assume that the initial software architectural model is complete.
- The refactoring applied on the obtained GSPN model operates only on components with probability of failure, without eliminating or modifying other architectural elements, where changes performed on those components are limited to the error handling. For example, in the Semi-Active Replication pattern described in Section 4.2.2, the primary component is enriched exclusively with elements that allow sending messages to the backup component in order to signal that no errors occurred and the output can be discarded.
- The UML$^{\text{JASA}}$-GSPN bidirectional transformation is able to preserve the completeness of the solution with respect to the relations specified in the transformation itself. The changes defined in the refactoring patterns are mapped in changes involving only the corresponding components without eliminating or modifying other architectural elements. For instance, the modification described above is translated in UML by means of adding a message in the corresponding Sequence Diagram and a transition in the corresponding State Machine.
- Finally, this property is related to another property of bidirectional transformations, namely the *hippocraticness* [114]. A transformation $T$ is said to be *hippocratic* if for any model $m \in M$ and $n \in N$, $T(m,n)$ implies $\overrightarrow{T}(m,n)$ and $T(m,n)$ implies $\overleftarrow{T}(m,n)$. In our context, it means that the backward execution of the UML$^{\text{JASA}}$-GSPN transformation does not modify any part of the UML initial model that still complies, along the specified relation, with the refactored GSPN. In other words, the transformation only modifies the portions of the UML model where refactoring patterns have been applied in the related GSPN model portions. The capability of the JTL framework to guarantee hippocraticness is discussed in [36, 49].

*Consistency*:

It is an internal property of an architectural model ensuring that the defined architecture does not contain contradicting information. In order to evaluate the consistency of the refactored UML model, let us consider the following aspects:

- We assumed that the initial software architectural model is consistent.
- Examples of inconsistencies are inconsistent names, interfaces, and refinements of architectural elements. The UML$^{\text{JASA}}$-GSPN bidirectional transformation specifies the mapping between UML and GSPN elements by preserving the consistency of names and structure (e.g., in the GPSN models the same names are used for the corresponding elements). On our example application, indeed, we observed that the generated architecture does not contain information that contradicts the initial one.
- Finally, the JTL framework helps in guaranteeing this property. In fact, the invertibility of a transformation can be severely affected in case of partial transformations that do not cover all the concepts. The consequent information loss may give place to unwanted behavior when the transformation is reversed. The traceability engine of JTL is able to preserve the missing information and restore it, thus avoiding loss of information [50].

### 4.3.7 RQ3: Pattern selection for availability improvements

It is obvious that the application of any fault-tolerance pattern should improve the system availability, as it will be shown and discussed in Table 4.5. It is, instead, less obvious to identify the patterns that more effectively improve the system availability when applied to specific components within defined scenarios.

This research question aims at addressing such issue, by showing the effects on the system availability of the application of fault tolerance patterns to different components in different scenarios.

We define the following notation for the remaining of this section. We denote by: $\mathfrak{A}_0$ an initial architectural model; $r_{ftp}(C)$ a single refactoring action, which consists in applying a single fault tolerance pattern $ftp$ to a specific component $C$; $R$ a refactoring strategy, that is the joint application of multiple $r_{ftp}$ actions to specific components ($R = \{r_{ftp}(C)\}$). A refactoring application obviously leads to a refactored architecture $\mathfrak{A}'$, namely: $R(\mathfrak{A}_0) \rightarrow \mathfrak{A}'$.

The system availability will be denoted by $Avail$, and it is intended to be computed on a specific architecture $\mathfrak{A}$, in the context of a specific execution scenario denoted by

$ES^x$ (where $x$ is the scenario name, e.g., $MT$ stands for *Managing Temperature* in our example application), while varying the failure probability ($FP_I^y$) of the architectural component $y$ within the range $I$.

We start by investigating how changes in failure probabilities affect the improvements introduced by the application of the fault tolerance patterns in a specific execution scenario. Figure 4.17 shows how the steady state availability of the *Managing Temperature* scenario ($ES^{MT}$) is altered when varying the failure probability of the *TemperatureSensor* component ($FP^{TS}$) in the interval $[0.01, 0.5]$.

The figure shows the availability $Avail(\mathfrak{A}_i, ES^{MT}, FP^{TS}_{[0.01,0.5]})$ computed for five alternative architectures:

i) the initial architecture $\mathfrak{A}_0$, in red, on which no refactoring action is applied;

ii) the architecture $\mathfrak{A}_1$, in heavy green, on which the refactoring action $r_{SAR}$ (i.e., *Semi-Active Replication* pattern) is applied on the *TemperatureSensor* component (i.e., $R(\mathfrak{A}_0) \rightarrow \mathfrak{A}_1$, where $R = \{r_{SAR}(TS)\}$);

iii) the architecture $\mathfrak{A}_2$, in light green, on which the refactoring action $r_{AR}$ (i.e., *Active Replication* pattern) is applied on the *TemperatureSensor* component (i.e., $R(\mathfrak{A}_0) \rightarrow \mathfrak{A}_2$, where $R = \{r_{AR}(TS)\}$);

iv) the architecture $\mathfrak{A}_3$, in heavy blue, on which the refactoring action $r_{SPR}$ (i.e., *Semi-Passive Replication* pattern) is applied on the *TemperatureSensor* component (i.e., $R(\mathfrak{A}_0) \rightarrow \mathfrak{A}_3$, where $R = \{r_{SPR}(TS)\}$);

v) the architecture $\mathfrak{A}_4$, in light blue, on which the refactoring action $r_{PR}$ (i.e., *Passive Replication* pattern) is applied on the *TemperatureSensor* component (i.e., $R(\mathfrak{A}_0) \rightarrow \mathfrak{A}_4$, where $R = \{r_{PR}(TS)\}$).

The results show that, while the *Active Replication* and *Semi-Active Replication* patterns perform better with small failure probabilities values, the *Passive Replication* and *Semi-Passive Replication* patterns are more robust to an increase in the failure probability of the components they are applied on. This figure shows how our approach can support the designer decisions to identify the best refactoring actions with respect to the variation of system parameters. More specifically, in this case the *Semi-Active Replication* pattern appears to be the best choice when the failure probability value of *TemperatureSensor* is within the range $[0.01, 0.115]$, whereas, for higher values, *Passive Replication* pattern should be preferred.

In order to move from single refactoring actions to combined ones, for each considered scenario, we have first measured the availability on the GPSN model before and

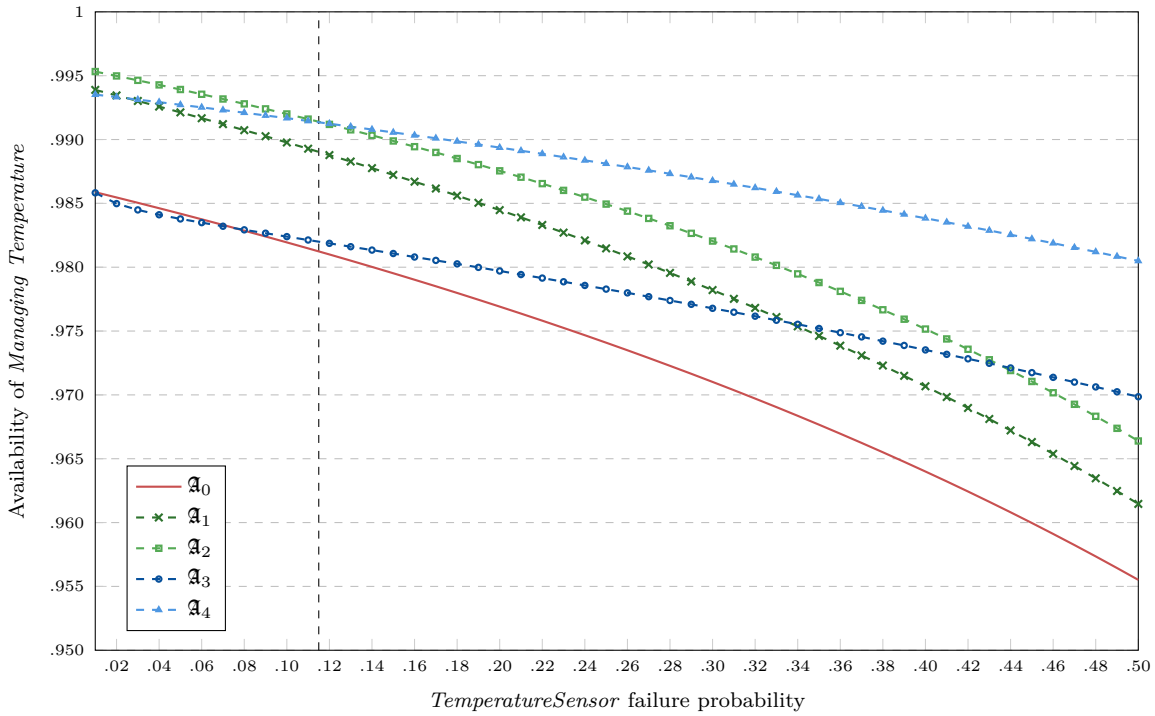



Fig. 4.17 Availability of *Managing Temperature* scenario vs. *TemperatureSensor* failure probability under single refactoring actions.

after applying the refactoring changes mentioned at the end of Section 4.3.3, namely $R(\mathfrak{A}_0) \rightarrow \mathfrak{A}'$, where $R = \{r_{SAR}(TS), r_{SAR}(HS), r_{SAR}(SHS), r_{AR}(DB)\}$. The observed steady state availability indexes resulting from the analysis are reported in Table 4.5. The availability is computed on the *Monitoring Conditions* ($ES^{MC}$), *Remote Monitoring* ($ES^{RM}$), and *Managing Temperature* ($ES^{MT}$) scenarios by considering the specific failure probabilities reported in Table 4.1. The measures highlight that the application of the fault tolerance patterns has improved, as expected, the availability in each considered scenario.

| | $\mathfrak{a}_0$ | $\mathfrak{A}'$ |
|---|---|---|
| Monitoring Conditions | 0.985392 | 0.990771 |
| Remote Monitoring | 0.991672 | 0.994207 |
| Managing Temperature | 0.977984 | 0.993316 |

Table 4.5 Steady state availability computed on the initial ($\mathfrak{A}_0$) and refactored ($\mathfrak{A}'$) architecture

The *Managing Temperature* scenario had an improvement of $15.332 \times 10^{-3}$ after the application of the *Semi-Active Replication* pattern on *TemperatureSensor*, and the *Active Replication* pattern on the *Database* component. The *Monitoring Conditions*

scenario had an improvement of $5.379 \times 10^{-3}$ after the application of the *Semi-Active Replication* pattern on *TemperatureSensor*, *HumiditySensor* and *SoilHumiditySensor*, and the *Active Replication* pattern on the *Database* component. Finally, the *Remote Monitoring* scenario had an improvement of $2.535 \times 10^{-3}$ after the application of the *Semi-Active Replication* pattern on *HumiditySensor*, and the *Active Replication* pattern on the *Database* component.

Then, we performed a sensitivity analysis of availability, for each considered scenario, by varying in the interval $[0.01, 0.5]$ the failure probability of each refactored component involved in the scenarios.

In what follows, we show how some changes in the failure probabilities of components affect both the initial architecture $\mathfrak{A}_0$ and the refactored architecture $\mathfrak{A}'$ obtained by applying $R$ defined above to $\mathfrak{A}_0$. In particular, Figures 4.18, 4.19, and 4.20 report the results for the *Monitoring Conditions*, *Remote Monitoring*, and *Managing Temperature* scenarios, respectively.

The curve notation is the same as for Figure 4.17. For example, in Figure 4.18 we depict $Avail(\mathfrak{A}_0, ES^{MT}, FP^{TS}_{[0.01,0.5]})$ and $Avail(\mathfrak{A}_0, ES^{MT}, FP^{DB}_{[0.01,0.5]})$ as solid curves, whereas $Avail(\mathfrak{A}', ES^{MT}, FP^{TS}_{[0.01,0.5]})$ and $Avail(\mathfrak{A}', ES^{MT}, FP^{DB}_{[0.01,0.5]})$ as dashed curves, respectively. For the other two figures, of course, the scenario and the related involved components are different. For the sake of clarity, in the legend of each figure we indicate, beside the architecture name, the involved component whose failure probability varies to obtain that specific curve.

The graphs clearly show improvements of the availability in all scenarios. Moreover, by comparing the effects of refactored components with those of original ones, we can see that, while the failure probability increases, the availability decreases more slowly after the refactoring. In other words, we can observe that the architecture $\mathfrak{A}'$ can better withstand an increase in failure probabilities than $\mathfrak{A}_0$ does.

Finally, we remark that this analysis provides further support to designers, by distinguishing the robustness of a refactoring strategy vs. failure probability variations of different components. For example, Figure 4.18 shows that $R$ is more effective on the architecture $\mathfrak{A}'$ when the *TemperatureSensor* failure probability increases with respect to when the *Database* one increases. Indeed, this effect is emphasized by the increasing distance between solid and dashed red curves, whereas the distance between orange curves remains more or less the same all across the *Database* probability failure range.

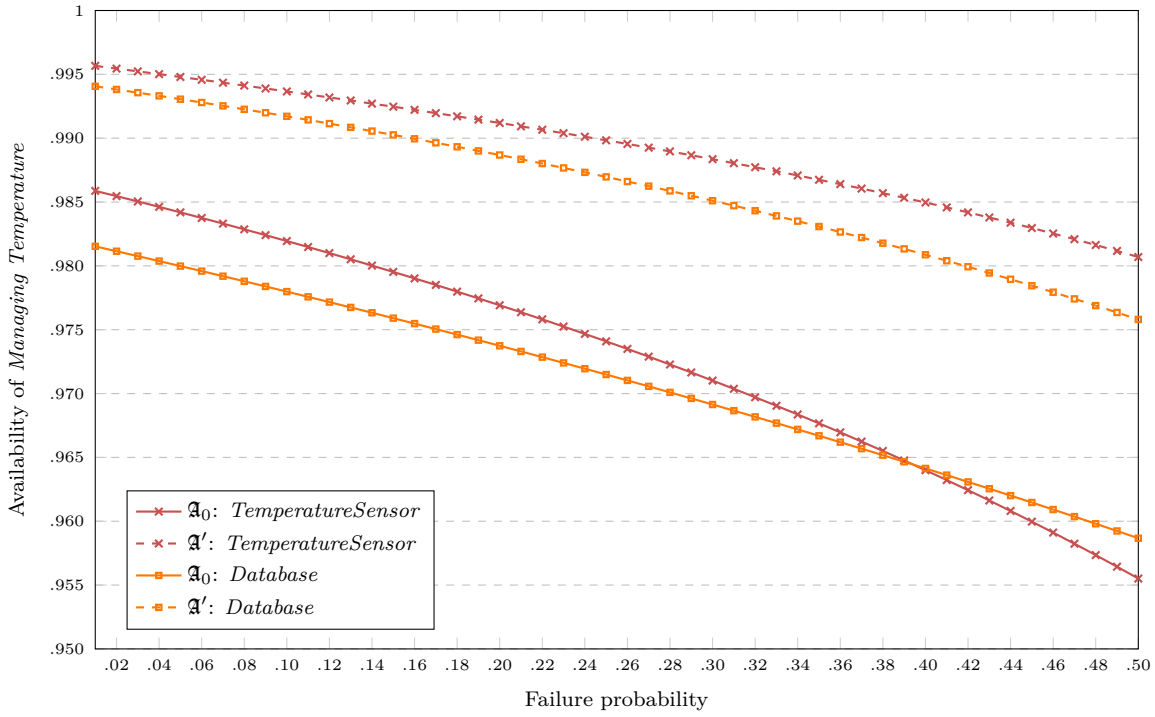



Fig. 4.18 Availability of *Managing Temperature* scenario on the initial ($\mathfrak{A}_0$) and refactored ($\mathfrak{A}'$) architecture vs. failure probabilities under combined refactoring actions.

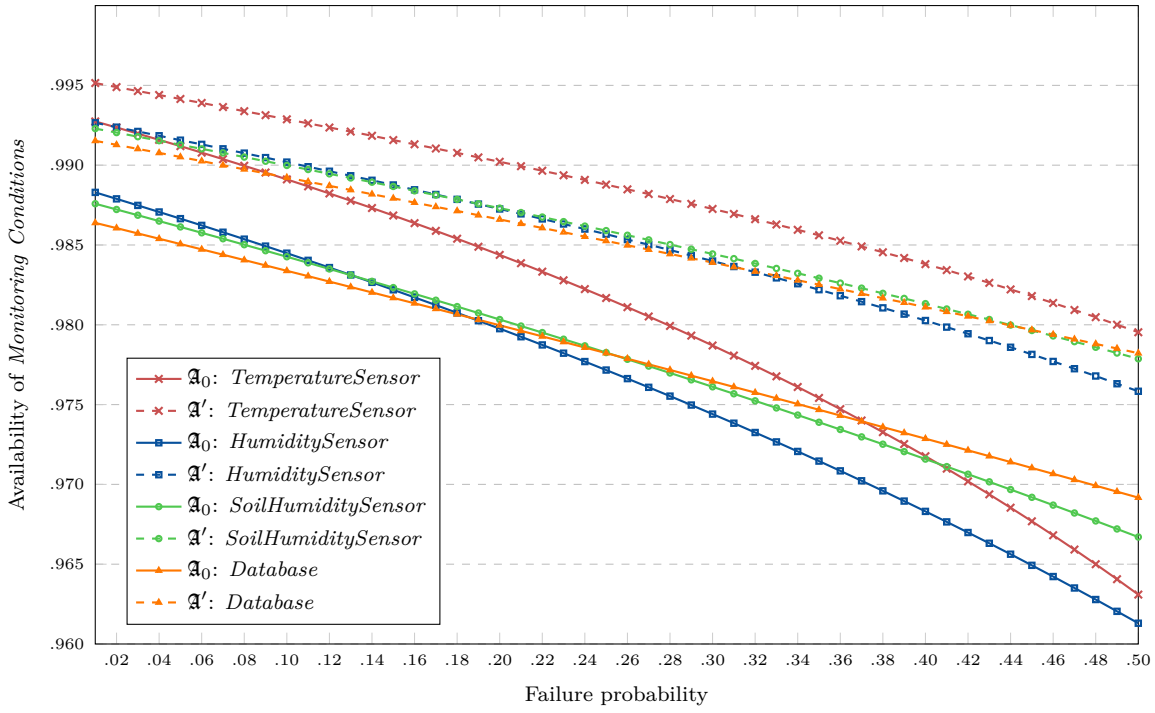



Fig. 4.19 Availability of *Monitoring Conditions* scenario on the initial ($\mathfrak{A}_0$) and refactored ($\mathfrak{A}'$) architecture vs. failure probabilities under combined refactoring actions.

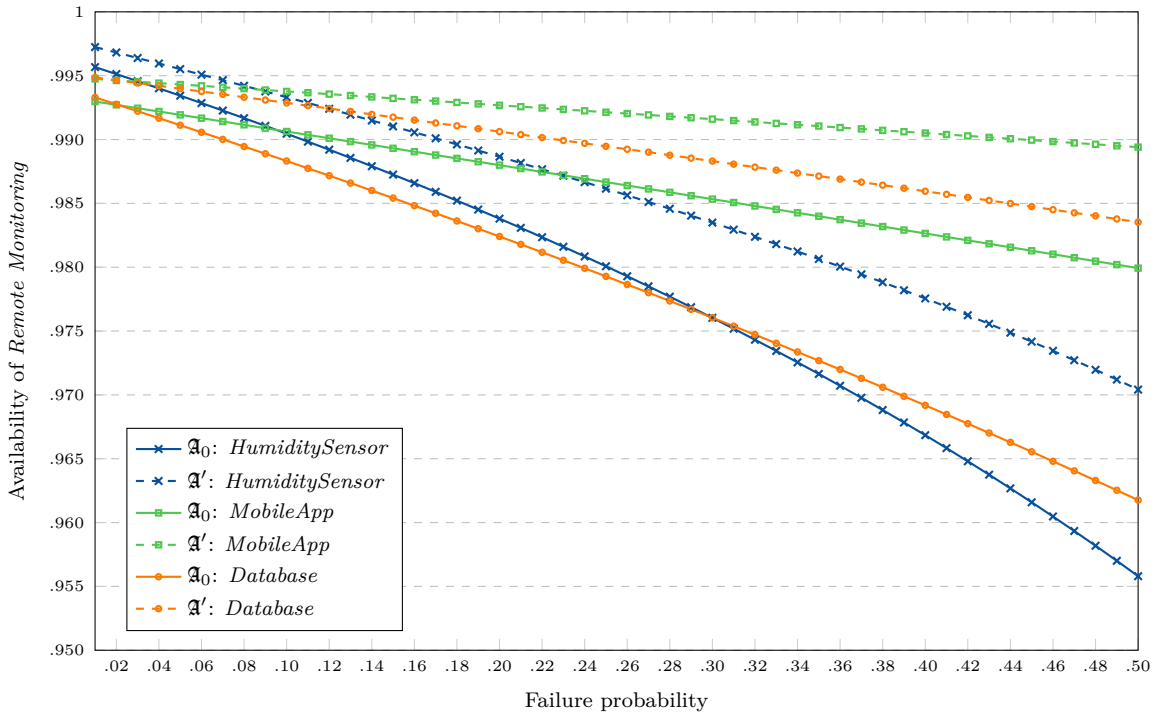



Fig. 4.20 Availability of *Remote Monitoring* scenario on the initial ($\mathfrak{A}_0$) and refactored ($\mathfrak{A}'$) architecture vs. failure probabilities under combined refactoring actions.

### 4.3.8 Threats to validity

In this section, potential threats to validity associated with the experimental evaluation are discussed, by distinguishing internal, external, construct and conclusion validity.

*Internal validity* concerns any extraneous factor that could influence our results. In general, the implementation of the approach could be defective, as well as the results of the analysis could be inaccurate. We mitigated these threats: i) by specifying our transformation on the base of already existing mapping from Sequence Diagrams and Statecharts to GSPNs [20]; ii) by considering already existing fault tolerance patterns [109]; iii) by considering well-established methods for stochastic availability assessment [60]; iv) by delegating the availability analysis to an external solver [35]. Obviously, all the above actions mitigate the possibility of introducing faults in the model transformation, because it is based on solid specifications. We recall here that, by construction, the transformation only produces, as output, models conforming to both metamodels, although we have not performed any formal proof on the semantic correctness of the results.

*External validity* refers to the generalizability of the obtained results. With reference to the model transformation, we have adopted standard metamodels, thus the approach

can be applied to any other conforming model, as long as it satisfies the design assumptions on involved diagrams, metaclasses and stereotypes. The analysis can be generalized to other models, even though the considered fault tolerance patterns obviously change their effectiveness depending on the specific software system. However, our approach can be extended to apply additional patterns at the cost of specifying them in GSPN. Finally, the size of the example application considered here is not very large, but complex enough to demonstrate the effectiveness of the approach. Nothing can be asserted about the scalability of the approach on large size architectures, which remains one of our future objectives. However, we remark that our approach is intended to be used within a decisional process that usually is not constrained by hard real-time requirements, like it could have been the assessment of availability at runtime. Hence, even several hours of processing time could represent a reasonable cost to be afforded in practice for exploring a solution space difficult to inspect without automation.

*Construct validity* concerns the validity of our results with respect to the evaluation criteria. As said, we considered well-know methodologies and methods existing in literature both for the transformation specification and the availability analysis. This mitigates the presence of factors that can compromise the validity of the experiment and of the results.

*Conclusion validity* concerns the reliability of the measures that, in this case, refers to the reproducibility of the results. In order to ensure that our results are reproducible, we repeated each measurement three times and made sure that there were no differences between the measured values with an approximation of $10^{-5}$. The artifacts considered in this experiment are supplied via a GitHub repository[5], and the experiment can be reproduced locally within the JTL framework.

## 4.4 Related work

Several approaches have been introduced in the last few years to derive analysis models from annotated software models. Bondavalli et al. [25] represents one of the first attempts at enriching a UML design to specify dependability aspects. The authors define UML extensions to generate Stochastic Petri Net models for dependability analysis automatically. High-level SPN models are derived from UML structural diagrams and later refined using UML behavioral specifications. The transformation relies upon an intermediate model, and no standard UML profiles are employed since none were available at the time of publication. Huszer et al. [66] propose a transformation of UML

[5] https://github.com/SEALABQualityGroup/JASA

statechart diagrams into Stochastic Reward Nets (SRN) to conduct a performance and dependability analysis. The transformation is defined as a set of SRN patterns, and the dependability analysis is performed under erroneous state and faulty behavior assumptions. Mustafiz et al. [94] also present a mapping between a probabilistic extension of statecharts and a Markov chain model for quantitative assessment of safety and reliability. Bernardi et al. [21] propose a transformation of UML sequence, statechart and deployment diagrams into a GSPN model for performability analysis. Software models are annotated using the former standard UML SPT profile. Our bidirectional transformation is based on the mechanisms related to statechart transformation as formally specified by Bernardi et al. [21], which we have implemented in JTL. By taking advantage of bidirectional transformations, the designer can automatically propagate the refactoring performed on the analysis model back to the UML model. In a more recent work, Bernardi et al. [19] use UML models annotated with MARTE to evaluate the performance of an industrial application. UML models are augmented with service demands obtained from execution logs, and later transformed to GSPN. The results of the simulation are presented to the user both in textual and graphical formats. The authors achieve this by means of the DICE-Simulation tool [58], an Eclipse-based tool that can generate Petri nets in the GreatSPN format from annotated UML models, and simulate them to assess performance and reliability. Gómez et al. [59] use QVTo to transform UML models annotated with MARTE to S-PMIF+. The UML models involved in the transformation are Deployment, Sequence, Class, and Composite Structure Diagrams. The authors prefer to employ an imperative transformation approach as they find it more suitable for the processing of ordered elements, like the elements in a Sequence Diagram.

On top of automated derivation of analysis models from software models, several approaches have been built for multi-objective software architecture optimization driven by non-functional attributes. None of these approaches explicitly consider availability as a target, even though some of them consider failure probabilities of components and/or platform devices.

In particular, Martens et al. [89] introduce an evolutionary algorithm for optimizing performance, reliability and cost. Failure probabilities are associated to hardware connectors only, and a discrete-time Markov chain is generated to calculate the probability for the whole system to be in a failure state. Hence, this approach considers different model elements to be subject to failures, as well as a different non-functional target property with respect to our work. Moreover, the architecture refactoring actions are not specifically targeted to fault tolerance as in our case, but rather generic refactoring

actions, such as component replication. These differences about target properties and non-specific fault tolerant actions remain in other similar works that have appeared in the context of architecture optimization, such as Cardellini et al. [33].

In the context of bidirectional model transformations, Hettel et al. [63] formally define a round-trip engineering process between models representing different views of the same system. In the performance analysis domain, in a previous paper [48], we have introduced a similar approach to the one presented in this chapter. In particular, we have defined a bidirectional model transformation between UML software models and Queueing Network (QN) performance models. The forward transformation path generates the performance model from the initial software model, whereas the backward one is used to generate, after the analysis, a new software model from the modified version of the performance model. Arcelli et al. [10] compare two methods to tackle the problem of deriving architectural changes from model-based performance analysis results: (i) to perform refactoring on the software side by detecting and solving performance antipatterns, or (ii) to modify the analysis model using bidirectional model transformations to induce architectural changes. This represents an interesting study for reasoning on the pros and cons of modifying a non-functional model as opposite to applying modifications to a software architectural model.

Tatibouet et al. [117] propose principles to use fUML (Foundational Semantics for Executable UML Models) and Alf (Action Language for fUML) as a simulation environment. However, this approach provides only the structural modeling constructs of UML, whereas the ability to model behavior is limited to UML activities. Hence, in order to exploit the simulation environment, availability parameters (such as the failure probabilities) should be defined within the modeling language and the simulation engine could require to be extended to process them. As opposite, the use of languages as DAM that natively supports the definition of dependability parameters, coupled with transformations towards analysis models like GSPNs, does not require to extend the modeling language and the solution/simulation engine. Finally, this process would be subject to scalability problems [18, 17].

To the best of our knowledge, this is the first approach proposing an automated propagation of changes performed on an availability model back to an architectural model. Even though the scope of this chapter is limited to the modeling notation context considered here (i.e., UML-DAM and GSPNs), our approach represents a first step towards the usage of bidirectional transformations for closing a round-trip process for software availability modeling and analysis.

# Chapter 5

# Software Performance Assessment and Improvement

Microservices have become a popular style for architecting a software system as a suite of small services, and they are nowadays adopted by many key technological players such as Netflix, Amazon, and Google. Major benefits of a microservice-based architecture are that it ensures loose coupling, and it supports rapid evolution and continuous deployment. In addition, having a large set of independently developed services helps in terms of developer productivity, scalability, and maintainability. Contextually, the rapidly growing complexity of software systems has forced practitioners to use and investigate different development techniques to tackle advances in productivity and quality. To this extent, software engineering needs to rely on automated approaches to keep low the development costs while tackling the rapid changes of software capabilities that may considerably impact non-functional properties like performance.

In this chapter, we introduce a model-driven approach to realize a continuous software engineering loop in microservice-based systems. The approach exploits design-runtime interactions to support designers of microservices in performance analysis and system refactoring tasks. In particular, we use distributed tracing to maintain a central log of microservices interactions and metrics. The observed system behavior at runtime is then related to the architectural design to investigate potential performance issues and to design and implement effective system refactoring actions.

The approach takes advantage of specific characteristics of these systems. One of the main characteristis is that microservices are autonomous entities which can change independently of each other. This allows us to make a change to a single service and deploy it independently of the rest of the system. In contrast, in monolithic applications, any refactoring would impact all the interconnected parts of the system,

requiring additional coordination among components when making changes. Also, in order to release changes, monolithic applications usually need to be redeployed entirely, imposing a higher management delta between releases and a higher risk of malfunctioning [95].

The approach employs JTL to generate traceability models between an architectural model designed in UML and the logs of the corresponding running system. The traceability links are used to annotate the architectural model (profiled with MARTE) with performance metrics like response time and utilization. These annotations are then used to detect performance antipatterns on the basis of which the approach recommends refactoring actions.

We show how the approach can be applied to two case studies and we evaluate the performance improvement induced by the application of the recommended refactoring actions on the running systems.

## 5.1 Background

In the following, we describe some additional background in terms of existing techniques that have been adopted to realize the approach.

### 5.1.1 Monitoring techniques

One of the defining characteristics of microservice-based systems is that each service must be independently deployable [34]. This aspect favors the independent development of services, but it also makes traditional application monitoring insufficient. While suitable for monolithic applications, the gathering of logs and metrics of each service does not provide a complete understanding of the system behavior. This is the main reason for the adoption of distributed tracing as a mean to correlate events generated in individual services with a transaction traversing the entire system.

In this chapter, we focus on microservices applications deployed on *Docker*[1] and developed with *Spring Boot*[2] and *Spring Cloud*[3]. As a consequence, we chose *Spring Cloud Sleuth*[4] to implement a distributed tracing solution. In *Spring Cloud Sleuth*, a trace consists of a series of casually related events that are triggered by a request as it moves through a distributed system. These events are called spans and they represent

[1] https://www.docker.com/
[2] https://spring.io/projects/spring-boot
[3] https://spring.io/projects/spring-cloud
[4] https://spring.io/projects/spring-cloud-sleuth

a timed operation occurring in a component. Spans contain references to other spans, which allow a trace to be assembled as a complete workflow. A span contains a set of basic information: the name of the operation, the name of the component providing the operation, the start timestamp and duration (or, alternatively, the finish timestamp), the role of the span in the request and a set of user-defined annotations called tags. Besides basic information, spans generated by *Spring Cloud Sleuth* also contain the IP address and port number of the service, the Java class and method implementing the operation, as well as the unique identifier of the *Spring Cloud* instance.

Once the application is instrumented to produce traces, an infrastructure is necessary to collect and store them. The traces produced by each service during the execution are gathered by the *Zipkin*[5] distributed tracing system. In turn, *Zipkin* is configured to forward the monitoring data to the distributed database and search engine *Elasticsearch*[6].

The resources utilization of individual microservices is another important aspect to consider when monitoring system performance. For this approach, we selected *perf*[7] among the wide range of performance analyzing tools. *Perf* is one of the most commonly used performance counter profiling tools on Linux and supports hardware and software performance counters, tracepoints and dynamic probes. We used *perf* to gather accurate CPU utilization for each microservice. We stored such measures in *Elasticsearch*, along with the traces collected by *Zipkin*.

### 5.1.2 PADRE

As part of our approach, we used PADRE to conduct the performance analysis and realize the model refactoring. PADRE (Performance Antipatterns Detection and model REfactoring), a proposal by Arcelli et al. [11], is a unified framework that tries to improve the performance quality of UML models through a performance antipatterns detection and a model-based refactoring engines, which exploit Epsilon [76] to implement detection rules and refactoring actions. Furthermore, PADRE employs UML models augmented by MARTE stereotypes in order to link performance data to UML elements. A performance antipattern [40] is a description of well-known bad design practices that might lead to performance degradation.

Moreover, PADRE is equipped with a performance analyzer, which exploits Queueing Networks and an MVA approximation algorithm to obtain performance indices.

[5] https://zipkin.io/
[6] https://www.elastic.co/products/elasticsearch
[7] https://perf.wiki.kernel.org/

In particular, PADRE can detect eight performance antipatterns, and it provides several refactoring actions. A refactoring action can be either a specific action, i.e., designed to remove specific antipatterns, or a general one, i.e., aimed at improving the system performance quality without targeting specific aspects. PADRE provides three different detection and refactoring sessions, namely user-driven, batch and multiple sessions. The multiple sessions allows applying more than one refactoring action in a row, the batch session performs refactoring actions until every performance antipattern has been removed, and the users-driven allows the performance expert to select a specific performance antipattern occurrence to be removed. Hereby we employ the user-driver session having the performance expert as part of our refactoring loop.

### 5.1.3 Performance antipatterns

In this section we introduce the performance antipattern (PA) concept. A performance antipattern describes a bad design practice that might lead to performance degradation in a system. This concept is mutuated from the design antipatterns one, which describes well-known bad practices that might cause system quality degradation (e.g., low cohesion).

Smith and Williams [113] had textually described a set of performance antipatterns (PA) that they have identified on the basis of existing experiences. Then, this textual descriptions have been translated in first-order logics representation [40], thus enabling the automated detection of PAs. The first-order logics representation of a PA is a combination of multiple literals, where each one maps on a specific system view. Furthermore, every literal is compared to a threshold that represents a safety limit that the system shall not overstep.

Here we consider the Blob and the Pipe and Filter performance antipatterns, because they have a larger potential to occur in microservice-based systems. In the following we recap the definition of these PAs.

**Blob** It occurs when a single component (also known as God Class) performs the most part of the work of a software system, and its manifestation results in excessive message traffic that may degrade performance. Expression 5.1 describes the Blob performance antipattern in first-order logics [40].

The first inequality of Expression 5.1 refers to the number of the exposed interfaces of a component, where a too high number of interfaces is considered as a precondition to identify the component as a God Class. The second inequality, instead, checks whether the God Class is effectively involved in the system. The third inequality

refers to hardware utilization of hardware where the component runs. Only if all three inequalities hold then the component is identified as a Blob performance antipattern.

$$\begin{gathered}\exists c_x, c_y \in \mathbb{C}, S \in \mathbb{S} \mid \\ F_{numClientConnects}(c_x) \geq Th_{maxConnects} \quad \wedge \\ F_{numMsgs}(c_x, c_y, S) \geq Th_{maxMsgs} \quad \wedge \\ F_{maxHwUtil}(P_{xy}, all) \geq Th_{maxHwUtil}\end{gathered} \tag{5.1}$$

**Pipe and Filter** It occurs when the slowest filter in a "pipe" causes the system to have unacceptable throughput. This situation is formalized in Expression 5.2 [40].

$$\begin{gathered}\exists OpI \in \mathbb{O}, S \in \mathbb{S}, i \in \mathbb{N} \mid \\ F_{resDemand}(Op) \geq Th_{resDemand} \wedge F_{probExec}(S, OpI) = 1 \quad \wedge \\ F_{maxHwUtil}(P_c, all) \geq Th_{maxHwUtil}\end{gathered} \tag{5.2}$$

Differently to the Blob performance antipattern, Pipe and Filter identifies an operation to be the cause of performance degradation. First of all, because the operation requires a too high amount of resources to be executed (i.e., a heavyweight resource demand), as described in the first inequality of Expression 5.2. Beside this, the operation has to be certainly executed in order to be the cause of performance degradation, and this is checked in the second equality. Finally, either the hardware utilization must exceed a safety threshold (i.e., third inequality) for the Pipe and Filter to manifest itself. Here the first two literals refer to design characteristics of the system, while the last two one to performance properties.

## 5.2 Approach

The idea underlying our approach exploits the correspondences between the architectural design and the runtime aspects of a software system, with the aim of improving its performance.

Figure 5.1 depicts four main steps of the continuous performance engineering loop we considered, as follows:

1. *Runtime data acquisition*: Microservice-based systems are monitored by means of distributed tracing; thus, logs are stored in a central location and metrics for all instances of a given service are aggregated to understand the overall state. Then, the collected data are represented in a model-based format compliant with EMF.

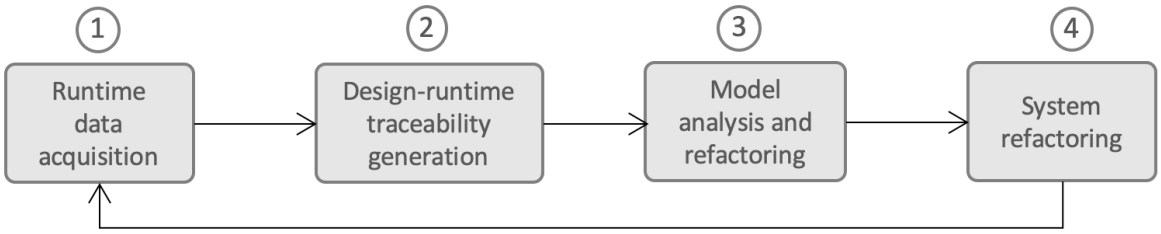



Fig. 5.1 A high-level workflow of our approach.

2. *Design-runtime traceability generation*: In this phase, the system behavior at runtime is matched with the architectural design by means of traceability models that are automatically generated on the base of correspondences that are formally pre-defined through a metamodel.

3. *Model analysis and refactoring*: The analysis of the above created traceability models aims to connect system performance issues with the affected design components that are identified as possible causes. The results of such analysis lead to the definition of model refactoring actions, that are applied on the system model in order to identify the most promising ones.

4. *System refactoring*: The emerging refactoring actions are implemented and applied to the running system, whose runtime data is acquired and used in the next iteration of this workflow.

In the rest of this section, we describe each step in detail.

### 5.2.1 Runtime data acquisition

As depicted in Figure 5.2, runtime data (i.e., logs/traces) are obtained through a monitoring infrastructure over a running system. The specific infrastructure adopted in this approach has been detailed in Section 5.1.1.

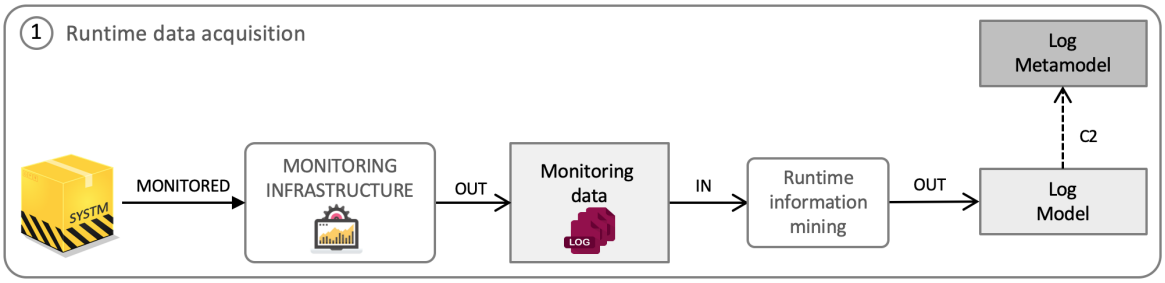



Fig. 5.2 Runtime data acquisition

traceId: 149c4cef3ac7f19f duration: 27,000 shared: true localEndpoint.serviceName: gateway localEndpoint.ipv4: 172.28.0.12 localEndpoint.port: 4000 timestamp_millis: November 20th 2018, 10:52:48.107 kind: SERVER name: http:/categories/category id: 16bb4e7b689f807a parentId: 149c4cef3ac7f19f timestamp: 1,542,707,568,107,000 tags.spring.instance_id: 002ffdb287d6:gateway:4000 _id: cE-JMGcBBzL8qQlHYHn4 _type: span _index: zipkin:span-2018-11-20 _score: -

traceId: 149c4cef3ac7f19f duration: 17,000 shared: true localEndpoint.serviceName: categories-server localEndpoint.ipv4: 172.28.0.18 localEndpoint.port: 5555 timestamp_millis: November 20th 2018, 10:52:48.115 kind: SERVER name: http:/categories/category id: 4ad2da86e8767b82 parentId: 16bb4e7b689f807a timestamp: 1,542,707,568,115,000 tags.mvc.controller.class: CategoriesController tags.mvc.controller.method: getCategory tags.spring.instance_id: 5b58aea6835e:categories-server:5555 _id: bk-JMGcBBzL8qQlHYHn2 _type: span _index: zipkin:span-2018-11-20 _score: -

traceId: 1b013fa75420bf54 duration: 6,000 shared: true localEndpoint.serviceName: gateway localEndpoint.ipv4: 172.28.0.12 localEndpoint.port: 4000 timestamp_millis: November 20th 2018, 10:52:48.976 kind: SERVER name: http:/categories/category id: 18c3f41109c8c6fd parentId: 1b013fa75420bf54 timestamp: 1,542,707,568,976,000 tags.spring.instance_id: 002ffdb287d6:gateway:4000 _id: o0-JMGcBBzL8qQlHZHkK _type: span _index: zipkin:span-2018-11-20 _score: -

traceId: cb9b9ab5d908bf18 duration: 6,000 shared: true localEndpoint.serviceName: gateway localEndpoint.ipv4: 172.28.0.12 localEndpoint.port: 4000 timestamp_millis: November 20th 2018, 10:52:50.500 kind: SERVER name: http:/categories/category id: 6850d3b3195db041 parentId: cb9b9ab5d908bf18 timestamp: 1,542,707,570,500,000 tags.spring.instance_id: 002ffdb287d6:gateway:4000 _id: 1k-JMGcBBzL8qQlHZ3n1 _type: span _index: zipkin:span-2018-11-20 _score: -

traceId: cb9b9ab5d908bf18 duration: 4,000 shared: true localEndpoint.serviceName: categories-server localEndpoint.ipv4: 172.28.0.18 localEndpoint.port: 5555 timestamp_millis: November 20th 2018, 10:52:50.501 kind: SERVER name: http:/categories/category id: 848f713a3fc3a108 parentId: 6850d3b3195db041 timestamp: 1,542,707,570,501,000 tags.mvc.controller.class: CategoriesController tags.mvc.controller.method: getCategory tags.spring.instance_id: 5b58aea6835e:categories-server:5555 _id: 1E-JMGcBBzL8qQlHZ3ny _type: span _index: zipkin:span-2018-11-20 _score: -

Fig. 5.3 A fragment of the raw log

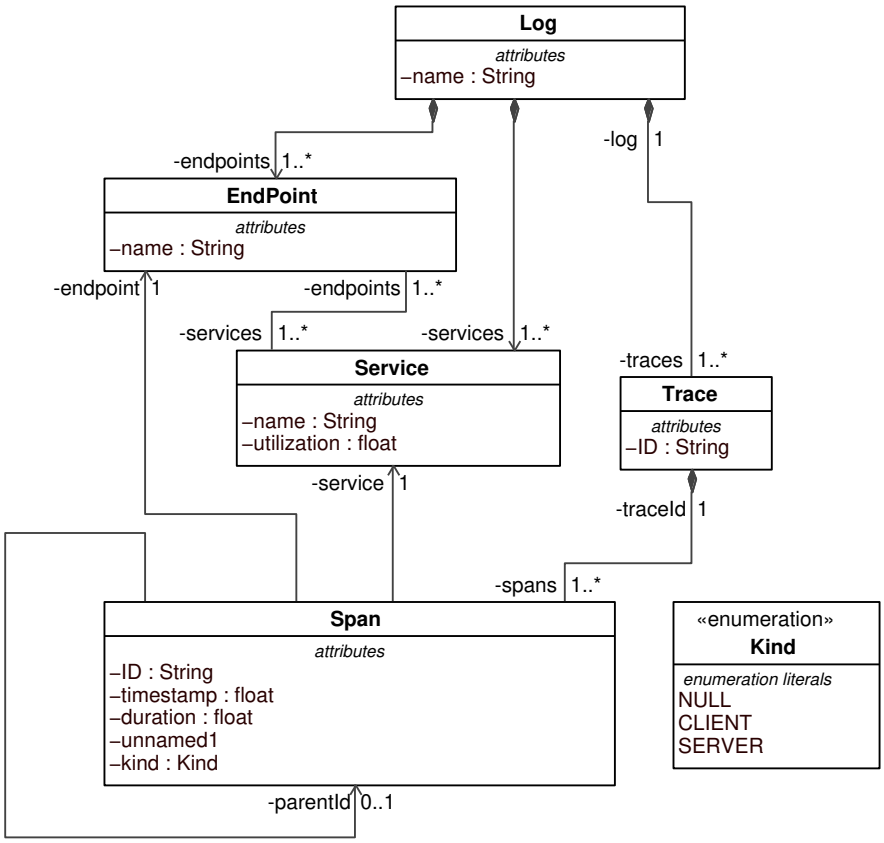



Fig. 5.4 Log Metamodel

The collected runtime data is then integrated in an EMF-based environment and translated in EMF artifacts. In this step, raw logs, as the one shown in Figure 5.3, are automatically transformed in Log Models conforming to a specific Log Metamodel reported in Figure 5.4.

The Log Metamodel defines the characteristics of a Log element, which is the root of a log model. A Log stores all the Trace information about requests being sent to EndPoints. Service elements, that may represent the microservices of an application, are associated to both Spans and EndPoints. Services also include a *utilization* attribute setting the percentage of CPU usage during the observation time. A Trace is identified by a unique ID and includes a set of Spans representing execution events. A Span is defined by the following attributes: timestamp describing when the event occurs, *duration* describing the time to complete the call, and kind that may be one of SERVER, CLIENT or UNDEFINED. Moreover, when a Span is triggered by another one, the parentId reference connects the triggered Span to the triggering one, called the parent Span. A Span also refers to an EndPoint, which is the URL used to perform a request.

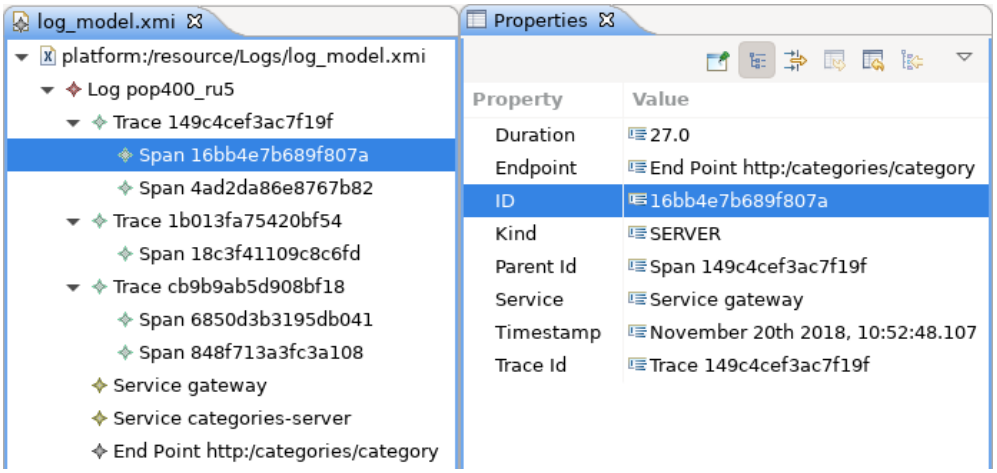



Fig. 5.5 A Log Model sample in Eclipse

Figure 5.5 depicts a sample of a Log Model that represents the original logs shown in Figure 5.3, where the information that is negligible for our purposes has not been included. For instance, the topmost Span (id *16bb4e7b689f807a*) represents the first span in Figure 5.3 with a *27ms* duration, of *SERVER* kind, and with the *November 20th 2018 10:52:48.107* timestamp for the call to the *http://categories/category* EndPoint belonging to the *gateway* Service. Such a model is automatically generated from the original raw log by means of a Java transformation able to serialize the textual representation of the logs into XMI-encoded models conforming to the Log Metamodel.

### 5.2.2 Design-runtime traceability generation

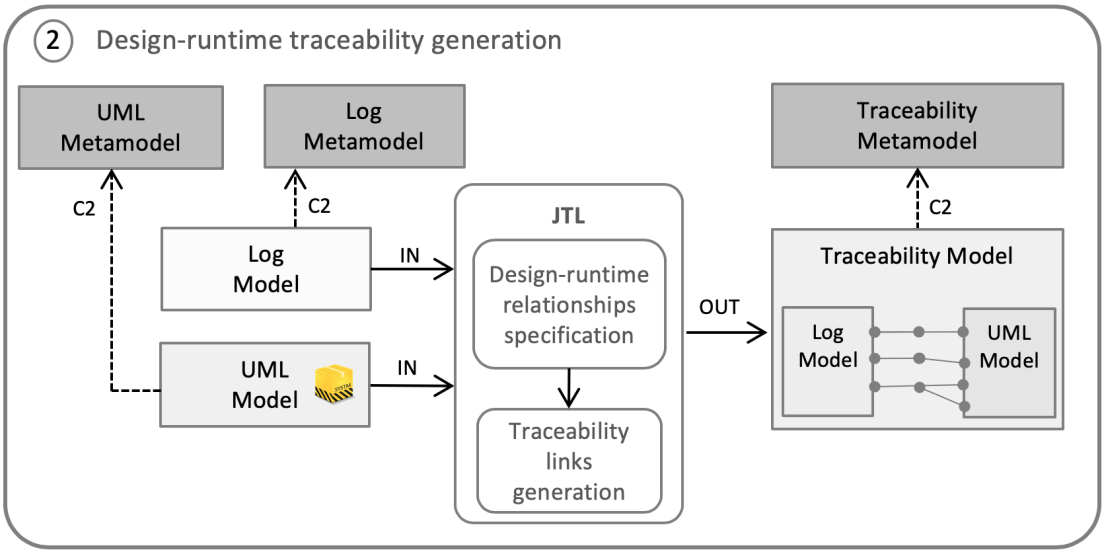



Fig. 5.6 Design-runtime traceability generation

In this phase, as depicted in Figure 5.6, the correspondences between the system behavior at runtime and the architectural design are defined and generated. Specifically, we generate traceability links between UML and Log Models by means of JTL. The JTL traceability engine is able to execute such bidirectional model transformations and automatically generate the corresponding traceability links between elements of the UML design model and the log model ones.

In Listing 5.1 we show an excerpt of JTL-defined correspondences between the design and runtime concepts. While runtime concepts are represented by means of elements belonging to the Log metamodel (as explained in Section 5.2.1), software design concepts are represented by elements belonging to the UML metamodel. In particular, for behavioral aspects we target elements of Sequence Diagrams, whereas for static aspects we target Use Case, Component, and Deployment Diagrams.

The specification is defined by means of relations between elements of the two involved metamodels. UML Use Cases are related to monitoring Traces, since they represent executions of the system. UML Messages in Sequence Diagrams are related to monitored Spans, as both represent operations occurring in a scenario. In order to map an Operation invoked by a Message to a specific API EndPoint, we relate an EndPoint of a Span to a Signature of a Message. Finally, since microservices are modeled as UML components, we relate them to Services of Spans. The above described mappings

can be specified in a declarative manner as correspondences in JTL, as described in the following.

In Line 1, variables log and uml are declared to match models conforming to the Log and UML metamodels, respectively. The specified relations are described as follows:

- the top relation Trace2UseCase (Lines 3-13) maps a container element of Trace type in the Log domain, and a container element of UseCase type in the UML domain. The where clause invokes the execution of the Span2Message relation;
- the Span2Message relation (Lines 15-23) maps a Span and a Message type elements involved in a use case interaction. The where clause invokes the execution of the EndPoint2Signature relation;
- the EndPoint2Signature relation (Lines 25-33) maps an EndPoint of a Span and an Operation type element that represents the signature of a message;
- the top relation Service2Component (Lines 35-43) maps a Service type container element to a Component type one.

```
1 transformation Log2UML (log:Log, uml:UML) {
2   ...
3   top relation Trace2UseCase {
4     checkonly domain log t : Log::Trace {
5       spans = s : Log::Span { }
6     };
7     checkonly domain uml uc : UML::UseCase {
8       ownedBehavior = ob : UML::Interaction {
9         message = m : UML::Message { }
10      }
11    };
12    where { Span2Message(s, m); }
13  }
14
15  relation Span2Message {
16    checkonly domain log s : Log::Span {
17      endpoint = ep : Log::EndPoint { }
18    };
19    checkonly domain uml m : UML::Message {
20      signature = s : UML::Operation { }
21    };
22    where { EndPoint2Signature(ep, s); }
23  }
24
```

```
25   relation EndPoint2Signature {
26     n : String;
27     checkonly domain log ep : Log::EndPoint {
28       name = n
29     };
30     checkonly domain uml s : UML::Operation {
31       name = n
32     };
33   }
34
35   top relation Service2Component {
36     n : String;
37     checkonly domain log s : Log::Service {
38       name = n
39     };
40     checkonly domain uml c : UML::Component {
41       name = n
42     };
43   }
44   ...
45 }
```

Listing 5.1 Log2UML correspondences specification

The described mapping assumes that the design of the system is consistent with its implementation (e.g., in terms of naming convention used). Moreover, the above described correspondences are specified according to the adopted notations. However, the approach can be extended to different modeling languages or monitoring technologies. In fact, JTL allows the specification of heterogeneous relations with different level of complexity, e.g., elements that do not trivially match by names, or relations between elements with one-to-many multiplicity [36].

The application of the *Log2UML* transformation on a pair of Log and UML models, as shown in the left and right part of Figure 5.7, generates the corresponding Traceability model in the middle part of the figure. In particular, the arrows in the figure cross trace links that connect the source and target model elements they refer to.

For instance, the *Trace2UseCase_149c4cef3ac7f19f* traceability link relates the *Get HomePage* use case in the right end and the corresponding *149c4cef3ac7f19f* log trace in the left end. Hence, for each message in the use case, we are able to know when the corresponding operation has started and its response time. As a consequence, the traceability model can be used to map complex performance measures, such as the

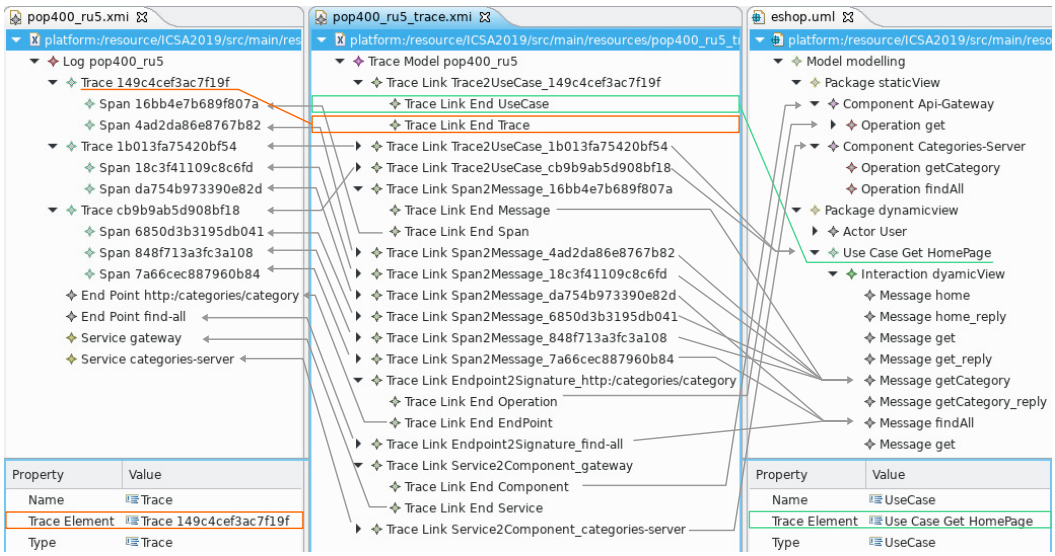



Fig. 5.7 An example of Traceability model between Log and UML models

average response time of a specific scenario or the average service time of an operation. This process will be described in detail in the next section.

### 5.2.3 Model analysis and refactoring

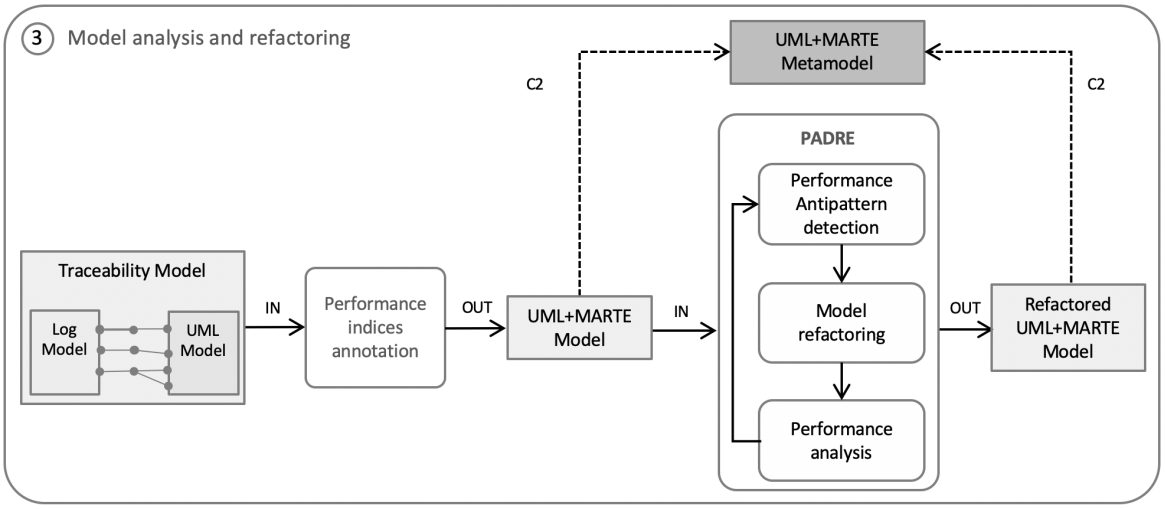



Fig. 5.8 Model analysis and refactoring

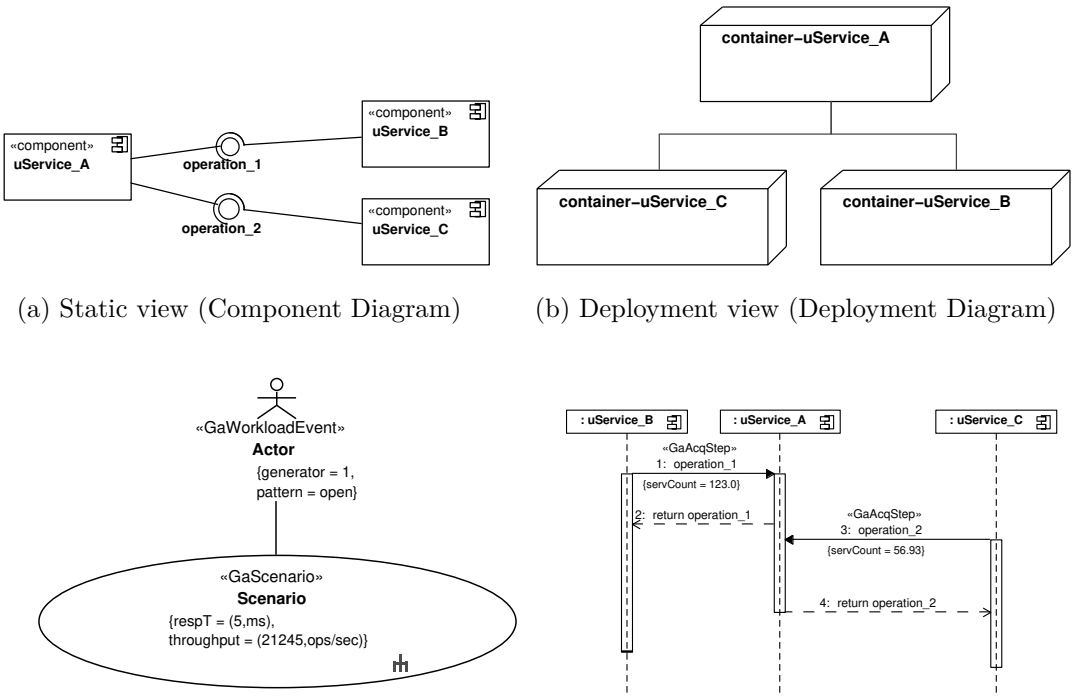
This step is aimed at analyzing the design model and removing possible performance flaws. In particular, we use runtime data (i.e., traces) to augment the design model, and then we execute a performance analysis driven by antipatterns [39] on the augmented model.

The *Performance indices annotator* in Figure 5.8 exploits the Traceability Model to report runtime data back to the model, and for this goal it exploits the MARTE stereotypes, in that they allow: i) to set input data (i.e., performance parameters) of performance analysis (e.g., operations service demand), and ii) to fill the output data (i.e., performance indices) of a performance analysis back to the model (e.g., utilization).

We adopt the *MARTE:GQAM* package stereotypes, among MARTE ones, as follows:

- *Input* Data:
  - *GaWorkloadEvent:generator*: it expresses the generational value of a workload. For example, we annotate here the exponential arrival rate $\lambda$ for an open class of jobs.
  - *GaWorkloadEvent:pattern*: it denotes if the job class is open or closed;
  - *GaAcqStep:servCount*: it expresses the service demand of a UML Operation. Hence, we annotate the UML Message that trigger that Operation in a UML Sequence Diagram to express its demand in that scenario;
- *Output* Data:
  - *GaScenario:respT*: it expresses a response time. We use it on a UML Use Case to report the response time of that scenario under a specific workload.
  - *GaScenario:throughput*: it expresses a throughput. We use it on a UML Use Case similarly to a response time;
  - *GaExecHost:utilization*: it expresses an utilization. We use it for a UML Node within a Deployment Diagram.

In order to collect operation service demands, we stimulate the system with a lightweight workload. Indeed, in this parameterization step we aim to avoid generating queues in the system, so that the *Service Demand* $D = V * S$ definition holds [81]. In particular, $D$ is the Service Demand, $V$ is the number of visits, and $S$ is the service time. Following the definition, the service demand of an operation, in our experimentation, is given by its response time when the lightweight workload is executed, because we guarantee (by observation) that waiting time in queue is never originated by that workload.




(a) Static view (Component Diagram) (b) Deployment view (Deployment Diagram)

(c) Static view (Use Case Diagram) (d) Dynamic view (Sequence Diagram)

Fig. 5.9 The *example* UML Software Model

Once *Service Demand*s have been collected and filled back to the input *GaAcqStep:servCount* tags, we stimulate the system with a selected (possibly heavy and realistic) workload to discover performance flaws.

Upon the system execution is completed under the selected workload, the *Performance Indices Annotator* fills the performance indices tags back to the model. Then, a performance-driven refactoring loop can start. PADRE [11] has been adopted for this goal, as it is an approach that detects performance antipatterns and provides a list of possible refactoring actions that shall mitigate the performance degradation.

The PADRE refactoring loop is made of three main steps: i) *Performance Antipattern Detection*; ii) *Model Refactoring*; iii) *Performance Analysis*. First, the *Performance Antipattern Detection* is executed in order to detect performance antipatterns occurrences. It is worth noticing that PADRE employs multi-views models to discover performance antipatterns. For this reason, in our experimentation, we use a multi-view UML design model, as depicted in Figure 5.9. In case performance antipatterns arise in the model, a *Model Refactoring* step is performed in order to remove them. In this step, PADRE provides a list of possible refactoring actions for each performance

antipattern.[8] While executing one refactoring action at a time, the refactored design model is given as input to the PADRE *Performance analysis* step, in which a model transformation is executed to transform the UML-MARTE design model into a closed Queueing Model [39]. Thereafter, the Queueing Model is solved through the *Mean-Value Analysis (MVA)* algorithm [106], which allows to rapidly carry out performance indices. The latter ones are exploited to recognize whether the refactoring action is promising or not.

Within this approach, we have restricted the PADRE refactoring actions portfolio to the actions that we found more appropriate for a microservice context, namely:

- Clone refactoring:

  The clone refactoring action is aimed at introducing a replica of a microservice. In our modeling assumptions, we consider a microservice as a *UML Component*, and a docker container as a *UML Node*. The action at a glance is shown in Figure 5.10.

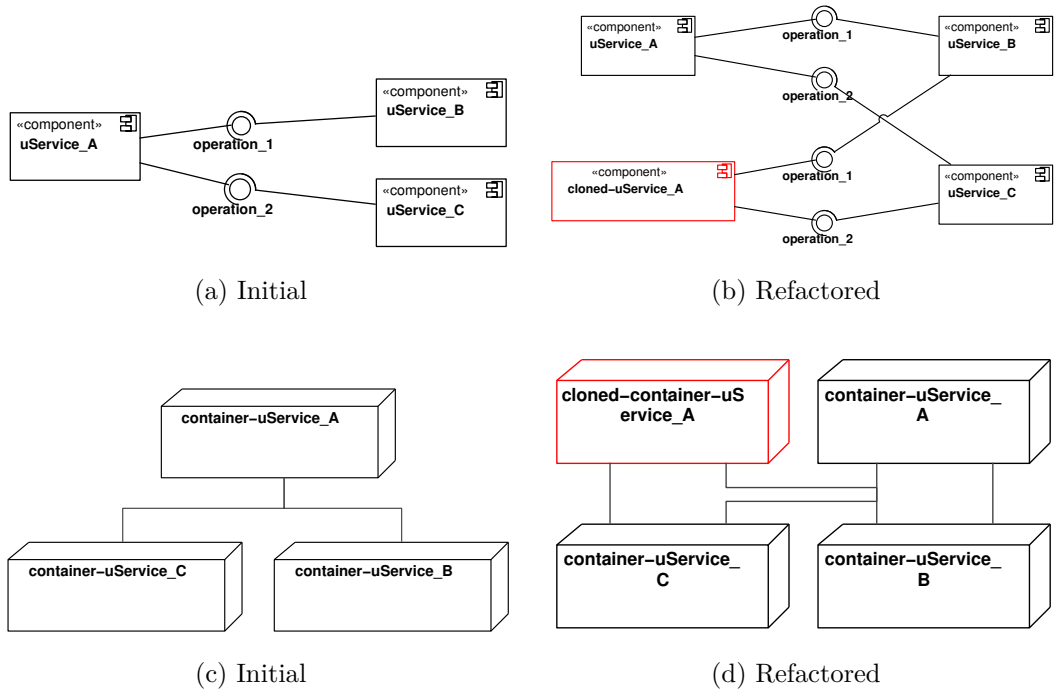
  Figure 5.10a and 5.10c depict the initial design model, while Figure 5.10b, and 5.10d show the refactored design model. The clone refactoring action in a nutshell: i) creates a new UML Component (i.e., *cloned-uService_A*), and ii) creates a new UML Node (i.e., *cloned-container-uService_A*) on which the replica is deployed. In this particular case, the dynamic view is not depicted because the refactoring action does not affect it.

- Move operation refactoring:

  The move operation refactoring action is aimed at moving a "critical" operation (e.g., due to its Service Demand) to a new microservice. Figures 5.11a, 5.11c and 5.11e depict the initial design model example, while Figures 5.11b, 5.11d and 5.11f show the refactored version. In the example, we move *operation_2* of *uService_A* microservice. Thus, the action creates a replica of this microservice (i.e., *cloned-uService_A*) and then it changes the behavior and deployment views, respectively.

  Differently to the Clone refactoring action, the move operation involves the dynamic view. Therefore, a new UML Lifeline (i.e., *cloned-uService_A*) representing the replicated microservice is created (see Figure 5.11d). Then, every message referring to operation_2 is now transferred towards the new lifeline. Furthermore, the Deployment Diagram is refactored as well. A new UML Node (i.e.,

[8] The complete PADRE refactoring actions portfolio is described in [11]




(a) Initial (b) Refactored

(c) Initial (d) Refactored

Fig. 5.10 The *clone* refactoring action example on component *uService_A* through a UML Software Model.

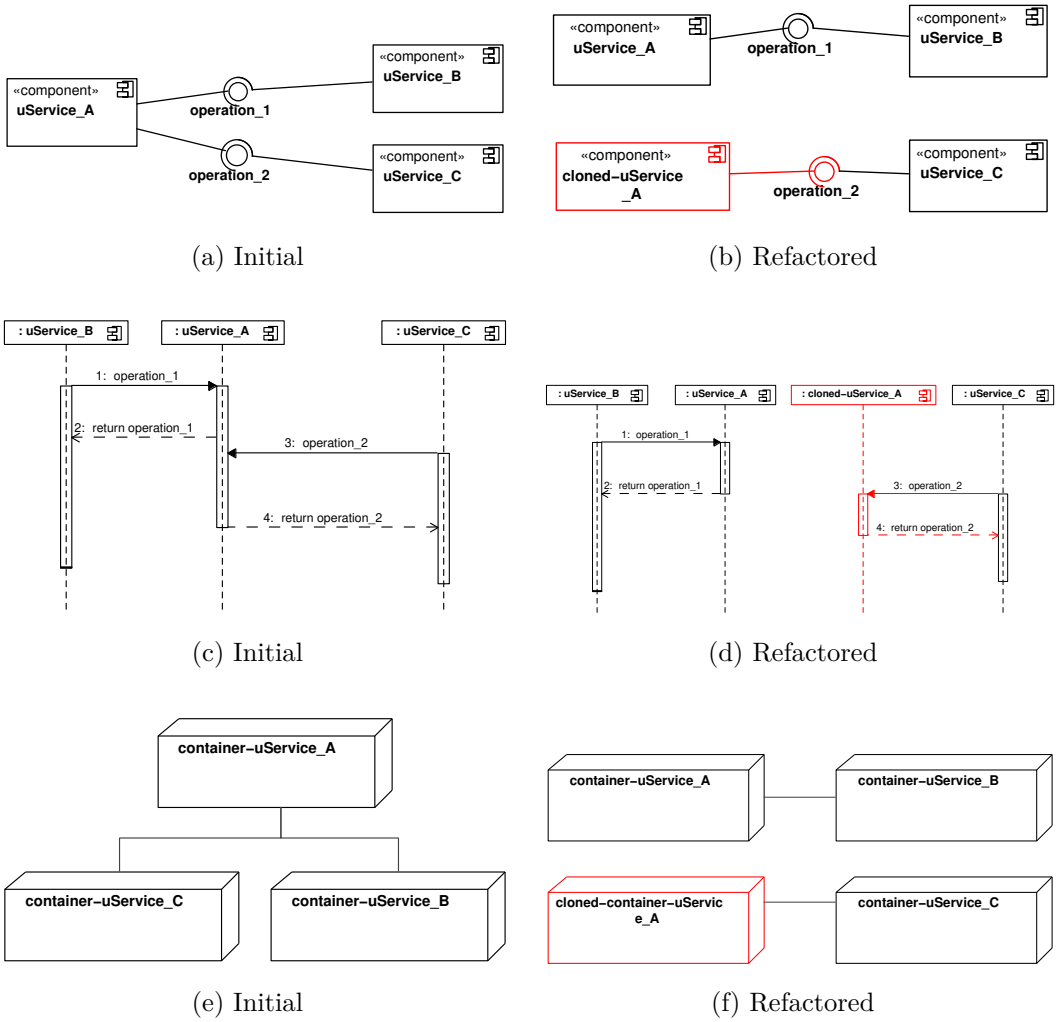
*cloned-container-Service_A*) is created and, finally, the action defines the newly required connections among UML Nodes (i.e., the connection between the UML Node *cloned-container-uService_A* and *container-uService_C*). In particular, the new node is linked to all other nodes that were originally connected to the node on which the microservice hosting the "critical" operation is deployed.

### 5.2.4 System refactoring

In this step, the refactoring actions performed on the model are translated into changes of a microservice-based system.

Refactoring actions on the system have been implemented using the *Docker Client*[9] Java library for the operations performed on docker instances. Regarding the online modifications of configuration files, we developed a Java library that is publicly available.[10]

[9] https://github.com/spotify/docker-client
[10] https://github.com/SEALABQualityGroup/microservices-refactoring




(a) Initial (b) Refactored

(c) Initial (d) Refactored

(e) Initial (f) Refactored

Fig. 5.11 The *move operation* refactoring action example on *operation_2* through a UML Software Model

**Clone refactoring** This action creates a replica of a microservice. In the running system, this translates into the creation of a new container deploying the same *Spring Boot* microservice we intend to clone. This refactoring is achieved by exploiting the Docker API to create and start a new container using the image of the original microservice. Once the replica is up and running, we need to balance it along with the original microservice. Specifically, we want to ensure that half of the traffic that was targeted at the original microservice is now redirected to its clone. Depending on the technology used to forward requests among microservices, three different scenarios are open:

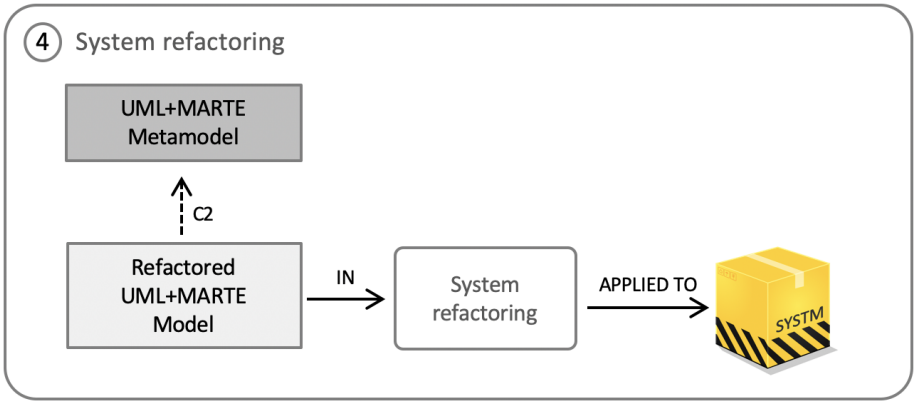



Fig. 5.12 System refactoring

- *Zuul and Eureka.* This scenario is straightforward because the combination of the *Zuul*[11] proxy with the *Eureka*[12] registration service automatically balances the additional microservice. When the new microservice registers to *Eureka*, *Zuul* adds it to the physical locations available for requests forwarding. Internally, *Zuul* uses *Ribbon*[13] to balance incoming requests using a round-robin policy. Therefore, no further modifications are required in this case.

- *Nginx.* When *Nginx*[14] is used as a reverse proxy, we need to modify its configuration to add a new server group containing the original and the cloned microservices. Moreover, the mapping of requests has to be updated to address the requests to the newly created server group. By default, Nginx uses a round-robin algorithm to balance the servers in a server group.

- *No proxy.* Finally, when no proxy is deployed in front of the original microservice, we can add one without disrupting the running system. In this case, we preferred to deploy *HAproxy*[15], because it is able to automatically generate a configuration by deriving the composition of server groups from the network links among docker instances.

**Move operation refactoring** This action moves an operation from a service to a newly created one that has the purpose of exclusively offering the operation. This is implemented by creating a replica of the original service that contains the operation

[11] https://github.com/Netflix/zuul
[12] https://github.com/Netflix/eureka
[13] https://github.com/Netflix/ribbon
[14] https://nginx.org/
[15] http://www.haproxy.org/

we want to move and, consequently, by forwarding all the requests that were intended for the moved operation to the replica. Similarly to the clone refactoring action, this action can be implemented in the same three scenarios:

- *Zuul and Eureka.* Once the replica has been created and has registered itself to *Eureka*, *Zuul* automatically detects it. In order to forward the requests for the moved operation to the replica, we need to add a new route in the *Zuul* configuration file. Such route is needed to map the path of the moved operation to the endpoint URL of the replica.
- *Nginx.* Analogously to the previous scenario, also when using *Nginx* as a reverse proxy, we need to add a route to redirect the requests to the moved operation. This is accomplished in *Nginx* by using the *location* directive to map a path to an endpoint URL.
- No proxy. In this case, a new proxy is added to the application. Both *HAproxy* and *Nginx* can serve this purpose with similar configurations for redirecting URL paths.

## 5.3 Evaluation

In this section, we discuss the evaluation we have performed with the aim of answering the following research questions:

- *RQ1: Do the proposed model refactoring actions improve the performance of the running system?*
- *RQ2: To what extent does performance antipattern (PA) removal improve the whole performance?*

### 5.3.1 Experiment setup

In order to validate the approach, we considered the following case studies:

- *E-Shopper*[16] is an e-commerce web application. The application is developed as a suite of small services, each running in its own Docker container and communicating with RESTful HTTP API. E-Shopper is composed by 9 application microservices developed with the Spring framework, each requiring a different database to operate.

[16] https://github.com/SEALABQualityGroup/E-Shopper

- *TrainTicket*[17] is a web ticketing application within the railway domain. It has been developed by Zhou et al. and it has been also presented in [131, 132, 130]. It is made up of 40 microservices and uses different programming languages. The most used framework is again Spring, since the most used programming languange in Train Ticket is Java, and for this reason we have selected it in our experimentation. In our previous work we reverse engineered its UML representation and presented it as a reference case study [45], while here we employ it to test our performance improvement approach.

We have generated different scenarios for each application to stimulate different parts of the system and discover which ones may suffer from performance degradation under concurrent usage.

For the E-Shopper application we have realized three scenarios with specific workloads:

- *Desktop*, the request of the homepage, with a workload of 3.8 user/sec;
- *Mobile*, the request of a specific service through an API call (e.g., it maps a call from the E-Shopper mobile app), with a workload of 225 user/sec;
- *Warehouse*, the request from a warehouse worker to set and control the availability of items, with a workload of 17.5 user/sec.

For the Train Ticket case study, we have realized two scenarios with respective workloads:

- *Rebook Ticket*, the scenario on which a customer can change a ticket reservation, with a workload of 4.5 user/sec;
- *Update User*, the scenario on which the admin changes user information, with a workload of 2.75 user/sec.

It is worth noticing that we stimulated both the applications with workloads heavy enough to stress them, but not too heavy to originate errors and timeouts. Each container was restricted to a single CPU core in order to limit the amount of resources necessary to generate the workloads. In order to achieve steady-state performance, for each experiment we executed a 20 minutes initial warm-up with additional 10 minutes warm-up after the application of each refactoring action. Intermediate warm-ups were necessary because the applications were never restarted to perform the refactoring actions, thus to simulate a production environment. Response time and utilization were continuously measured for 10 minutes after (the warmup following) every refactoring

[17] https://github.com/FudanSELab/train-ticket

action. Finally, all tests were repeated three times in order to avoid circumstantial external influence.

All the experiments were performed on a server with dual Intel Xeon CPU E5-2650 v3 at 2.30GHz, for a total of 40 cores and 80GB of RAM.

### 5.3.2 RQ1: Performance improvement

In order to answer RQ1, we evaluate if the approach is able to produce refactoring decisions that improve the performance metrics computed on the software models, and consequently the performance of the running system. To this end, we show how the approach applies refactoring actions that are promising on the basis of the model, and how these actions are propagated to the running system, in the context of the considered case studies. Moreover, we compare refactoring actions that were induced by performance antipatterns against other actions that we randomly perform on the model and on the running system. In this way, we are able to show that: i) our approach can select refactoring actions which improve the performance of the running system, and ii) basing the selection of such actions on the combination of QN analysis and performance antipatterns detection is more effective than randomly refactoring the system.

We start the assessment of the refactoring impact on the model by comparing utilization and response time before and after the modifications. Figures 5.13 and 5.14 show a comparison of response times and utilization computed on the QN for both case studies. Utilization is reported only for relevant microservices, that are those involved in at least one scenario and for which the utilization varies among refactoring actions.

In the E-Shopper case study, starting from the initial system, our approach proposed two alternative refactoring actions to improve the *Desktop* scenario: cloning the *web* microservice (*clone(web)*), and moving the *findfeaturesitemrandom* operation from the *items* microservice to a newly created one (*moveop(items/findfeaturesitemrandom)*). Among the other feasible actions, we randomly selected cloning the *items* microservice (*clone(items)*) and moving the *findproductsrandom* operation from the *products* microservice to a newly created one. These randomly selected actions are marked with a pattern in the histograms of Figure 5.13. We can see how cloning the *web* microservice is remarkably beneficial for the response time of *Desktop* scenario, while the randomly selected action of the same type, that is cloning the *items* microservice, has a negligible impact on the same scenario and a small impact on the *Mobile* one. When comparing the actions that move an operation to a new microservice, we can notice a difference

in response time, even if small, in favor of the refactoring targeting the operation *findfeaturesitemrandom*.

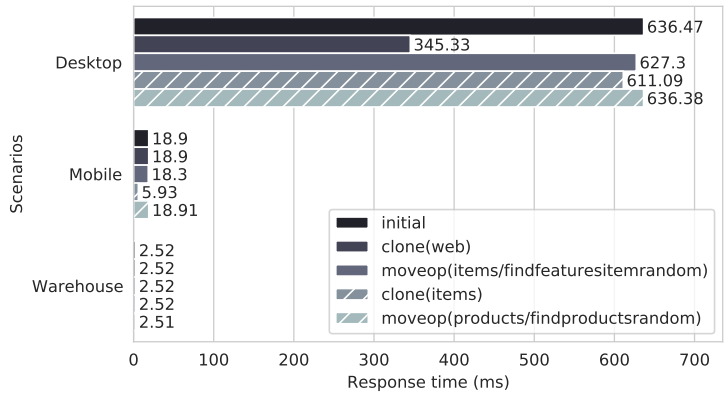
In order to improve the *Rebook Ticket* scenario of the Train Ticket case study, our approach proposed to clone the *rebook* microservice (*clone(rebook)*), or alternatively to move the operation *generate* from the *verification-code* microservice to a newly created one (*moveop(verification-code/generate)*). The randomly selected actions are: cloning the *admin-user* microservice (*clone(admin-user)*), and moving the *login* operation from the *sso* microservice to a new one. Also in this case, the actions selected on the basis of performance antipatterns induced a larger improvement in the response times of the targeted scenario, as we can notice from Figure 5.14.

As introduced before, we are also interested in evaluating if the application of the refactoring actions on the running system improves its software performance by comparing utilizations and response times before and after the modifications. Figures 5.15 and 5.16 show the measures obtained by monitoring both case studies as described in Section 5.1.1.

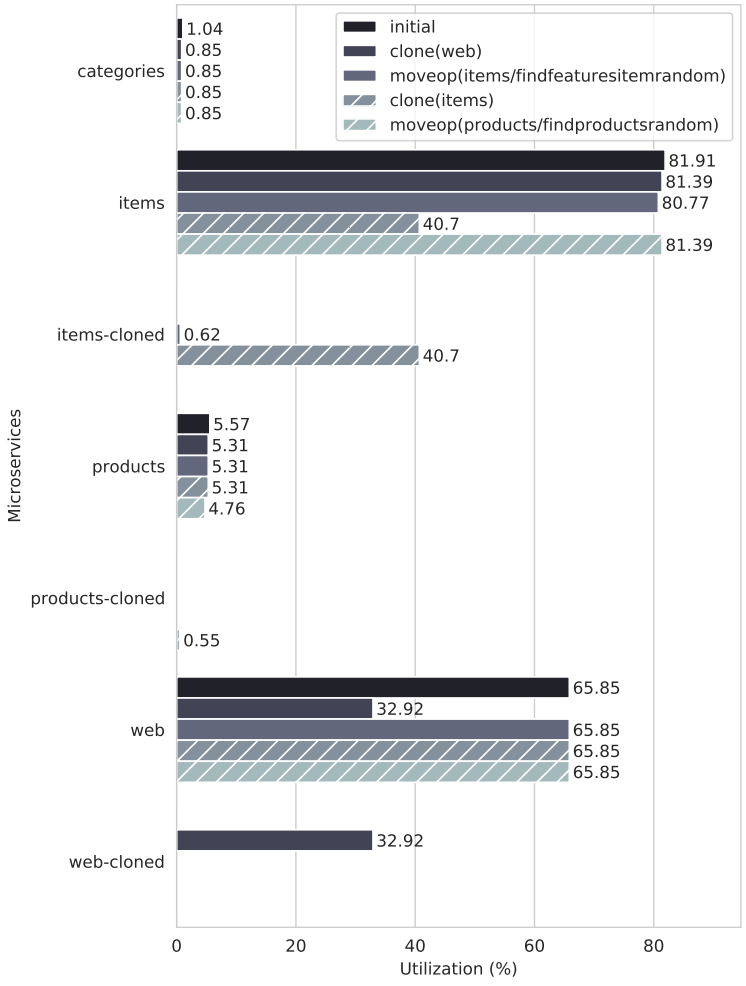
The results show that, in both case studies, the refactoring actions that were selected on the basis of antipatterns are more effective in improving the response times of the targeted scenarios. Furthermore, there are some interesting aspects to notice. For instance, by just looking at the utilizations computed on the QN for the E-Shopper case study (Figure 5.13b), a performance analyst would have probably guessed that cloning the *items* microservice would be the best action to perform. Instead, such refactoring only marginally decreases the response time of the *Desktop* scenario on the QN and on the system. In this case, cloning the *web* microservices was far more effective as also shown by the measures obtained from the running system.

We can notice an even more extreme situation in the Train Ticket case. Both the actions that were randomly selected actually increased the response time of the *Rebook Ticket* scenario on the running system (Figure 5.16a). Even if, in general, the cloning and moving operation refactoring actions are designed to produce a performance improvement, when applied without taking into consideration the design of the application may indeed result in degrading the performance.

These are just some examples of how the combination of design and runtime knowledge can induce a more thorough selection of the convenient refactoring to perform. More generally, design-runtime traceability provides additional knowledge that can be automatically maintained while supporting design decisions, also when the software is running in production.

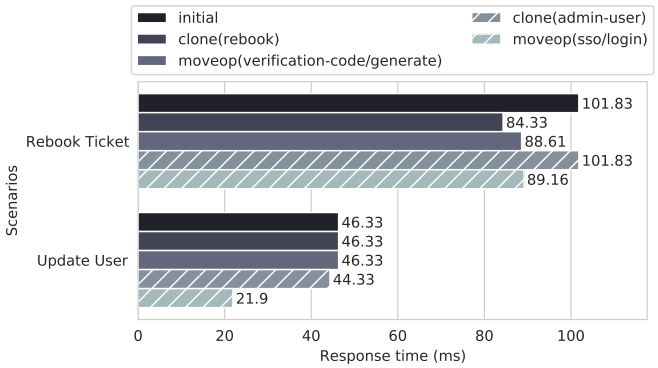



(a) Comparison of the effect of refactoring on response times.

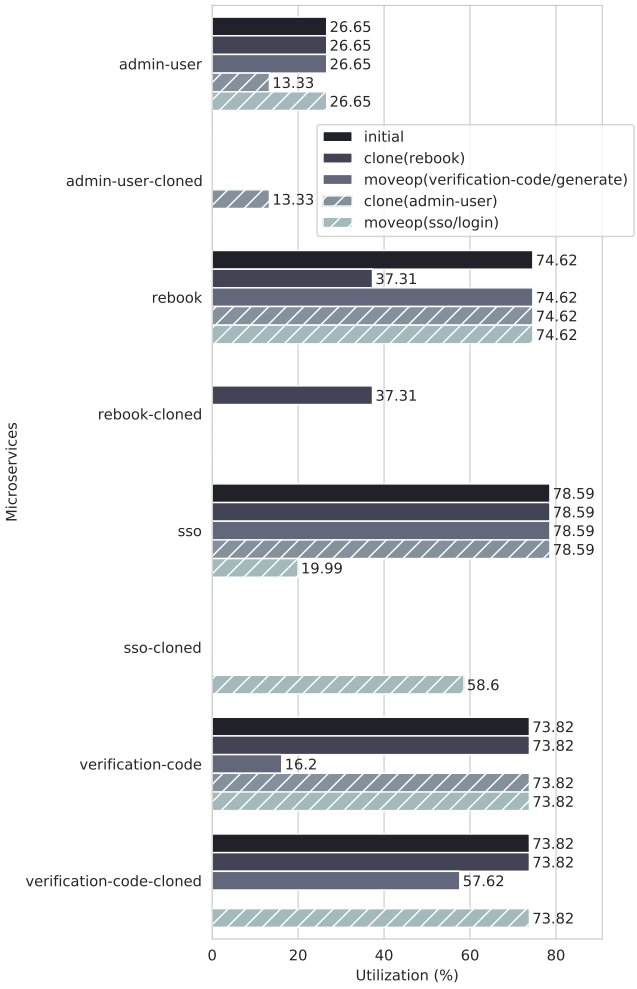



(b) Comparison of the effect of refactoring on the utilization of microservices.

Fig. 5.13 Average response times and utilizations computed on the QN for the E-Shopper case study.

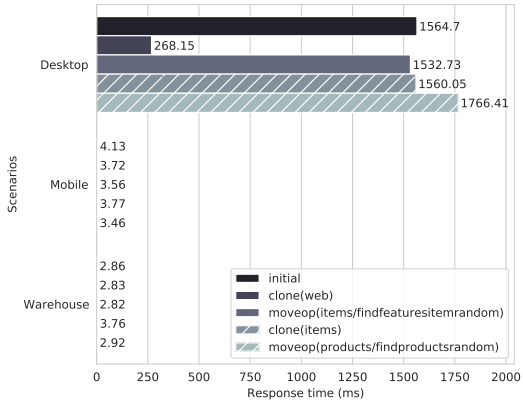



(a) Comparison of the response times of scenarios.

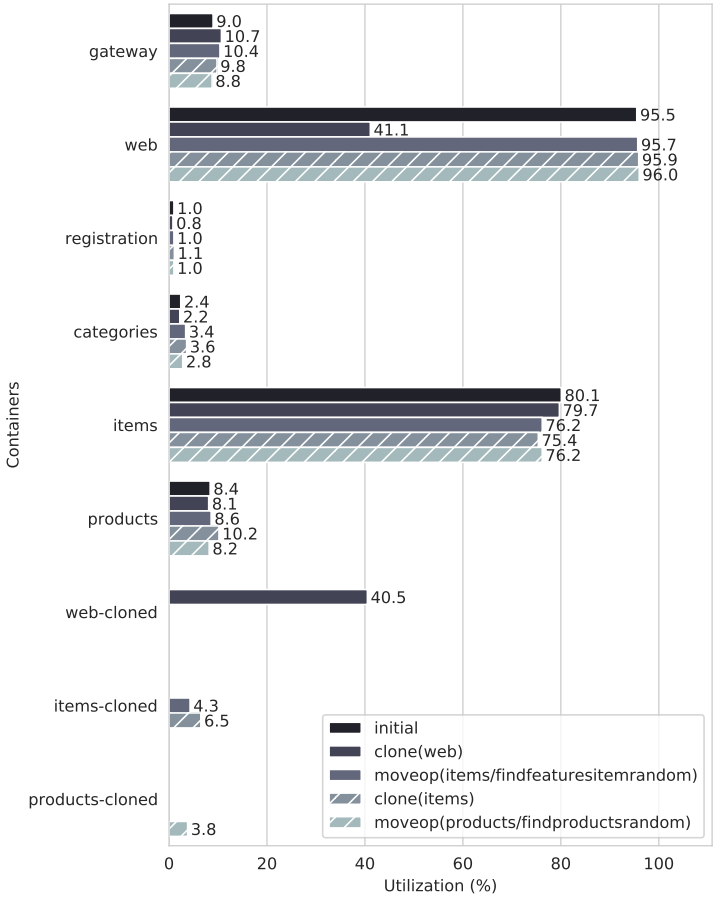



(b) Comparison of the utilization of microservices.

Fig. 5.14 Average response times and utilizations computed on the QN for the Train Ticket case study.

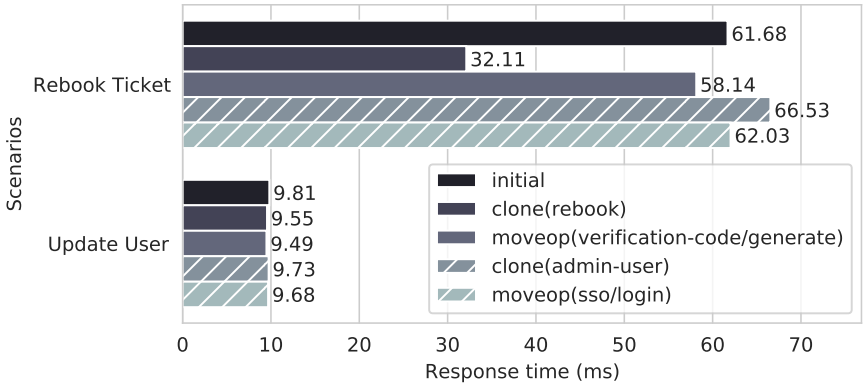



(a) Comparison of the effect of refactoring on response times.

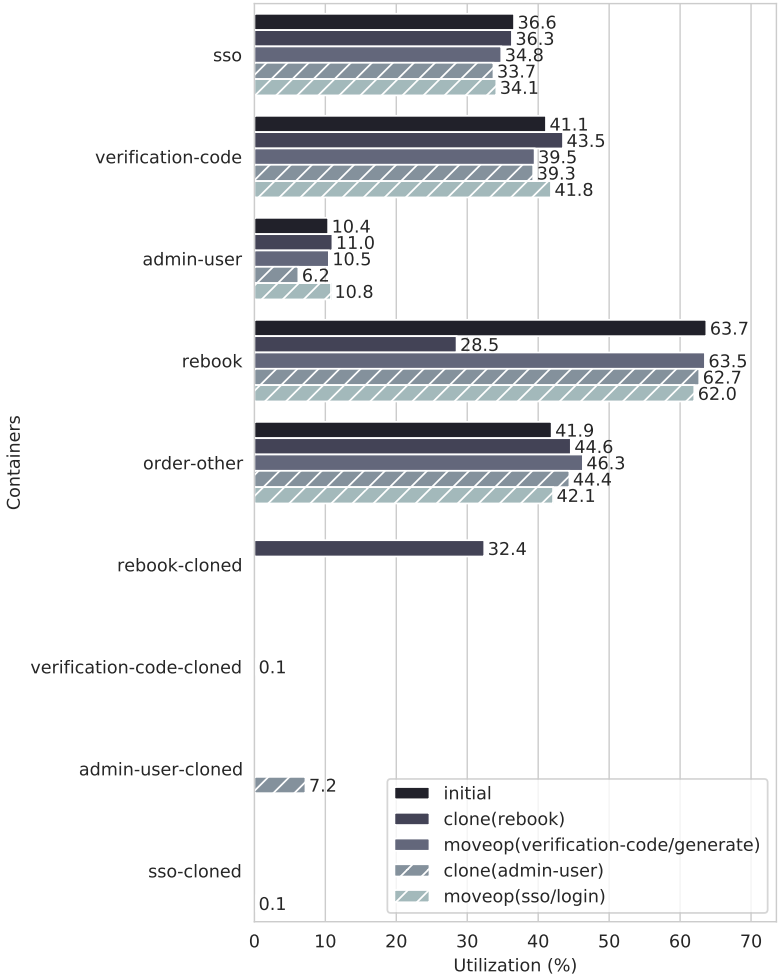



(b) Comparison of the effect of refactoring on the utilizations of microservices.

Fig. 5.15 Average response times and utilizations measured on the running system for the E-Shopper case study.




(a) Comparison of the response times of scenarios.




(b) Comparison of the utilizations of microservices.

Fig. 5.16 Average response times and utilizations measured on the running system for the Train Ticket case study.

### 5.3.3 RQ2: Performance antipatterns

In order to answer RQ2, we discuss the benefits of employing detection and refactoring of PAs to performance improvement forecasting. We consider here performance antipatterns that can suitably fit with microservice-based systems, namely Blob and PaF (Pipe and Filter) [113].

As described in Section 5.1.3, literals of a performance antipattern first-order logic representation are compared to thresholds. The definition of fixed values for these thresholds is an application-dependent task that can become very complex in some cases. Therefore, here we employ the concept of fuzzy thresholds [13]. Instead of deterministically identifying the occurrence of a performance antipattern, fuzziness in thresholds induces a probability for an antipattern to occur, as the combination of probabilities of threshold violations. The probability for an element $x$ to violate a fuzzy threshold $Th_k$ on $k$ metric is defined as follows:

$$P_k(x) = 1 - \frac{UBTh_k - F_k(x)}{UBTh_k - LBTh_k} \tag{5.3}$$

This expression considers $UPTh_k$ and $LBTh_k$ as the upper and lower bounds of $Th_k$ threshold, respectively, and $F_k(x)$ as the value that the $k$ metric assumes in the $x$ element. $UPTh_k$ and $LBTh_k$ are equals to the maximum and minimum observed value of $F_k(x)$, respectively.

**Blob** Each inequality of Expression 5.1, described in Section 5.1.3, undergoes a fuzzy evaluation like the one in Expression 5.3. Hence, the probability of a Blob occurrence is obtained as follows:

$$P(Blob) = P_{numClientConnects}(C_x) * P_{numMsgs}(C_x, C_y) * P_{maxHwUtil}(P_{xy}) \tag{5.4}$$

**Pipe and Filter (PaF)** Each inequality of Expression 5.2, described in Section 5.1.3, also undergoes a fuzzy evaluation like the one in Expression 5.3. Hence, the probability of a PaF occurrence is obtained as follows:

$$P(PaF) = P_{resDemand}(Op) * P_{maxHwUtil}(P) \tag{5.5}$$

We remark that the second literal in Expression 5.2 must be always equal to one to trigger a PaF, thus it can be omitted in Expression 5.5 that quantifies the PaF occurrence probability. □

In the following, Table 5.1 – Table 5.5 describe the effects on performance antipattern probabilities of refactoring actions that are either randomly selected or driven by performance antipattern detection. In each table, the $M_0$ column lists the initial probability, i.e., the value obtained on the initial configuration, the $M_1$ column lists the probability obtained after applying the action suggested by PADRE to remove the antipattern, and the $M_2$ column lists the probability obtained after applying a random refactoring action. Each row represents a literal of the performance antipattern expression, while the last row reports the occurrence probability of the whole antipattern.

### E-Shopper

Table 5.1 and Table 5.2 describe the probability of *Web* and *Items* being Blob performance antipatterns, respectively, in the E-Shopper case study. In particular, column $M_1$ of Table 5.1 corresponds to the *Clone* refactoring action on *Web* microservice as suggested by PADRE, while column $M_2$ corresponds to the *Clone* refactoring action on *Items* as a randomly selected recfactoring action. It is noteworthy that the probability drops from 0.80 to 0.39 after the refactoring action suggested by PADRE, while the probability grows to 1 after the random action of *Items* cloning.

We recall that the application of the same refactoring actions on the running system, as shown in Section 5.3.2, in case of cloning *Web* reduces the utilization of that microservices by 50.2%, whereas cloning *Items* does not change the utilization of the Web microservice.

Although the initial probability of *Items* being a *Blob* is lower than in the *Web* case, the *Move Operation* on *items/findfeaturesitemrandom*, suggested by PADRE, reduces the initial probability ($M_0$) from 0.25 to 0.06, while the application of the *Move Operation* on the randomly selected *products/findproductsrandom* does not change the probability at all.

We recall that in the running system, as shown in Section 5.3.2, the effect of the PADRE suggestion is negligible in terms of performance, as the utilization of *Items* is reduced by 0.4% and the response time (see Figure 5.15) is quite the same. However, the effect of the random action in terms of performance is twofold: the utilization of *Items* is decreased by 4%, and the utilization of *Web* is increased, albeit it is close to being saturated.

| Performance Antipattern Literal | $M_0$ | $M_1$ | $M_2$ |
|---|---|---|---|
| $P_{numClientConnects}(Web)$ | 1 | 1 | 1 |
| $P_{numMsgs}(Web)$ | 1 | 1 | 1 |
| $P_{maxHwUtil}(Container-Web)$ | .80 | .39 | 1 |
| $P_{Blob}(Web)$ | .80 | .39 | 1 |

Table 5.1 Probability of *Web* being a Blob. $M_0$ is the initial model, $M_1$ is the model refactored through a *Clone* refactoring action on *Web*, and $M_2$ is the model refactored through a *Clone* refactoring action on *Items*.

| Performance Antipattern Literal | $M_0$ | $M_1$ | $M_2$ |
|---|---|---|---|
| $P_{numClientConnects}(Items)$ | .5 | .25 | .5 |
| $P_{numMsgs}(Items)$ | .5 | .25 | .5 |
| $P_{maxHwUtil}(Container-Items)$ | 1 | 1 | 1 |
| $P_{Blob}(Items)$ | .25 | .06 | .25 |

Table 5.2 Probability of *Items* being a Blob. $M_0$ is the initial model, $M_1$ is the model refactored through a *Move Operation* refactoring action on *items/findfeaturesitemrandom*, and $M_2$ is the model refactored through a *Move Operation* refactoring action on *products/findproductsrandom*.

### Train Ticket

We report in Table 5.3 – Table 5.5 performance antipattern probabilities due to different refactoring actions applied to the Train Ticket case study.

Table 5.3 reports the probability of *verification-code/generate* operation being a Pipe and Filter (PaF). In particular, column $M_1$ refers to a *Move operation* refactoring action on *verification-code/generate* suggested by PADRE, while column $M_2$ refers to a randomly chosen *Move Operation* refactoring action effects on *sso/login* operation. We notice that the PaF probability decreases from 0.80 to 0 by applying the refactoring action suggested by PADRE. If we look at the running code (see Figure 5.16), this refactoring action decreases the utilization of *Container-Verification* microservice as well as the response time of the *Rebook Ticket* scenario. The random action, instead, increases the probability of *verification-code/generate* being a PaF to 0.87, and if we look at the running system, this refactoring action leads to increase both the utilization of *Container-Verification* and the response time of *Rebook Ticket* scenario.

| Performance Antipattern Literal | $M_0$ | $M_1$ | $M_2$ |
|---|---|---|---|
| $P_{resDemand}(verification-code/generate)$ | .88 | .88 | .88 |
| $P_{maxHwUtil}(Container-Verification)$ | .91 | 0 | .98 |
| $P_{PaF}(verification-code/generate)$ | .80 | 0 | .87 |

Table 5.3 Probability of *verification-code/generate* being a Pipe and Filter (PaF). $M_0$ is the initial model, $M_1$ is the model refactored through a *Move operation* refactoring action on *verification-code/generate*, and $M_2$ is the model refactored through a *Move operation* refactoring action on *sso/login*.

Table 5.4 reports the probability of *rebook* being a Blob. In particular, column $M_1$ lists the probabilities related to the *Clone* refactoring action on *rebook*, as PADRE suggests, while the column $M_2$ lists the probabilities related to the same refactoring action on *admin-user*. We notice that probability decreases from 0.46 to 0.12 by applying the refactoring action suggested by PADRE, while it remains unchanged after the randomly chosen action. If we look at the running code (see Figure 5.16), the effect of the random action of cloning the *admin-user* microservice on the utilization is negligible, but it increases the response time of the *Rebook Ticket* scenario by 7%. Instead, the effect of the refactoring action suggested by PADRE on system performance is twofold: i) the response time of the *Rebook Ticket* scenario decreases by 47.9%, and ii) the utilization of the *Container-Rebook* microservice decreases by 55%.

| Performance Antipattern Literal | $M_0$ | $M_1$ | $M_2$ |
|---|---|---|---|
| $P_{numClientConnects}(rebook)$ | .5 | .5 | .5 |
| $P_{numMsgs}(rebook)$ | 1 | 1 | 1 |
| $P_{maxHwUtil}(Container-Rebook)$ | .92 | .24 | .93 |
| $P_{Blob}(rebook)$ | .46 | 0.12 | .46 |

Table 5.4 Probability of *rebook* being a Blob. $M_0$ is the initial model, $M_1$ is the model refactored through a *Clone* refactoring action on *rebook*, and $M_2$ is the model refactored through a *Clone* refactoring action on *admin-user*.

Table 5.5 reports the probability of *verification* being a Blob. In particular, column $M_1$ lists the probabilities of the *Move Operation* refactoring action on *verification-code/generate*, as PADRE suggests, while column $M_2$ reports the probabilities of the same refactoring action on *sso/login*. We notice that the probability decreases from 0.46 to 0 by applying the refactoring action suggested by PADRE, which also reduces

the *Verification.code* utilization and the response time of the *Rebook Ticket* scenario (as shown in Figure 5.16). The application of the random action instead increases the probability of *verification-code* being the Blob to 0.98.

| Performance Antipattern Literal | $M_0$ | $M_1$ | $M_2$ |
|---|---|---|---|
| $P_{numClientConnects}(verification-code)$ | .5 | 0 | 1 |
| $P_{numMsgs}(verification-code)$ | 1 | 0 | 1 |
| $P_{maxHwUtil}(Container-Verification)$ | .91 | 0 | .98 |
| $P_{Blob}(verification-code)$ | .45 | 0 | .98 |

Table 5.5 Probability of *verification-code* being a Blob. $M_0$ is the initial model, $M_1$ is the model refactored through a *Move Operation* refactoring action on *verification-code/generate*, and $M_2$ is the model refactored through a *Move Operation* refactoring action on *sso/login*.

### 5.3.4 Summarizing discussion

On the basis of the results obtained, we can state that the removal of performance antipatterns leads to a performance improvement of a running system. On the other hand, ignoring the performance antipatterns knowledge induces either unchanged performance, in the best case, or performance detriment in all other cases.

Indeed, we experience on the Train Ticket case study a performance improvement in terms of lower hardware utilization from 0.411 to 0.395 (i.e., by about 4%) when the *verification-code/generate* operation is moved, and the response time of the *Rebook Ticket* scenario is reduced by 5% as well. It is noteworthy that the same refactoring action also removes the "Pipe and Filter" performance antipattern (as shown in Table 5.3).

Also, we experience on the E-Shopper case study a lower hardware utilization from 0.955 to 0.411 (i.e., by about 57%) when the *Container-Web* is cloned (as suggested by PADRE), and the response time of the *Desktop* scenario is reduced by about 83% (i.e., from 1,564 ms to 268 ms as shown in Figure 5.15), and the probability of Web being a "Blob" performance antipattern decreases by 39%, as shown in Table 5.1.

We have also noticed that the application of a random refactoring action induces either unchanged performance or performance detriment. For example, the Train Ticket case study has shown a higher utilization from 0.411 to 0.418 (i.e., an increment by 1.7%), while the response time of the *Rebook Ticket* scenario remained unchanged.

The probability of *verification-code/generate* being a "Pipe and Filter" performance antipattern is increased by 11% (as shown in Table 5.3. Instead, the random action on the E-Shopper case study has caused no changes both for the utilization, the response time, and the probability of *Web* being a "Blob" performance antipattern (as shown in Table 5.1).

### 5.3.5 Threats to validity

In this section, potential threats to validity associated with the experimental validation are discussed.

*Conclusion validity* concerns the reliability of the measures. As explained in Section 5.3.1, we properly and rigorously designed our experimental setup. We attempted to avoid any bias by: i) generating different scenarios with respective workloads to simulate different parts of the systems; ii) stressing the execution of the applications to check performance under realistic conditions; iii) repeating the experiments several times in order to avoid circumstantial external influences. Furthermore, during the experimental evaluation, we ensured that the observed performance improvement was actually induced by the refactoring actions selected by our approach. To this end, we compared our solutions with performance variations caused by randomly selected refactoring actions.

*Internal validity* concerns any extraneous factor that could influence our results. In general, the implementation of the approach could be defective, as well as the results of the analysis could be inaccurate. In order to avoid any bias: i) we have assured that the implementation was aligned with the software design to generate accurate traceability models and consistently annotate the performance models; ii) we completely delegated the analysis of the considered performance model to an external consolidated tool; iii) we employed technologies that are widely tested in production (e.g., Zuul, Eureka, Nginx) that provide unified interfaces to allow refactoring of microservices without exposing the internal structure. Furthermore, we provided a detailed discussion of the code instrumentation and made publicly available the source code in order to allow other researchers to reproduce and inspect the experiments.

*Construct validity* concerns any factor that can compromise the validity of the experiment and the resulting observations. Evaluation results are highly dependent on the quality of the considered measures. In the evaluation, we apply monitored data observed from running system and performance measures. As described above, we properly designed our experimental setup to avoid influence factors. For instance, we disabled as many system services (Linux *systemd* units) as possible to lessen the

effects of context switching. To avoid interferences that may be caused by internal errors, we also monitored the machine by looking at system logs (*dmesg* buffer and *systemd* journal) for unexpected entries.Another threat may be posed by missing links in the generated traceability models. This may be caused by errors in the generation of Log models from monitoring traces, or by incorrectly matching the patterns defined in the correspondences specification, either for UML or Log models. To avoid this, we performed several manual assessments of the generated traceability models by looking for missing links or links connecting the wrong elements. Finally, we have annotated the performance indices in the UML-MARTE models in a consistent way with the consolidated literature in the field of software performance.

*External validity* refers to the generalizability of the obtained results. Case studies may be selected to facilitate a deeper understanding of the approach and this could affect their representativeness. In order to mitigate this, we selected two microservice-based web application benchmarks that have been successfully used in previous work for their size and heterogeneity [12, 45]. This choice also provides indicators for cases having similar properties. With reference to the used languages and tools, we adopted specific development and monitoring technologies (i.e., based on the Spring framework), as well as the specific modeling standards like UML and MARTE. These technologies are widely used both in academia and industry, thus supporting the approach to be generalizable and easily reproduced in other contexts.

Adopting specific tools (JTL and PADRE) could threat the generalizability of the approach. In this merit, very few other alternative tools are available for replacing JTL, and actually none for replacing PADRE to detect performance antipatterns in UML models. However, both tools have proven to be suitable for the specific purposes of the approach and to be used in heterogeneous contexts.

Finally, the size of the example application considered here is not very large, but complex enough to demonstrate the effectiveness of the approach. Nothing can be asserted about the scalability of the approach on large size systems, which remains one of our future objectives.

## 5.4 Related work

In this section we discuss existing work that proposes approaches for architectural-based improvement of software systems driven by the observation of monitoring data and/or their interleaving with the software design modeling. Researchers from several areas (e.g., self-adaptive systems, software engineering and continuous system engineering)

have actively studied a wide variety of methods and techniques applicable at design time and/or at runtime. Hereafter, we focus on approaches dedicated to software performance and on approaches that make use of software models in the domain of microservices.

### 5.4.1 Software performance engineering approaches

A vast literature exists on performance modeling, performance monitoring, and performance problem identification techniques, as quite separate research domains. We report on the most significant papers that attempt at merging such domains.

Trubiani et al. [121] have provided a systematic process to identify performance issues from runtime data, based on load testing coming from operational profile and application profiling. In particular, from runtime data, performance antipatterns are detected, aimed at identifying common performance issues and their solutions. Software refactoring is then (manually) applied to solve identified performance antipatterns. Apart from technological and implementation aspects, a methodological difference distinguishes our approach from the one by Trubiani et al. [121], namely: we bring runtime data up to the design level, by annotating a UML model with the MARTE profile, and one of the main advantage of addressing performance issues at design level, among other, is to narrow down the search space for potential actions that can beneficially affect system performance.

Menascé et al. [103] have proposed the DeSARM approach, whose scope is the derivation of architectural models at runtime. Such models can be used in decentralized decision-making for architecture-based adaptation in large distributed systems. To this aim, DeSARM is able to identify important architectural characteristics of a running application, such as components, connectors, nodes and communication patterns. DeSARM has been used by Albassam et al. [57] to introduce runtime failure analysis and architectural recovery on the discovered system architecture. However, DeSARM has not been adopted for identifying performance problems.

Petriu et al. [7] deal with the automated generation of performance models from UML-MARTE architectural models, and the propagation of performance analysis results back to the latter. The main difference with our work is that this approach fully works at the model level, and it does not consider the availability of a running software system from which runtime information can be extracted for a more accurate identification of performance problems.

Logs obtained from monitoring a running software system are exploited by van Hoorn et al. [125] to automatically extract workload specifications for load testing

and performance models parameterization. However, this approach is limited to the parameterization of performance models, whilst our approach provides support for the interpretation of analysis results carried out by performance antipattern detection and their possible solutions.

Mazkatli et al. [90] have presented an approach for continuous integration of performance models that considers parametric dependencies after analyzing the source code changes. The approach builds upon the Palladio approach and the goal is to automatically keep the performance models up-to-date to allow architecture-based performance prediction. In contrast with this approach, we have provided an approach to identify model-based design alternatives to overcome detected performance problems and to apply them on the system.

In a more recent work, Bernardi et al. [19] use UML models annotated with MARTE to evaluate the performance of an industrial application. UML models are augmented with service demands obtained from execution logs, and later transformed to GSPN. Event logs are aligned to the design scenario in order to ensure correctness, remove traces that are not related to performance, and discover new system behaviors. Finally, the results of the simulation are presented to the user both in textual and graphical formats.

Recently, Heinrich [62] has proposed an approach to align architectural models used in development and operation by means of a correspondence model between implementation artifacts and component-based Palladio architectural models. In contrast with this work, our approach uses UML that is a widely adopted standard respect to the Palladio Component Model. They use an ad-hoc correspondence model that is then exploited by a complex pipeline of transformations, while we propose a general approach where the correspondences are generated from a declarative specification easily adaptable to other contexts. They use a specific infrastructure monitoring, while we combine application instrumentation and infrastructure monitoring. Also the scope is different, as their contribution is to assess software performance with the aim to support design decisions, whereas our approach is not limited to build design-runtime correspondences and predict performance issues. We indeed enable the identification of design alternatives and the implementation of system refactoring actions to improve performance. On the other hand, in contrast with our approach, Heinrich [62] also considers workload characterization and model structure updates, where the model is updated online, and scalability and accuracy are validated. Finally, it is not limited to the microservices domain.

### 5.4.2 Model-based approaches for microservices

In self-adaptive systems, software models have been mostly used at development and specification time, and a few works considered software models for microservice application adaptations. In this respect, Rademacher [105] surveyed the use of models in microservice and service-oriented architectures. Also, Derakhshanmanesh [43] provided a vision and future challenges on the use of domain-specific modeling languages and model transformations across the full software lifecycle (including runtime) to define and evolve a microservice application at the architectural level.

Weyns [128] described relevant aspects and future challenges in the field of software engineering for self-adaptive systems. In particular, the author puts the concrete realization of runtime adaptation mechanisms that leverage software models at runtime to reason about the system and its goals. The use of models at runtime (known as models@run.time) [24, 16] has been proposed to extend the applicability of software models produced in MDE approaches to the runtime environment. Such models should represent the system and its current/updated state and behavior. The envisioned goal is to support adaptive systems, e.g., to drive subsequent adaptation decisions, to fix design errors or to explore new design decisions. As an alternative to models at runtime, we used traceability models to represent runtime information and its relation with design models. Such solutions allow us to exploit existing monitoring infrastructure and existing design models throughout all phases of the approach.

Dullmann and van Hoorn [47] proposed a preliminary framework to generate microservice environments that can then be used for measurement-based evaluation of performance and resilience. The approach allows developers to create models and generate Java code and deployment files. The generated microservices are automatically instrumented to collect metrics at runtime. In contrast with our work, the proposed setup was developed and used as benchmarking environments for their evaluation of approaches for performance and resilience. Although the framework is able to generate microservice environments with specified properties, it has not been adopted for the adaptation of microservice-based systems.

Zuniga-Prieto et al. [133] proposed an incremental and model-driven approach that supports the integration of cloud service applications and their dynamic architecture reconfiguration. Models are used to generate skeletons of microservices, integration logic, and also scripts to automatically deploy and integrate the microservices in the specific cloud environment where a microservice will be deployed. The approach supports the integration of increments composed of several microservices, whereas it has

not been adopted for the improvement of existing services; also, runtime information is not considered.

Sampaio et al. [108] proposed a platform-independent runtime adaptation mechanism to reconfigure the placement of microservices based on their communication affinities and resources usage. The authors propose to identify the runtime aspects of microservice execution that impact the placement of microservices by using models at runtime. In contrast with our approach, the main contribution is limited to the reconfiguring mechanism to manage the placement of microservices.

# Chapter 6

# Safety Critical Assessment in Complex Systems

One of the major open challenges in the complex systems domain is to achieve an efficient integration between design and runtime aspects: the system behavior at runtime has to be better matched with the original system design in order to understand critical situations that may occur, as well as corresponding potential failures in design [31]. Methods and tools already exist for monitoring system execution and performing measurements of some runtime properties. However, many of them do not rely on models and, usually, do not allow a relevant integration with corresponding design models.

The European MegaM@Rt2 project[1] notably intends to address such issues. As part of its continuous system engineering approach [4], the project aims at providing a runtime-design time feedback loop that could be deployed and used in different industrial domains. Among other benefits, such a feedback from runtime-level to (architectural) design-level allows software engineers to control and manipulate elements they would not be able to access otherwise.

This chapter reports on a practical experience of using a model-based approach and related techniques to deal with the interactions between design time and runtime in the development of a safety-critical software system. Notably, the novelty resides in the combination of different complementary solutions for model traceability and model views in order to provide support for such a practical use case. Using the proposed approach, we also show how we can automatically infer some design deviations, and identify elements affected by these deviations, from a possibly large spectrum of runtime system configurations or conditions.

[1] https://megamart2-ecsel.eu/

The reported model-based experiment is based on an real industrial use case, a *Railway system* developed by CLEARSY[2], one of the industrial partners of the MegaM@Rt2 project. In this context, CLEARSY aims at improving the robustness of its system by integrating the use of model-based techniques in its development cycle. Their goal is *1)* to determine whether environmental conditions are met, and *2)* to detect variations in the behaviour of the system in order to anticipate on possible failures.

Here we propose both a conceptual model-based approach and an implementing solution that relies on Eclipse and EMF-based tools. The core components of our solution are *1)* the definition of correspondences between design and runtime elements (as traceability models), and *2)* the building of related views aggregating design and runtime models in a transparent way. To this intent, our implementation leverages the existing JTL [36] and EMF Views [32] tools.

## 6.1 The platform screen doors control case study

The work is motivated by a practical use case proposed by CLEARSY in the context of the MegaM@Rt2 project. The Coppilot system[3] is a controller for platform screen doors that has been deployed on several subway lines across the world. It is a typical example of a complex life-critical system and its role is to ensure that platform screen doors never open at a time when passengers' lives may be at risk (e.g., absence of train, moving train, etc.).

A Coppilot system usually manages a metro, tram or train station. It uses a pair of lidars (Laser Imaging Detection and Ranging) at each extremity of the platform to measure the position and speed of trains entering the station, and one lidar over each door of the vehicle that detects the movement of the doors. On top of the lidars, several computation units are dispatched, one for each lidar that runs image processing algorithms and a central one that calculates the safety critical outputs, i.e. opening authorization for each platform doors.

Although certification authorities often require the use of formal methods to develop such critical systems, there are both economical and technical issues reducing the possibilities to formally validate all the components of the system. As a consequence, one common strategy to develop such critical systems is to structure them around *1)* a safety core component that is formally validated and that provides the outputs of

[2] https://www.clearsy.com/en/
[3] https://www.coppilot.fr/en/coppilot-system/

the system, and *2)* a set of satellite components for which formal methods are not (or barely) used. The safety core must consider all the other components as unreliable (i.e. without any assumption that they will behave as expected). In Coppilot, the safety core is run by the central calculator and the lidar computation units are the satellite components.

This strategy allows to decouple the safety analysis from the software development. The safety analysis identifies both the nominal (intended and expected) behavior of the system and the defective behavior caused by the unsafe components. As a result, a list of safety requirements is obtained and have to be implemented into the safety core. The formal methods used to develop the software then provide a formal proof of the correct implementation of those safety requirements. In Coppilot, the safety core has been developed and certified by using the B method[4], formally ensuring that the final implementation satisfies all the safety requirements. According to the safety analysis, the system is required to take a fallback position as soon as a life-endangering situation becomes possible. Examples of situations that require entering (or moving to) a fallback position may be sensor inconsistencies, failures, or data/message corruption. Different fallback position situations require different kinds of intervention (e.g. the system could be manually restarted or it could automatically restore its previous state)

In order to obtain a balanced trade-off between safety and availability, we aim at reducing fallback position occurrences. Some occurrences are legitimate and are part of the nominal operation of the system. Although unavoidable, they are rare enough to have little impact on the overall availability of the system (e.g. hardware failures or exceptional temporary conditions). Moreover, other fallback situations may be caused by design defects. For instance, a recurring fallback situation may be the result of an overly conservative safety analysis and significantly reduce the system availability. In order to detect and resolve such defects, the system needs to be tested in operational conditions. In this context, the information collected at runtime play a key role. In fact, linking (runtime) logs that point out fallback situations back to the corresponding (design) safety requirements can help engineers in taking the appropriate corrective actions (e.g., maintenance, bug tracking or safety analysis review).

### 6.1.1 Current log analysis practice

CLEARSY collected hundreds of gigabytes of logs over several months of system operation. The monitoring infrastructure had access to the messages exchanged in the

[4] https://www.methode-b.com/en/

system and to the components' states. Logs are stored in comma-separated values files and are manually analyzed in a spreadsheet processor.

The logs analysis is performed by initially identifying all the occurrences of fallback. The cause of fallback is analyzed by considering that one of the last events has usually triggered a transition that led the automaton in a wrong position. Discovering such events may require to track back to a huge number of log items to find an explanation to the defect. To this intent, system engineers require some support in order to facilitate their exploitation work.

### 6.1.2 A concrete example

As shown in Listing 6.1, logs are composed of a set of events with a timestamp. Each event refers to a specific automaton and reports the value of variables from sensors.

```
1  [...]
2  20161017_231136106;u;dbg;192.168.10.101;1;0;M11.LogiqueSecu.
       ↪ Algo_DonneesUtiles__Position_M21;33139
3  20161017_231136106;u;dbg;192.168.10.101;1;0;M11.LogiqueSecu.
       ↪ Algo_DonneesUtiles__Position_M24;33116
4  20161017_231136106;u;dbg;192.168.10.101;1;0;M11.LogiqueSecu.
       ↪ Algo_DonneesUtiles__Vitesse_M21;6
5  20161017_231136106;u;dbg;192.168.10.101;1;0;M11.LogiqueSecu.
       ↪ Algo_DonneesUtiles__Vitesse_M24;2
6  [...]
7  20161017_231136377;u;dbg;192.168.10.101;1;0;M11.LogiqueSecu.
       ↪ Algo_DonneesUtiles__Position_M24;32766
8  20161017_231136377;u;dbg;192.168.10.101;1;0;M11.LogiqueSecu.
       ↪ Algo_DonneesUtiles__Vitesse_M21;0
9  20161017_231136377;u;dbg;192.168.10.101;1;0;M11.LogiqueSecu.
       ↪ Algo_DonneesUtiles__Vitesse_M24;1
10 [...]
11 20161017_231136480;u;dbg;192.168.10.101;1;0;M11.LogiqueSecu.
       ↪ Algo_DonneesUtiles__Position_M24;33117
12 20161017_231136480;u;dbg;192.168.10.101;1;0;M11.LogiqueSecu.
       ↪ Algo_DonneesUtiles__Vitesse_M24;65535
13 20161017_231136480;u;dbg;192.168.10.101;1;0;M11.LogiqueSecu.
       ↪ Algo_DonneesUtiles__fallback_M11;inconsistent_position
14 [...]
```

Listing 6.1 A sample of raw log.

Checking position consistency is one of the functions provided by the safety core to ensure that sensors report correct values. When a train has stopped at the right position, each sensor should report its position as zero. In real cases, sensors may not correctly report the real position of the train; this may be caused by slow buffer, delay in the communication network, or a displacement of the laser. The system tolerates such inconsistency within a specified threshold limit value.

The raw logs in this Listing 6.1 show a fallback due to inconsistent position data that occurs at timestamp 231136480 (see line 13). This inconsistency is caused by the sensors reporting positions that differ more than a specified threshold (in the current model it is 372 millimeters). The positions that have to be considered are the ones that precede the fallback events; in fact, Sensor M21 reports a position of 33139 (timestamp 231136106, Line 2) and Sensor M24 reports a position of 32766 (timestamp 231136377, Line 7).

Currently, this fallback would be analyzed by hand by looking at the last position data communicated by the sensors. Those positions can be located a few lines above the fallback (as for position of M24) or dozens of lines above. Having the ability to automatically detect the fallback and associate the values reported by the sensors causing it, as well as their timestamps, would considerably simplify the log analysis process. System engineers would be oriented more quickly towards the appropriate corrective actions, such as maintenance, bug tracking, or safety analysis review.

## 6.2 Exploiting design-runtime traceability for system improvement

As an answer to the problem described in the previous section, we propose a model-based approach that facilitates the establishment and exploitation of correspondences between design and runtime elements of a system. In particular, the system behavior at runtime is monitored, logged and then related to the initial system design whenever appropriate. The final objective is to connect fallback critical situations (at runtime) with their corresponding potential causes (in the system design and runtime). We specify design-runtime correspondences by means of a traceability model that links design and runtime information. These correspondences, along with the design and runtime models, are used as input to an integrated view that transparently relates the runtime logs with the initial system design. Navigating and querying this view can help the system engineer to discover fallback situations, identify their causes without manually analyzing very verbose logs, and navigate back to related safety requirements.

Figure 6.1 depicts the overall approach proposed as an answer to our motivating case study from Section 6.1. In this section, we outline the approach main concepts whereas detailed explanations are provided in Section 6.3.

Components developed specifically for this experiment, and existing solutions that had to be extended, are underlined in the figure. Note that the proposed approach can be considered as a model-based instantiation (relying on traceability models as

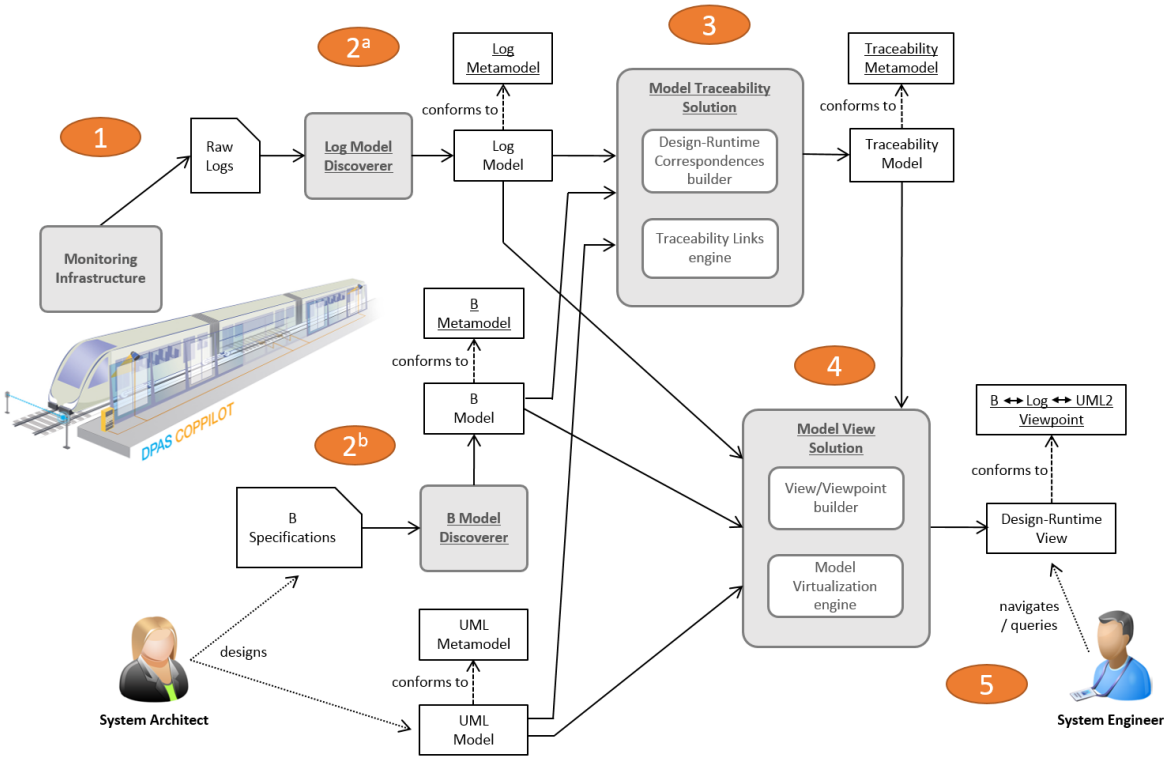



Fig. 6.1 Outline of the model-based approach (full-line arrows for the data/model flow).

a "knowledge base") of the well-known MAPE-K loop, that is frequently used in the context of self-adaptive systems for instance [9]. The main steps of our approach are the following:

1. **Monitoring.** The considered system first has to be correctly instrumented in order to generate usable traces of its runtime execution. In practice, execution traces are often serialized in specific and (semi-)structured textual formats (e.g. comma-separated values). However, for large volumes of data, a binary format may be more efficient (e.g. the Common Trace Format (CTF) standard[5]). In any case, the runtime traces must contain the relevant information required to later relate the runtime events with design elements. Moreover, we considered UML diagrams describing a static view of the system.

2. **Discovering design and runtime models.** When not already stored as models that conform to an explicit metamodel, the design and runtime artefacts (requirements specification, runtime traces, etc.) should be converted to models. This may require specifying corresponding metamodels if not available. In our

[5] https://diamon.org/ctf/

approach, we specified both the *Log Metamodel* that describes the runtime traces and the *B Metamodel* that covers B specifications (see *Log Model* and *B Model* in Figure 6.1, respectively). To convert runtime traces and B specifications into models, we use metamodel-driven *model discoverers* [29].

3. **Computing design-runtime traceability links.** Once the design and runtime models are available, they can be linked together in several ways. We use traceability relationships [100] which have been designed to help users understand associations and dependencies of heterogeneous models. For some previously-identified cases, this can be performed automatically thanks to the definition of a list of patterns (using an appropriate formalism or language) to be detected from the analyzing the runtime data. Thus, the generated traceability model relates together the previously obtained runtime and design models according to these defined patterns [12].

4. **Building the design-runtime view.** In order to provide a transparent and integrated access to the runtime-design traceability information, we build a model view [28] based on the previously obtained runtime, design and traceability models. This view acts as a "virtual" model that refers to these input models and connects them together according to the traceability information computed in the previous step. It has to be able to handle possibly large input models (e.g. verbose runtime traces) [30]. Moreover, it is important to note that this view conforms to a viewpoint that specifies how the different corresponding metamodels (*B*, *Log* and *UML* in our present case) are interconnected together. Such a viewpoint can also filter the element types that are not required by the users/engineers for the targeted engineering activities.

5. **Navigating and querying the design-runtime view.** The view is an integrated interface provided to the system engineer in charge of analyzing and diagnosing the system. It offers a single entry point to the engineer, thus hiding the unnecessary complexity of the individual models (including the Traceability one) contributing to the view. From the view, the engineer can transparently navigate and query all the information relevant to the system, its runtime behavior and design specification. She/he can notably specify her/his own particular queries, and also rely on predefined libraries of queries that are adaptable in the context of her/his scenario. This way, the view can be used to diagnose the system and conduct its evolution (in collaboration with the system architect for example).

In this approach, when the system engineer wants to build a view on another runtime trace, only two steps need to be run again: first building a model from the new trace (2a), then recomputing the traceability links (3). The view can then be navigated and queried accordingly.

If the engineer wants to analyze a new fallback situation, then the only change required is to add a pattern for this fallback in the traceability link computation step (3). If the view should contain more information, e.g. specific to this new fallback situation, then the view configuration should be altered, or a new view should be created to reflect the updated situation (step 4).

## 6.3 Experiences on building and using a model-based solution

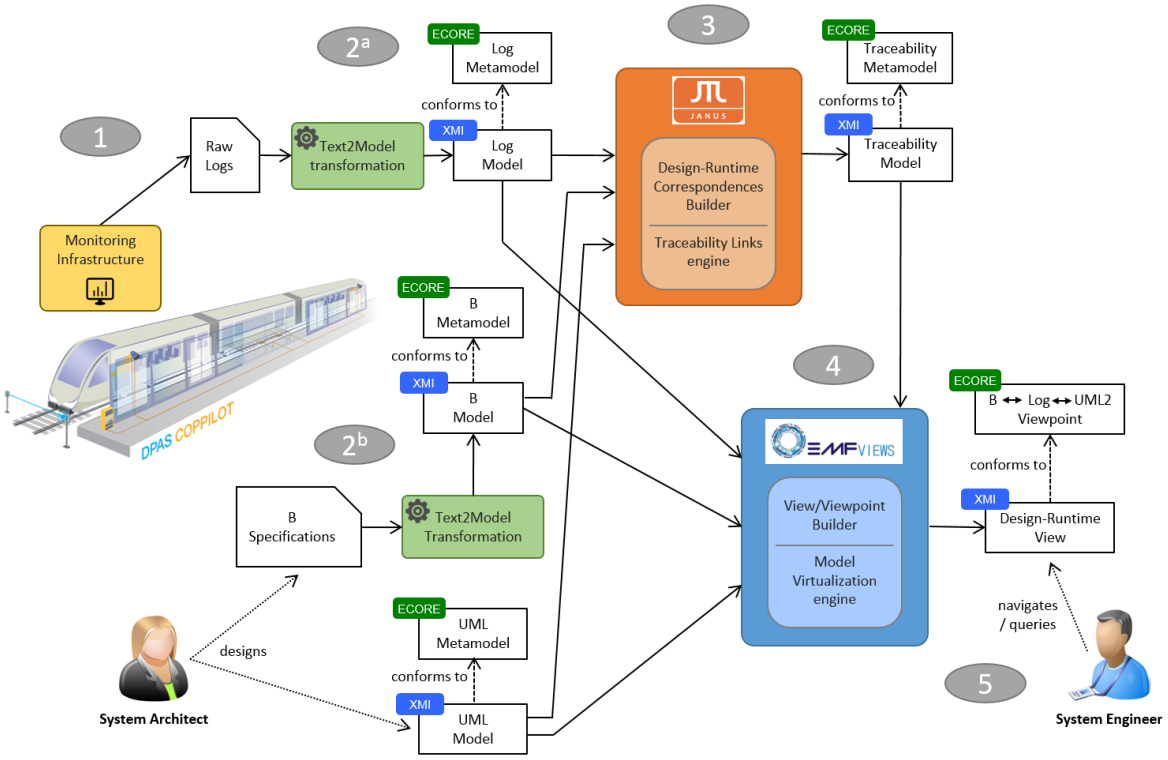



Fig. 6.2 Implementation of our model-based approach from Figure 6.1.

An overview of our technical solution, implementing the approach from Section 6.2, is depicted in Figure 6.2. The large orange and blue boxes represent the model-driven frameworks that have been used to build the traceability links and related model views. They are JTL and EMF Views [32].

EMF Views is an Eclipse plugin allowing to aggregate several (heterogeneous) models together in a view. A view can contain virtual associations that do not exist in any input model, and that can be used to link elements coming from different related models. Elements from input models can also be filtered out, making views useful for presenting information to the end user.

The *Monitoring Infrastructure* that has been used to monitor the system under analysis is represented by the yellow box in the left part of the figure. Both the design artifacts and the runtime data need to be mapped in their corresponding model-based representation (in Ecore format). Thus, Log models and B models have been automatically obtained by means of model discoverers implemented as *Text2Model Transformation*s (as represented by the green boxes in the figure). Whereas, the UML component diagram that we used in the experiments has been directly modeled by the system architect within EMF. The main steps of our approach, as previously introduced in Section 6.2, have been implemented as follows[6].

### 6.3.1 Monitoring the considered system

The CLEARSY monitoring infrastructure is realized by means of a specific monitoring system connected to Coppilot. The monitoring system is able to collect messages exchanged by components as well as memory dumps of the safety core. In particular, such raw logs represent the output data produced by the safety core and consist of concatenations of binary messages, time-stamped by the sensors, and arrays of internal variables of the safety core.

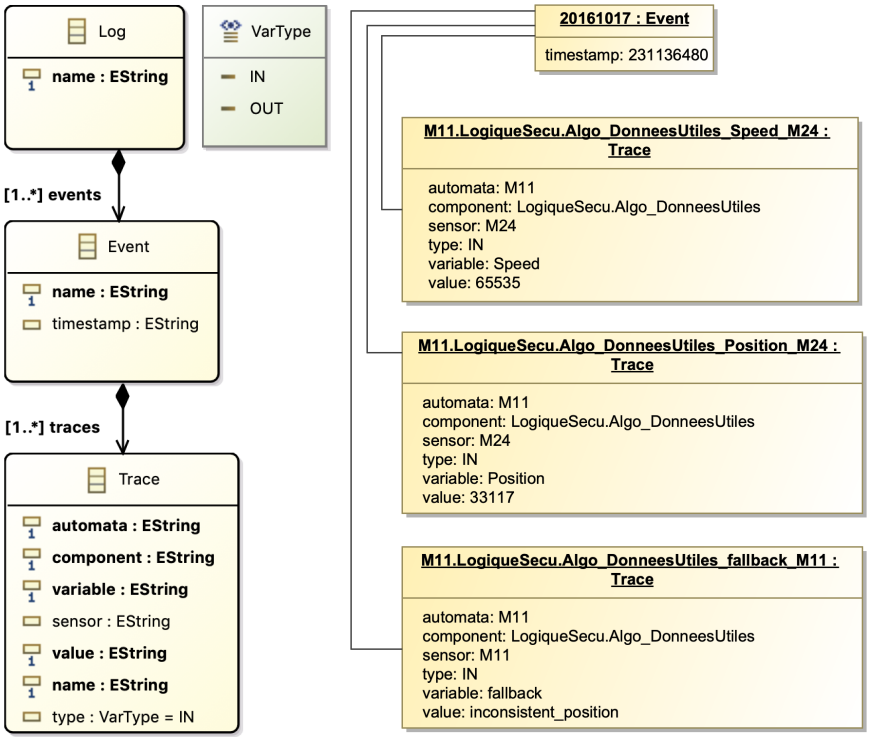
For the purpose of this collaboration, a simplified public version of the safety core specification has been produced by CLEARSY engineers, still in the B language. This specification is voluntarily simpler than the original one and implements fewer safety properties. CLEARSY collected the raw logs used in this experiment by animating this new version of the B specification of the safety core and using the previously described monitoring infrastructure.

### 6.3.2 Discovering design and runtime models

The collected runtime information and the system design need to be integrated in the EMF-based environment and translated into EMF artifacts. In this step, we

[6]The full implementation is available for download at: https://raw.githubusercontent.com/MDEGroup/JTL/master/downloads/Model-driven.Design-Runtime.Interaction.in.Safety.Critical.System.Development.zip

describe how raw logs and the B specification are automatically transformed into EMF-compatible models that conform to metamodels specified in Ecore. Once it is set up a first time with our support, this process can be run again by CLEARSY as many times as needed. We also describe how we manually created the UML component diagram from an informal architecture design provided by CLEARSY.




(a) Log metamodel. (b) A sample of Log model.

Fig. 6.3 Model-based representation of logs.

### From raw logs to Log models

The raw logs obtained from the infrastructure described in the previous section have to be specified by means of a model-based representation. To this aim, we have defined a dedicated metamodel, as depicted in Figure 6.3a. It starts with *Log*, which is the root element of a log model. A Log stores *Event*s that are characterized by a *name* and the *timestamp*. Each *Event* contains a set of *Trace* elements that stores the following information: *automata* name, *component* name, *sensor* name, *variable* name and its *value*, and *type* that can be input *IN* or output *OUT*.

Figure 6.3b depicts a sample of a Log Model that represents the original logs shown in Listing 6.1. For instance, the Event (with name *20161017*) represents one of the

event in Listing 6.1 and is composed of three Traces that refer to the automata *M11*. In particular, the first one refers to the sensor *M24* and to the variable *Speed*, the second one refers to the sensor *M24* and to the variable *Position*, whereas the last one refers to the variable *fallback* with value *inconsistent_position*.

The Log model (in XMI format) that conform to the Log Metamodel is automatically generated from the original raw log by an automatic Java text to model transformation.

### From the B specification to B models

The safety core of the Coppilot system, which contains the logic for detecting and triggering fallback situations, is written in the B language [2]. For this experiment, we focused on a simplified version of this safety core, which comprises 640 lines of B specification across 11 files, while retaining the ability to trigger fallback situations that we are interested in. The excerpt in Listing 6.2 shows a simple operation that can trigger different fallback situations (inconsistent position or speed) depending on the state of other components.

```
1  fallback <-- aopp_authorization =
2  VAR
3      train_in_par, train_stopped, doors_opening
4  IN
5      train_in_par <-- is_train_in_par;
6      train_stopped <-- is_train_stopped;
7      doors_opening <-- are_doors_opening;
8
9      IF
10         train_in_par = INCONS_POSITION
11     THEN
12         fallback := INCONSISTENT_POSITION
13     ELSIF
14         train_stopped = INCONS_SPEED
15     THEN
16         fallback := INCONSISTENT_SPEED
17     ELSE
18         fallback := NOMINAL
19     END
20 END
```

Listing 6.2 Excerpt of B specification from the simplified safety core.

In order to include the B specification in the view intended for the system engineer, we need to turn this B specification into an EMF-compatible model. To that end, we used a model2text framework. In particular, we wrote an Xtext [51] grammar for the

subset of the B language that was used by the simplified safety core. Starting from such a grammar consisting of 382 lines of code and 71 rules, the Xtext supporting tools generated an Ecore metamodel and all the related plugins allowing one to open any B file of the simplified safety core as an EMF-compatible model. The B specification could now be navigated and queried and, more importantly, it can be used as an input for building the traceability links.

### UML modeling

Starting from the informal architecture design provided by CLEARSY, we designed a UML component diagram within EMF-Eclipse as shown in Figure 6.4. The diagram describes the components involved in the simplified Coppilot system: the *M11 Computing Unit* represents the main component that exploits the information obtained by the sensors. Sensors are represented by further components, called *M21 Positioning LCU1*, *M22 Door LCU1*, *M24 Positioning LCU1*, and *M23 Door LCU1*.

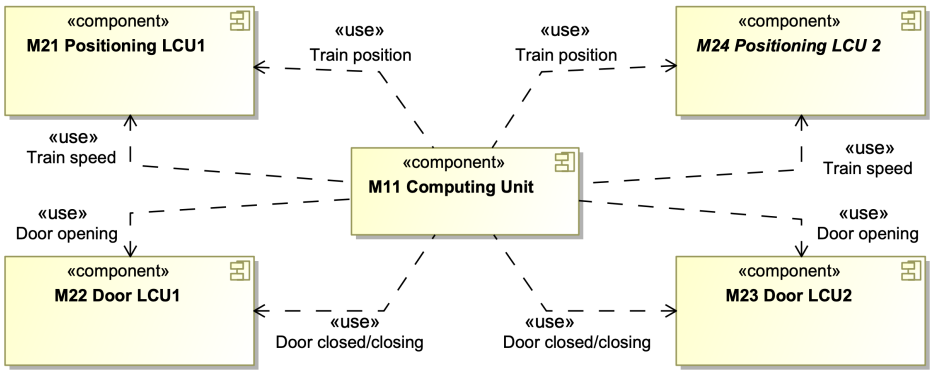



Fig. 6.4 UML Component Diagram of the simplified Coppilot system.

## 6.3.3 Computing design-runtime traceability links

Starting from the runtime and design models, a Traceability Model is automatically generated by means of JTL. In order to do this, we initiated the solution by specifying required inter-model correspondences based on the inputs from CLEARSY. This can be run again and/or extended by CLEARSY in the future if needed to consider more traceability scenarios.

```
1 transformation Log2B (log:Log, b:B) {
2     top relation Trace2Variable {
3         v, s : String;
4         checkonly domain log trace: Log::Trace {
```

```
5               variable = v,
6               sensor = s
7           };
8           checkonly domain b var: B::Variable {
9               name = s + "_" + v
10          };
11      }
12      top relation Trace2Ref {
13          v, s : String;
14          checkonly domain log trace: Log::Trace {
15              variable = v,
16              sensor = s
17          };
18          checkonly domain b ref: B::Ref {
19              var = var: B::Variable {
20                  name = s + "_" + v
21              }
22          };
23      }
24      top relation Trace2Print {
25          v, s, i : String;
26          checkonly domain log trace: Log::Trace {
27              variable = v,
28              sensor = s,
29              value = i
30          };
31          checkonly domain b ref: B::Call {
32              op = op: B::Operation { name = "print" },
33              args = a1: B::StringLiteral { value = s },
34              args = a2: B::StringLiteral { value = v },
35              args = a2: B::StringLiteral { value = i }
36          };
37      }
38  }
```

Listing 6.3 Specification of Log2B correspondences.

In Listing 6.3, a fragment of the correspondence specification between Log models and B models is reported. In particular, at Line 1 of Listing 6.3, variables *log* and *b* are declared to match models conforming to the Log and B metamodels, respectively. The specified relations are described as follows:

- The top relation *Trace2Variable* (Lines 2-11) maps an element of type *Trace* in the *Log* domain and an element of type *Variable* in the B domain. Thus, each

event that involves a sensor $s$ and a variable $v$ is mapped to the correspondent portion of the B specification where the variable is declared;

- The top relation *Trace2Ref* (Lines 12-23) maps an element of type *Trace* in the Log domain and an element of type *Ref* that represents a reference to a *Variable* element in the B domain. Thus, each event that involves a sensor $s$ and a variable $v$ is mapped to the correspondent portion of the B specification where the variable is called;

- The top relation *Trace2Print* (Lines 24-37) maps an element of type *Trace* in the Log domain and an element of type *Operation* with name *print* in the B domain. Thus, each event that involves a sensor $s$ and a variable $v$ is mapped to the operation that prints it, to establish the origin of the event.

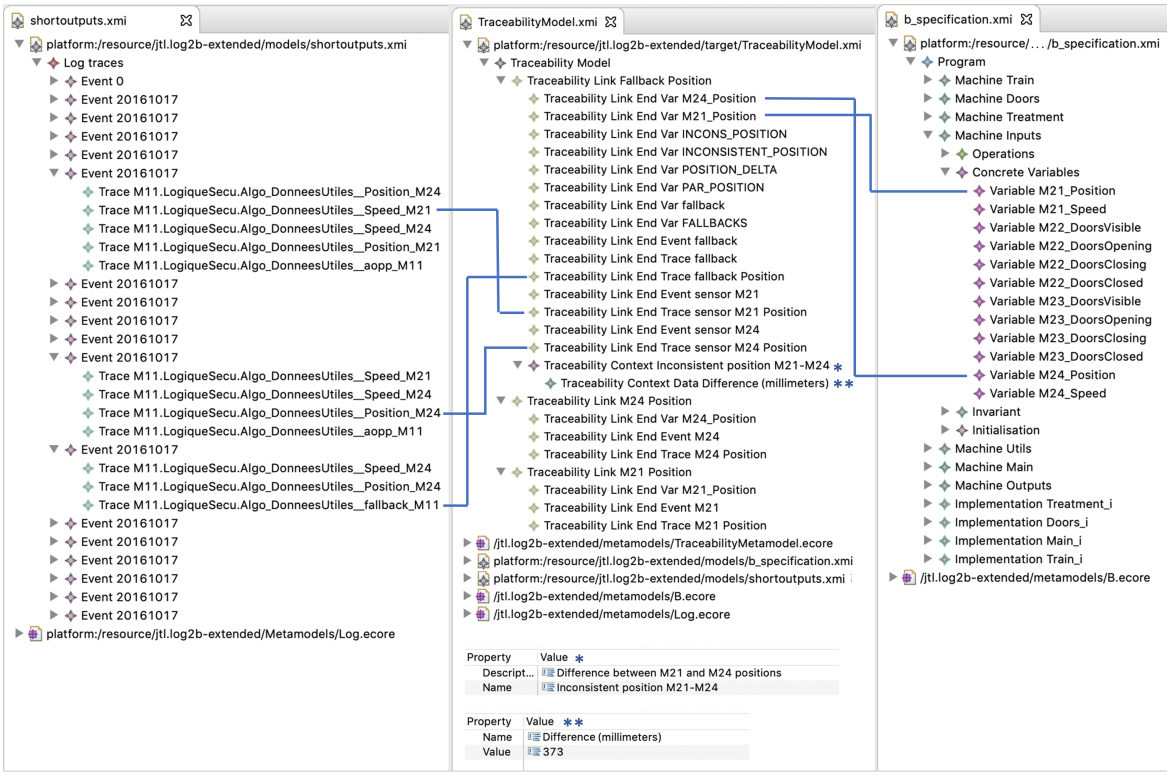



Fig. 6.5 Traceability model between the Log and B models.

The described mapping assumes that models are consistent with the formal specification. In this case study, models and code are also consistent in terms of the adopted naming convention.

The execution of the *Log2B* transformation takes as input the Log model and the B model (as shown in the left and right part of Figure 6.5) and generates the

corresponding Traceability Model (as shown in the middle of Figure 6.5). In particular, the arrows connect trace links with the source and target model elements they refer to. Furthermore, a *Context Data* containing the difference between the positions reported by the sensors M21 and M22 is created by means of a procedure able to navigate the log models and detect the last values from sensors (as explained in Section 6.1).

```
1  transformation Log2UML (log:Log, uml:UML) {
2      top relation Sensor2Component {
3          s : String;
4          checkonly domain log trace: Log::Trace {
5              sensor = s
6          };
7          checkonly domain uml comp: UML::Component {
8              name = s
9          };
10     }
11     top relation Automata2Component {
12         a : String;
13         checkonly domain log trace: Log::Trace {
14             automata = a
15         };
16         checkonly domain uml comp: UML::Component {
17             name = a
18         };
19     }
```

Listing 6.4 Specification of Log2UML correspondences.

In Listing 6.4, a fragment of the correspondences specification between Log models and UML models is reported. In Line 1, variables *log* and *uml* are declared to match models conforming to the Log and UML metamodels, respectively. The specified relations are described as follows:

- The top relation *Sensor2Component* (Lines 2-10) maps an element of type *Trace* in the Log domain and an element of type *Component* in the UML domain. Thus, each event that involves a sensor $a$ is mapped to the corresponding component named $a$;
- The top relation *Automata2Component* (Lines 11-19) maps an element of type *Trace* in the Log domain and an element of type *Component* in the UML domain. Thus, each event that involves a sensor $a$ is mapped to the corresponding component named $a$.

The execution of the *Log2UML* transformation takes as input the Log and UML models (as shown in the left and right part of Figure 6.6) and generates the corresponding Traceability Model (as shown in the middle of Figure 6.6). In particular, the arrows connect trace links with the source and target model elements they refer to.

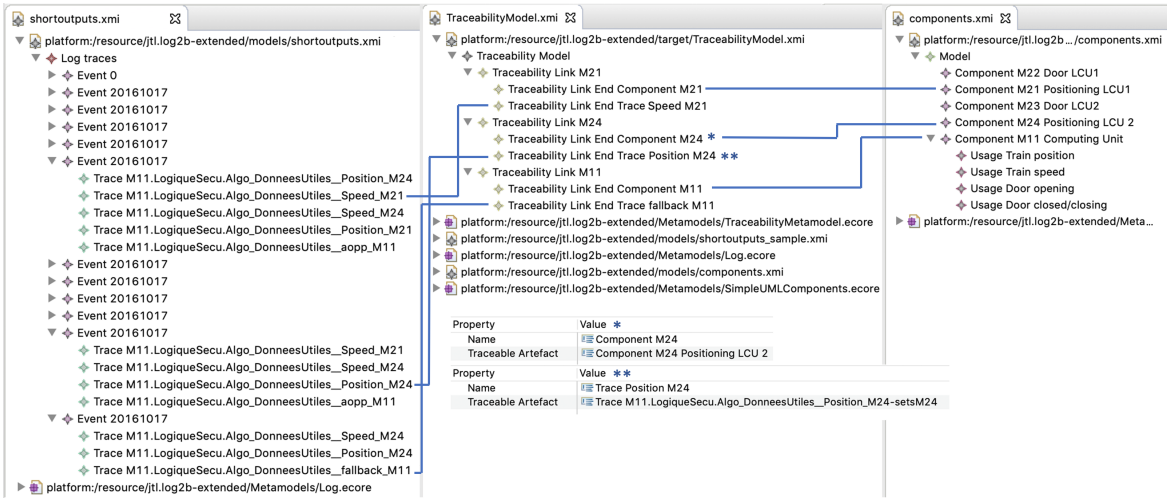



Fig. 6.6 Traceability model between the Log model and the UML component diagram.

### 6.3.4 Building the design-runtime traceability view

While the traceability links computed in the previous section provide enough information to help localize and diagnose fallbacks, they are not intended to be used directly by the system engineer. Instead, we propose to build a view that aggregates together all the models seen so far. This allows the system engineer to transparently point to the relevant information (spread in different models) while also allowing him/her to have a better vision of the full picture. We initiated such a view for our use case with the help of CLEARSY engineers. However, they are now able to refine this view and eventually define new ones for supporting different scenarios.

Technically, we built the view using EMF Views by using four different artifacts:

1. A *viewpoint description*, in which we list the metamodels the view requires. In our case, the metamodels are: the B metamodel, the Log metamodel, the Traceability metamodel, and the UML metamodel.

2. A *viewpoint weaving model*, which describes the new (virtual) features enriching the view. We add a new bidirectional association *designComponent*, between *Trace* and *Component*, which allows the system engineer to navigate from a

runtime trace back to the component that emitted it, and from a component to all its emitted traces.

We also rename the features *leftLinkEnd* (which holds references to B variables) and *rightLinkEnd* (references to events and traces) to *specification* and *events*, making the view more useful to the engineer. We perform such a renaming by filtering the existing features and creating new associations; the matching model described below then simply carries their content over.

3. A *view description*, where we list the models contributing to the view: the model of B specification, an excerpt of logs, the traceability links computed in 6.3.3, and the components model. The view description also points to the viewpoint description above, and to the matching model.

4. A *matching model*, where we describe how the virtual features should be populated. When loading the view in a model browser, EMF Views will use the matching model to compute an internal view weaving model (analogous to the viewpoint weaving model), and populate the virtual features accordingly.

   Different languages can be used for the matching model. In the present case, we use the Epsilon Compare Language (ECL) [74]. Listing 6.5 shows the ECL rule for populating the *designComponent* feature: if the *name* of the component (e.g., *M21 Position LCU*) starts with the *sensor* property of the trace (e.g. *M21*), the two are virtually linked in the view.

```
1 rule designComponent
2     match t : log!Trace
3     with  c : uml!Component {
4     compare {
5         return c.name.startsWith(t.sensor);
6     }
7 }
```

Listing 6.5 ECL matching model.

With these four artifacts, the system engineer can open the view description in a model browser. This causes EMF Views to first build the corresponding viewpoint, by loading the contributing metamodels and the viewpoint weaving model in order to build the virtual features. Then, EMF Views builds the actual view by loading the contributing models, executing the matching model, and populating the virtual features by collecting the matches. In this case, the process of building the view takes

less than a second, because the virtualization engine of EMF Views does not copy elements of the contributing models into the view, but rather creates proxies to access to them on-demand.

While four files are required to build the view, building a new view on another runtime trace only requires to modify the view description by pointing out to the new runtime trace model.

### 6.3.5 Navigating and querying the design-runtime view

The resulting model view can be navigated using the different types of available user interfaces. In our Eclipse context, opening the view description file with a standard model browser yields all the information the CLEARSY system engineer needs in order to analyze the fallback situations. However, for a more complete user experience, view navigation capabilities could also be integrated directly within the editors of the various contributing models (cf. Section 6.4.3).

Figure 6.7 gives an example of such a view on a trace where an inconsistent position fallback was triggered (still on the example from Section 6.1.2). On the left-hand side are the four models: the Log model containing all the output traces, the UML component model containing the design components, the Traceability model containing interesting traces (here there are three), and the B model containing the B specification. On the right-hand side, one traceability link is expanded so that its child feature are visible: *traceabilityContext* summarizes the situation of the fallback in plain text, *specification* points to B variables that are related to the fallback situation, and *events* contains the events and traces that are causes for the fallback, as well as the trace in which the fallback actually happened. One can furthermore consult the actual values of the involved sensors M21 and M24 and see that their difference is 373, which is above the threshold required to trigger a fallback situation (the threshold of 300 is displayed when expanding the traceability context). From the traces, the engineer can also follow the *designComponent* link to navigate to the UML component of the corresponding sensor.

In addition to navigating the model view manually, the system engineer can also query this view by using either dedicated query languages or more general-purpose languages she/he is familiar with. Queries can be useful notably for automating workflows, i.e. to support collecting data that will be processed by other tools, or to create useful summaries. In particular domains or for given purposes (cf. our case study for instance), it is possible to define some reusable sets of such queries. However,

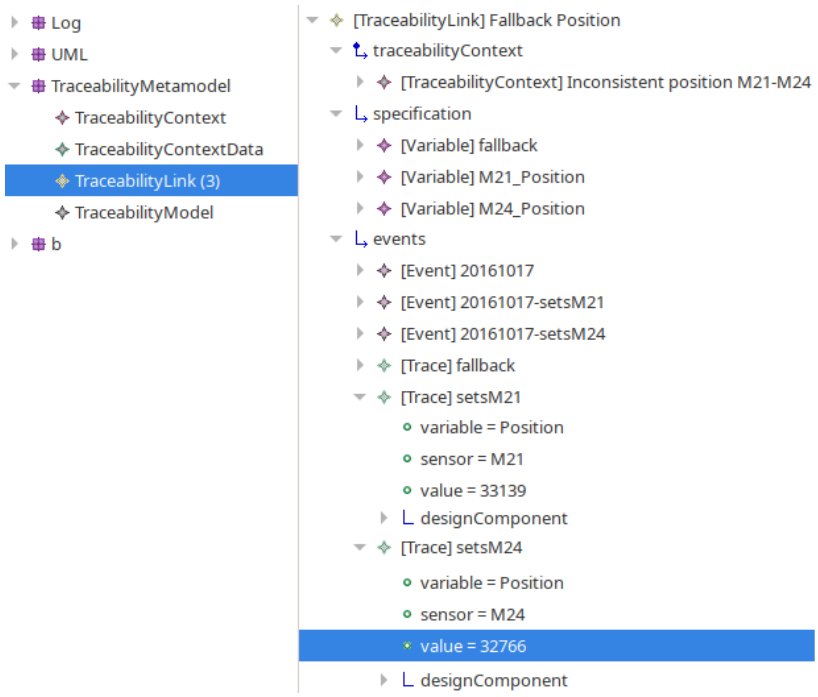



Fig. 6.7 View showing an inconsistent position between the two sensors M21 and M24.

this cannot be systematically performed in the general case and the engineers expertise is still fundamental in order to write down more complex queries.

As a concrete simple example, the OCL query of Listing 6.6 will list all fallback situations by date alongside with a one-line summary of this fallback (e.g. "Inconsistent position M21-M24"). Note that we have only experimented with queries defined in OCL so far, but other languages could also be used to the same intent (cf. Section 6.4.3).

```
1 TraceabilityLink.allInstances()
2   ->select(t | t.name.startsWith('Fallback'))
3   ->collect(t | t.events->asSequence()
4     ->first().timestamp + ' : ' + t.traceabilityContext.name)
```

Listing 6.6 Example OCL query to summarize fallback situations from a view.

As another example, the query `self.traces.values` can be used to extract all the sensor values for a given design component (the value of `self`). These values could then be further analyzed by the system engineer to diagnose hardware failures. Recall that *traces* is a virtual association that links UML components to *Trace* element of the log model. The fact that the view aggregates all models into one makes such queries easy to write and execute.

```
1 TraceabilityLink.allInstances()
```

```
2   ->select(t | t.name = 'Fallback Position')
3   ->collect(t | t.events->first().timestamp + ': ' +
4     t.traceabilityContext.name + ' : ' +
5     t.events->select(e | e->isTypeOf(Trace))
6   ->select(t | t.name.indexOf('setsM') > 0)
7   ->collect(t | t.designComponent.name + ' : ' +
8     t.value)->concat(', '))->println();
```

Listing 6.7 Epsilon query to analyze fallbacks due to inconsistent sensor positions.

Listing 6.7 is a last example, this time using the Epsilon Object Language [101]. We output one line for each fallback situation created by an inconsistent position between sensors, along with the event timestamp and last known sensor values. When run against the log of Listing 6.1, we get the line below, providing all the information that was previously manually extracted.

```
231136106: Inconsistent position M21-M24: M21 Positioning LCU1:
   ↪ 33139, M24 Positioning LCU2: 32766
```

## 6.4 Discussion

The experiment has been conducted in two complementary phases. According to the CLEARSY interest in adopting model-based solutions, the first phase of the experiment was devoted to 1) the study of their practices as currently operating in the company and 2) the understanding of how their needs could be possibly supported by our technologies. From this first phase, it was evident that a significant effort would be required to integrate model-based methodologies directly within the existing internal development processes at CLEARSY (as already strictly structured) .

After collecting the required experiment material, the second phase has been dedicated to the design of the conceptual approach and the implementation of the corresponding technical solution (as described in Section 6.3). Thanks to the performed experiments, we have been able to integrate various existing artifacts from CLEARSY within the EMF Eclipse platform. This way, we contributed to the apparition of a new model-based process at CLEARSY complementary to the existing ones. This one is specifically dedicated to the analysis of runtime information internally collected by the company, and can be possibly generalized to other critical systems they have to model (in B) for their different customers.

We now highlight the strengths and the limitations of the proposed approach. This is motivated by the feedback received from CLEARSY, based on the usage they have been able to make so far of our technical solution.

### 6.4.1 Benefits of the approach

Our initial experiment indicates that the use of traceability links between runtime and design/architectural aspects is promising. CLEARSY has observed that providing accessible correlations between situations of failures and the corresponding design elements can help their system engineers to better understand critical issues occurring at runtime. In fact, our approach has the potential for faster turnaround when analyzing fallback situations: provided traceability links highlight such situations and link directly to their root cause. Since system engineers can also jump from a fallback trace to the related variables in the B specification, and also to the corresponding element in the component diagram, the provided view can help them understanding if fallback situations are actually caused by specification bugs or rather by involved hardware components.

As a side benefit of discovering models from the raw (log) data, the structured information aggregated in the view can be directly reused by model-based tools for different complementary engineering activities (e.g. for automated system documentation or execution reports).

### 6.4.2 Limitations of the approach

One technical limitation of our current implementation is that the discovery of the log model needs to be done manually. The transformation of raw logs into Log models is automatic, but the creation of Log models need to be triggered separately from other steps of the process. In contrast, the B code is discovered automatically as a model by the Xtext plugins when loading them as EMF resources. The same principle could be applied to Log models, but currently requires a separate step.

Another limitation of our current solution is that the final view is not yet deeply integrated with some tools the system engineer may be familiar with. For instance, while the view lets you browse the B specification as a model (somehow equivalent to an AST), the system engineer may be more comfortable working directly with the B source code. The view also currently lets you see UML components and their usage relations, but not in a dedicated UML graphical editor. In other words, the view exposes a model representation of the aggregated design-runtime information, while the system engineer may be more used to deal with corresponding concrete (textual or graphical) syntax. How to integrate the model view elements seamlessly in order to enhance existing concrete syntax remains an open scientific and technical challenge. Similarly to what has been done in [3], we plan to overcome such limitations by employing bidirectional

model transformations to define and implement the relationships among the concepts belonging to the different concrete syntaxes of the modeling artifacts at hand.

### 6.4.3 Planned technical improvements

In the context of this first experiment, we focused on detecting one specific cause of fallback situations: when two sensors report inconsistent values. However, a fallback may also be caused by faulty sensors or other types of conditions. To detect these other causes in our current solution, we need to specify more general patterns that can then be used by JTL when computing the traceability links. Supporting CLEARSY in this process, we plan to study how to (semi-)automatically discover such patterns based on the raw runtime data.

As far as the model view navigation is concerned, more work could be done on improving the integration with the different already available editors (cf. also Section 6.4.2). For example, in our use case, it would be interesting to be able to navigate directly from an entry in the Log model back to actual corresponding textual elements in the B source code (without having to actually open and browse the view itself).

Concerning the model view querying aspects, an interesting continuation of our work could be to test the support for other languages than OCL to define the needed queries. In the context of our use case, we can work in close collaboration with the CLEARSY system engineers in order to identify the language(s) (existing or to be specified) which are possibly better-suited for them to write and maintain such queries on the long run.

## 6.5 Related work

The presented work directly belongs to the area of design-runtime interactions in complex systems, as already prominent in the context of CPSs [44]. Among the different underlying challenges [31], our proposed approach and related experiments are mostly related to the following main topics in this area: *1)* the discovery and representation (as models) of runtime information and *2)* the creation and use of correspondences between runtime and design models.

The idea of extracting relevant information out of execution logs is not new and has already been used quite a lot in other domains, such as in databases [5]. As far as software execution logs are concerned, it appears that there is not any standard representation that emerged even if some common formats have been proposed quite

recently (e.g., the Common Trace Format). As a consequence, and also due to the relatively simple structure of the raw logs we were able to collect from the studied system, we decided to opt in our approach for a simple custom Log metamodel we designed ourselves.

Complementary to log modeling, some existing approaches intend to represent system runtime behaviors with more details, either in the general case [84] or by means of common notations such as dataflow diagrams [91]. Similar approaches are also used when the final goal of the runtime analysis process is to perform more formal verification activities, based on UML statecharts [46]. However, in the present case and practical scenario, we have been more interested in studying the produced runtime logs than in defining the system behavior in a more general and abstract way.

Concerning the definition and representation of the inter-model correspondences themselves, there has already been a significant work in the community on supporting model-based principles and techniques such as model weaving, for instance in the context of model transformation [52] or language interoperability [69]) or model comparison [74].

Finally, formal methods are widely used at CLEARSY especially using the B language, notably at the design time, for the specification of models or systems [83]. Some techniques are also applied at runtime, such as formal validation [82]: Rules or properties are defined over a model and large data sets are validated against this model. For example, this allows to detect inconsistencies or design flaws in sets of software parameters.

# Chapter 7

# A Study on the Impact of Maintainability Refactoring on Execution Time

State-of-the-art refactoring recommenders target the improvement of code quality from a narrow perspective, focusing on improving code readability or removing well-known anti-patterns or code smells [122, 92, 15]. Basically, they aim at improving code maintainability without considering the possible side effects that the recommended refactorings may have on other, maybe more important, non-functional requirements. In other words, they do not consider the priority that different non-functional requirements may have. For this reason, some researchers started investigating the impact of "maintainability-driven" refactorings on other non-functional attributes.

In this chapter, we present a study that is the result of an ongoing collaboration among our whole research group at University of L'Aquila, the University of Molise, and the Software Institute of the Università della Svizzera Italiana. This study has the objective of investigating the impact of 16 different types of refactoring on the execution time of 20 Java systems. Using RefMiner [124], we mined the subject systems for "refactoring commits", i.e., commits that contain refactoring operations. We manually inspected each commit to ensure that refactoring was its only goal. Through dynamic code analysis we identified the code components executed by the performance benchmarks coupled to each system, and the refactoring operations that impacted them. Overall, we collected 82 commits implementing 167 refactoring operations impacting performance-relevant components. Each commit provides several data points for our study, since refactorings implemented in the same commit can impact different performance-relevant components and, thus, exercise different performance

benchmarks. The total number of data points involved in our study (i.e., pairs of refactoring actions, benchmarks) is 1,598. The collection of this data required ∼476 machine days. Besides presenting quantitative results that show the impact of (different types of) refactoring on execution time, we also qualitatively analyze cases in which refactoring had a negative impact on execution time, thus distilling lessons learned useful to (i) developers, for avoiding specific refactoring scenarios when performance is key, and (ii) researchers, for developing performance-aware refactoring recommenders.

The contents of this chapter are based on a submitted article [120], which is currently under review.

## 7.1 Data collection and analysis

The *goal* of the study is to investigate the impact of refactoring operations on software performance. Measuring performance encompasses multiple metrics, such as response time, utilization, etc. In the context of this chapter, we focus on execution time, intended as the time that a section of code needs to be executed, without any concurrency and/or resource sharing with other software running on the same platform.

In order to collect the data needed for our study, we have: (i) selected Java open-source projects with performance benchmark suites, (ii) detected refactoring operations to assess their performance impact, and (iii) run benchmarks before and after the refactoring operations were performed.

### Projects selection

We selected projects in which developers defined micro-benchmarks for performance assessment. To do so, we queried GitHub for Java projects having a dependency with Java Microbenchmarking Harness (JMH)[1], the *de facto* standard for micro-benchmarks.

We used the GitHub APIs to obtain the list of the 1,000 most recently indexed projects that (i) used Maven as the dependency manager, and (ii) had an explicit dependency with `org.openjdk.jmh.jmh-core`, i.e., the core library required to run JMH. We considered only projects having at least 100 stars, and 88 projects satisfied this criterion. In addition to them, we considered two popular Java projects already used in a previous microbenchmark-related study [79], i.e., RxJava and Log4j2.

We manually analyzed the list of projects to find the commands that would build and run the benchmark suite. We were able to identify working commands and runnable benchmarks for 31 projects (including RxJava and Log4j2).

[1] https://openjdk.java.net/projects/code-tools/jmh/

### Refactorings and benchmarks collection

We used RefactoringMiner [123] to extract refactoring operations performed on the default branch of each project. We did not consider seven refactoring types that likely have negligible or no performance impact: *Rename-related* refactoring operations (Rename Method, Rename Class, Rename Variable, Rename Parameter and Rename Attribute) and *package-related* refactoring operations (Change Package, Move Class).

We collected 494,826 maintainability refactoring operations (by 48 different types) performed in 181,020 commits and 31 projects. We identified benchmarks suitable to evaluate their performance impact, by verifying for each benchmark in the project whether the code affected by refactoring operations is executed by it. We achieved this by running each benchmark (for 1 second) and recording the methods invoked in the execution using Java Flight Recorder (JFR)[2]. In this way, we identified an initial set of 3,533 data points, each one defined by the project name, a commit, a benchmark, and a set of refactoring operations. This set of data points was then manually filtered to exclude those cases in which the refactoring operations are not the only changes affecting the code executed by the benchmarks. The dataset resulting from this process contains 1,534 data points involving 69 commits, 150 refactoring operations, and 16 refactoring types across 17 projects.

### Benchmarks execution

The performance comparison of different software versions in Java applications is far from being trivial. There are a number of sources of non-determinism, such as Just-In-Time (JIT) compilation and optimizations in the Java Virtual Machine (JVM) [55]. We relied on steady state performance [55]: we repeated a benchmark execution for several *iterations* and collected measurements only after a steady state had been reached. We discarded measurements obtained in the first iterations, also called *warm-up iterations*, to avoid noise due to performance variations in transient states, usually caused by class loading and JIT (re)-compilation. Different VM invocations running multiple benchmark iterations may result in different steady-state performance data. For this reason, we also repeated benchmark iterations multiple times on different VM invocations. JMH allows defining the number of warm-up iterations, measurements iterations and VM invocations directly in Java code or via command line arguments. We used the number of iterations defined in the code by benchmark developers for warm-up and measurement iterations, and we fixed the number of VM invocations to

[2] https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm

10 (i.e., the JMH default) as in previous studies [55, 78]. The execution of benchmarks for the 1,598 data points of our study required 79 machine days on a dual Intel Xeon CPU E5-2650 v3 at 2.30GHz, totaling 40 cores and 80GB of RAM. The benchmarks were executed on Ubuntu Linux, kernel version 4.15.0.

#### Reliably detecting performance change

In order to determine whether refactoring operations lead to non-negligible performance change, we used the approach proposed by Kalibera & Jones to build confidence intervals for ratio of mean execution times [70, 71]. To build the confidence interval we used bootstrapping [77], with hierarchical random re-sampling [107] and replacement. Re-sampling was applied on two levels [71]: VM invocations and iterations.

We ran 1,000 bootstrap iterations. At each iteration, we obtained new measurements for the previous and refactored commit from re-sampling and we computed the relative performance change. After the termination of all iterations, we collected a set of simulated realizations of the relative performance change and estimated the 0.025 and 0.975 quantiles on it, hence for a 95% confidence interval. Given a commit, a refactoring operation leads to a *regression* for a benchmark if the lower limit of the confidence interval for relative performance change of mean execution times is greater than 0 (i.e., the benchmark becomes slower after the commit). Similarly, there is an *improvement* if the upper limit of the confidence interval is less than 0 (i.e., the benchmark is faster before the commit). Otherwise, we consider performance as *unchanged*.

## 7.2 Results

In this section we summarize the insights that we have gained from answering to several research questions.

#### $RQ_1$: To what extent do developers refactor performance-relevant code components?

To answer $RQ_1$, we compared the density of refactoring operations in performance-relevant code to the one in other parts of the system.

In only two projects (i.e., `camel` and `drools`), developers performed more refactoring operations on performance-relevant methods. For all other projects, the refactoring density is higher in performance-non-relevant methods. Among those, only two projects (i.e., `jooby` and `vert.x`) have a p-value larger than 0.05 (i.e., the difference is not

statistically significant according to the Fisher's exact test). For all other projects, the refactoring density difference is statistically significant. This result indicates that, in most projects, the density of refactoring operations is higher in non-performance relevant methods.

**$RQ_2$: What is the impact of refactoring on performance?**

In this RQ, we consider the dataset gathered from the first round of data collection.

Most refactoring-related commits lead to performance change, with these changes usually affecting only a subset of the involved benchmarks. Moreover, we found that a large percentage of commits (>55%) leads to regression in at least one benchmark. Performance regressions and improvements due to refactoring-related commits have relatively similar frequencies, and they can bring a performance change up to 12% in most of the cases. Finally, our results indicate that the analysis of the performance impact of refactoring activities may be non trivial even for experienced developers, as these changes can have diverse (and often mixed) effects on performance-relevant methods. This problem is further exacerbated by the long execution time required to run benchmark suites, which may prevent developers from verifying the performance impact of their refactoring operations.

**$RQ_3$: What types of refactoring operations are more likely to impact performance?**

To answer $RQ_3$, we need to isolate the effect of different refactoring types on software performance. We selected from our dataset the data points having all refactoring operations of the same type. We analyzed 1,156 data points from 18 systems involving 7 refactoring types: Extract Method (166 data points), Extract Superclass (90), Inline Variable (65), Extract Class (398), Move Method (184), Inline Method (66), and Extract Interface (187).

Figure 7.1 reports the percentage of benchmarks in which the performance is positively or negatively impacted by each type of refactoring operation considered in $RQ_3$.

The chart reveals that all of the refactoring types can lead to both improved and regressed performance. Overall, Extract Class/Interface/Method/Superclass refactoring operations are more likely to impact performance than Inline Method/Variable and Move Method. When performing Extract Class/Interface/Method and Inline Method the performance is more likely to degrade, while when performing Inline Variable and Move Method there is a higher chance of performance improvement. Extract Superclass

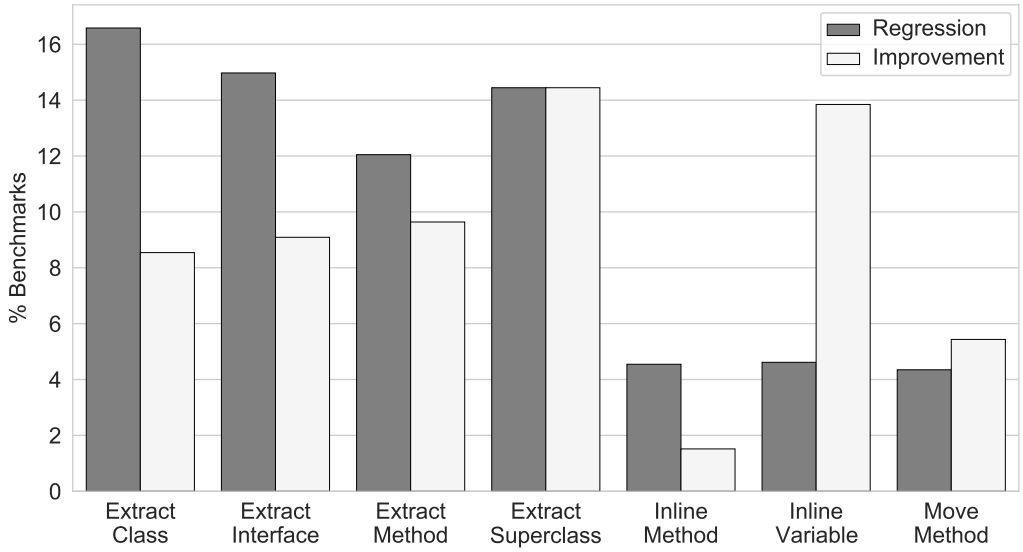



Fig. 7.1 $RQ_3$. Performance impact of different types of refactoring on the associated benchmarks (i.e., data points). Percentages of benchmarks showing regression or improvement are reported for each refactoring type.

leads to similar amounts of performance regression and improvement. The Extract Class refactoring is the most closely related to performance regression, with more than 16% of impacting benchmarks showing such a trend. Moreover, the magnitude of regression introduced by Extract Class is higher when compared to other types of refactorings (50% of regressions lead to a performance change between 2% and 7%). Inline Variable has a relatively high chance to lead to performance improvement (14% of the benchmarks with a performance change ranging from 3% and 6% in 50% of the cases), while Move Method and Inline Method have lower chances to bring performance change. Moreover, the performance change caused by the Move Method has never reached 5%.

In summary, the impact of refactoring on performance varies from type to type. No refactoring type guarantees the absence of performance regression. Extract Class and Extract Method have a higher chance of causing larger performance regression than other types of refactoring. When Extract Method causes performance improvement, it leads to larger performance changes.

## 7.3 Afterthoughts

The achieved results show that the impact of refactoring on execution time varies depending on the refactoring type, with none of them being 100% "safe" in ensuring performance regression absence. Some refactoring types, such as Extract Class and Extract Method, can result in substantial performance regression and, as such, should be carefully considered when refactoring performance-critical parts of a system.

Our findings, by disclosing the potential side-effects of refactoring on execution time, pave the way to the development of (i) approaches to predict the impact on performance of planned refactoring operations before they are actually implemented in the system, and (ii) *sensible* refactoring recommender systems, able to consider trade-offs among multiple non-functional requirements when making recommendations. Our future agenda is driven by these two research directions.

# Chapter 8

# Conclusion

The main research direction of this PhD was to find new solutions for the integration of non-functional validation in the software development process. In order to achieve this, we employed bidirectional model transformations because they seemed a promising mean to attain a seamless transition from software models to non-functional models and back. We found that, to use bidirectional transformations in this context, some aspects needed substantial improvement, like integration with modeling environments already in place, overall usability, and traceability management. We saw fit to realize such improvements by advancing the development of the JTL framework. As a consequence, we re-engineered most of it, and we developed a new traceability management engine designed to deal with the specific challenges occurring when defining bidirectional relations between traditional software models and non-functional ones. The main challenge is probably posed by the very nature of such relations. In most cases, transforming a software model into a non-functional model leads to the definition of partial and non-bijective relations. This kind of transformations are in clear contrast with the quality properties that are generally considered beneficial in model-driven engineering. In fact, the quality of a transformation is usually measured by computing the coverage of the source model, that is how many concepts of the source domain are considered in the transformation, and by evaluating the information loss induced by the transformation when more than one concept in the source domain is mapped to a single concept in the target domain. Therefore, partial and non-bijective transformations are rarely targeted by model-driven approaches, even if they arise in many practical use cases.

The first solution we presented was in the context of software availability. We introduced an approach for the generation, analysis and refactoring of Petri Nets models derived from static and dynamic views of UML diagrams. Even if Petri Nets

are frequently adopted for the assessment of availability, there was no solution to automatically propagate changes made on a Petri Net back to the software model it is derived from. We also provided a refactoring catalog for Petri Nets, consisting of four patterns based on well-known error masking techniques. We demonstrated how to apply our solution on a case study, compare the effects of different refactoring actions on availability, select the combination of refactoring that best improve the availability, and, finally, how to automatically update the UML model with the new changes. Despite the case study we chose is complex enough to show the advantages of using our approach, the main limitation probably resides in the scalability of the approach to larger and real case studies, which remains one of our future objectives.

Another challenge is represented by the ever-increasing adaptability required from modern software. Many scenarios from the industry advocate for the development of software that is able to promptly react to changes in the execution environment, especially after the deployment in production. In this context, integrating non-functional validation in the development process is even more demanding. We tried to tackle this problem from a model-driven point of view by introducing a solution for the automated generation of interactions between design and runtime information of microservices-based systems. Such interactions, implemented as traceability links, allowed us to enrich UML models with performance measures from runtime. By combining design and runtime knowledge, we detected performance antipatters on the basis of which we can suggest model refactoring actions that should improve the performance of the system. We evaluated this approach on two case studies by performing the recommended refactoring actions online on the running systems. The results show that the recommended refactoring actions always remove the targeted performance antipatterns, improve the performance of the system, and are more effective than random refactorings.

Design-runtime interactions have great potential in several domains. In fact, in the context of the MegaM@Rt2 project, we were able to provide a solution in the domain of safety-critical software systems. We applied a combination of model traceability (in JTL) and model views (in EMF Views) techniques to a real industrial use case of a railway system developed by CLEARSY. The goal was to provide support for the root cause analysis of some errors occurring only on the production system. To achieve this, we first related runtime models to design models and to the formal method specification used to develop the system, and then we generated model views to conveniently querying the models and investigate possible causes for the errors. CLEARSY observed that providing accessible correlations between situations of failures and the corresponding

design elements can help their system engineers to better understand critical issues occurring at runtime.

As we showed through an empirical study, there exist circumstances in which a quality attribute may be neglected in favor of another. In fact, by inspecting several open source software project, we found that refactoring operations that are designed to improve the code quality can have a substantial impact on execution time, both in a positive and negative way. This should motivate further research on approaches to recommend refactoring operations that consider adequate trade-offs among multiple non-functional requirements.

As future work we intend to explore the application of design-runtime interactions to different notations in other domains. We also intend to investigate how the generation of traceability links scales with very large datasets of logs, which are increasingly common nowadays. Providing an efficient solution for the parsing and classification of monitoring logs is a required step for any approach that could be considered by practitioners. Another future direction may be exploiting machine learning techniques to automatically infer design-runtime interactions, to define more complex pattern matching of sequential log data, or to provide hints for the root cause detection of deviations in non-functional properties.

# References


[1] Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., and Stevens, P. (2015). Notions of bidirectional computation and entangled state monads. In Hinze, R. and Voigtländer, J., editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, pages 187–214. Springer.

[2] Abrial, J. (1996). *The B-book - assigning programs to meanings*. Cambridge University Press.

[3] Addazi, L., Ciccozzi, F., Langer, P., and Posse, E. (2017). Towards seamless hybrid graphical-textual modelling for UML and profiles. In Anjorin, A. and Espinoza, H., editors, *Modelling Foundations and Applications - 13th European Conference, ECMFA@STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings*, volume 10376 of *Lecture Notes in Computer Science*, pages 20–33. Springer.

[4] Afzal, W., Brunelière, H., Di Ruscio, D., Sadovykh, A., Mazzini, S., Cariou, E., Truscan, D., Cabot, J., Gómez, A., Gorroñogoitia, J., Pomante, L., and Smrz, P. (2018). The megam@rt2 ECSEL project: Megamodelling at runtime - scalable model-based framework for continuous development and runtime validation of complex systems. *Microprocess. Microsystems*, 61:86–95.

[5] Agrawal, R., Gunopulos, D., and Leymann, F. (1998). Mining process models from workflow logs. In Schek, H., Saltor, F., Ramos, I., and Alonso, G., editors, *Advances in Database Technology - EDBT'98, 6th International Conference on Extending Database Technology, Valencia, Spain, March 23-27, 1998, Proceedings*, volume 1377 of *Lecture Notes in Computer Science*, pages 469–483. Springer.

[6] Alhaj, M. and Petriu, D. C. (2010). Approach for generating performance models from UML models of SOA systems. In *Proceedings of the 2010 conference of the Centre for Advanced Studies on Collaborative Research, November 1-4, 2010, Toronto, Ontario, Canada*, pages 268–282.

[7] Altamimi, T., Zargari, M. H., and Petriu, D. C. (2016). Performance analysis roundtrip: automatic generation of performance models and results feedback using cross-model trace links. In Mindel, M., Jones, B., Müller, H. A., and Onut, V., editors, *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, CASCON 2016, Toronto, Ontario, Canada, October 31 - November 2, 2016*, pages 208–217. IBM / ACM.

[8] Anjorin, A., Buchmann, T., and Westfechtel, B. (2017). The families to persons case. In García-Domínguez, A., Hinkel, G., and Krikava, F., editors, *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017*, volume 2026 of *CEUR Workshop Proceedings*, pages 27–34. CEUR-WS.org.

[9] Arcaini, P., Riccobene, E., and Scandurra, P. (2015). Modeling and analyzing MAPE-K feedback loops for self-adaptation. In Inverardi, P. and Schmerl, B. R., editors, *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*, pages 13–23. IEEE Computer Society.

[10] Arcelli, D. and Cortellessa, V. (2013). Software model refactoring based on performance analysis: better working on software or performance side? In Buhnova, B., Happe, L., and Kofron, J., editors, *Proceedings 10th International Workshop on Formal Engineering Approaches to Software Components and Architectures, FESCA 2013, Rome, Italy, March 23, 2013*, volume 108 of *EPTCS*, pages 33–47.

[11] Arcelli, D., Cortellessa, V., and Di Pompeo, D. (2018). Performance-driven software model refactoring. *Inf. Softw. Technol.*, 95:366–397.

[12] Arcelli, D., Cortellessa, V., Di Pompeo, D., Eramo, R., and Tucci, M. (2019). Exploiting architecture/runtime model-driven traceability for performance improvement. In *IEEE International Conference on Software Architecture, ICSA 2019, Hamburg, Germany, March 25-29, 2019*, pages 81–90. IEEE.

[13] Arcelli, D., Cortellessa, V., and Trubiani, C. (2015). Performance-based software model refactoring in fuzzy contexts. In Egyed, A. and Schaefer, I., editors, *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9033 of *Lecture Notes in Computer Science*, pages 149–164. Springer.

[14] Avizienis, A., Laprie, J., Randell, B., and Landwehr, C. E. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33.

[15] Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., and Lucia, A. D. (2014). Methodbook: Recommending move method refactorings via relational topic models. *IEEE Trans. Software Eng.*, 40(7):671–694.

[16] Bencomo, N., Götz, S., and Song, H. (2019). Models@run.time: a guided tour of the state of the art and research challenges. *Softw. Syst. Model.*, 18(5):3049–3082.

[17] Berardinelli, L., Langer, P., and Mayerhofer, T. (2013). Combining fUML and profiles for non-functional analysis based on model execution traces. In Kruchten, P., Koziolek, A., and Nord, R. L., editors, *Proceedings of the 9th international ACM SIGSOFT conference on Quality of Software Architectures, QoSA 2013, part of Comparch '13 Federated Events on Component-Based Software Engineering and Software Architecture, Vancouver, BC, Canada, June 17-21, 2013*, pages 79–88. ACM.

[18] Berardinelli, L., Marco, A. D., and Pace, S. (2014). fUML-driven design and performance analysis of software agents for wireless sensor network. In Avgeriou, P. and Zdun, U., editors, *Software Architecture - 8th European Conference, ECSA 2014, Vienna, Austria, August 25-29, 2014. Proceedings*, volume 8627 of *Lecture Notes in Computer Science*, pages 324–339. Springer.

[19] Bernardi, S., Domínguez, J. L., Gómez, A., Joubert, C., Merseguer, J., Perez-Palacin, D., Requeno, J. I., and Romeu, A. (2018). A systematic approach for performance assessment using process mining - an industrial experience report. *Empir. Softw. Eng.*, 23(6):3394–3441.

[20] Bernardi, S., Donatelli, S., and Merseguer, J. (2002). From UML sequence diagrams and statecharts to analysable petrinet models. In *Third International Workshop on Software and Performance, WOSP@ISSTA 2002, July 24-26, 2002, Rome, Italy*, pages 35–45. ACM.

[21] Bernardi, S. and Merseguer, J. (2006). QoS assessment via stochastic analysis. *IEEE Internet Comput.*, 10(3):32–42.

[22] Bernardi, S., Merseguer, J., and Petriu, D. C. (2011). A dependability profile within MARTE. *Softw. Syst. Model.*, 10(3):313–336.

[23] Bernardi, S., Merseguer, J., and Petriu, D. C. (2013). *Model-Driven Dependability Assessment of Software Systems*. Springer.

[24] Blair, G. S., Bencomo, N., and France, R. B. (2009). Models@ run.time. *Computer*, 42(10):22–27.

[25] Bondavalli, A., Cin, M. D., Latella, D., Majzik, I., Pataricza, A., and Savoia, G. (2001). Dependability analysis in the early phases of UML-based system design. *Comput. Syst. Sci. Eng.*, 16(5):265–275.

[26] Boronat, A., Carsí, J. A., and Ramos, I. (2005). Automatic support for traceability in a generic model management framework. In Hartman, A. and Kreische, D., editors, *Model Driven Architecture - Foundations and Applications, 1st European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings*, volume 3748 of *Lecture Notes in Computer Science*, pages 316–330. Springer.

[27] Boronat, A., Carsí, J. A., and Ramos, I. (2006). Algebraic specification of a model transformation engine. In Baresi, L. and Heckel, R., editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 262–277. Springer.

[28] Brunelière, H., Burger, E., Cabot, J., and Wimmer, M. (2019). A feature-based survey of model view approaches. *Softw. Syst. Model.*, 18(3):1931–1952.

[29] Brunelière, H., Cabot, J., Dupé, G., and Madiot, F. (2014). MoDisco: A model driven reverse engineering framework. *Inf. Softw. Technol.*, 56(8):1012–1032.

[30] Brunelière, H., de Kerchove, F. M., Daniel, G., and Cabot, J. (2018a). Towards scalable model views on heterogeneous model resources. In Wasowski, A., Paige, R. F., and Haugen, Ø., editors, *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, pages 334–344. ACM.

[31] Brunelière, H., Eramo, R., Gómez, A., Besnard, V., Bruel, J., Gogolla, M., Kästner, A., and Rutle, A. (2018b). Model-driven engineering for design-runtime interaction in complex systems: Scientific challenges and roadmap - report on the mde@derun 2018 workshop. In Mazzara, M., Ober, I., and Salaün, G., editors, *Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*, volume 11176 of *Lecture Notes in Computer Science*, pages 536–543. Springer.

[32] Brunelière, H., Perez, J. G., Wimmer, M., and Cabot, J. (2015). EMF views: A view mechanism for integrating heterogeneous models. In Johannesson, P., Lee, M., Liddle, S. W., Opdahl, A. L., and López, O. P., editors, *Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings*, volume 9381 of *Lecture Notes in Computer Science*, pages 317–325. Springer.

[33] Cardellini, V., Casalicchio, E., Grassi, V., Presti, F. L., and Mirandola, R. (2009). QoS-driven runtime adaptation of service oriented architectures. In van Vliet, H. and Issarny, V., editors, *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 131–140. ACM.

[34] Chen, L. (2018). Microservices: Architecting for continuous delivery and devops. In *IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018*, pages 39–46. IEEE Computer Society.

[35] Chiola, G., Franceschinis, G., Gaeta, R., and Ribaudo, M. (1995). GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets. *Perform. Evaluation*, 24(1-2):47–68.

[36] Cicchetti, A., Di Ruscio, D., Eramo, R., and Pierantonio, A. (2010). JTL: A bidirectional and change propagating transformation language. In Malloy, B. A., Staab, S., and van den Brand, M., editors, *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, volume 6563 of *Lecture Notes in Computer Science*, pages 183–202. Springer.

[37] Cortellessa, V., Eramo, R., and Tucci, M. (2018). Availability-driven architectural change propagation through bidirectional model transformations between UML and petri net models. In *IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018*, pages 125–134. IEEE Computer Society.

[38] Cortellessa, V., Marco, A. D., and Inverardi, P. (2007). Non-functional modeling and validation in model-driven architecture. In *Sixth Working IEEE / IFIP*

*Conference on Software Architecture (WICSA 2007), 6-9 January 2005, Mumbai, Maharashtra, India*, page 25.

[39] Cortellessa, V., Marco, A. D., and Inverardi, P. (2011). *Model-Based Software Performance Analysis*. Springer.

[40] Cortellessa, V., Marco, A. D., and Trubiani, C. (2014). An approach for modeling and detecting software performance antipatterns based on first-order logics. *Softw. Syst. Model.*, 13(1):391–432.

[41] Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. F. (2009). Bidirectional transformations: A cross-discipline perspective. In Paige, R. F., editor, *Theory and Practice of Model Transformations - 2nd International Conference, ICMT@TOOLS 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer.

[42] Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–646.

[43] Derakhshanmanesh, M. and Grieger, M. (2016). Model-integrating microservices: A vision paper. In Zimmermann, W., Alperowitz, L., Brügge, B., Fahsel, J., Herrmann, A., Hoffmann, A., Krall, A., Landes, D., Lichter, H., Riehle, D., Schaefer, I., Scheuermann, C., Schlaefer, A., Schupp, S., Seitz, A., Steffens, A., Stollenwerk, A., and Weißbach, R., editors, *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016*, volume 1559 of *CEUR Workshop Proceedings*, pages 142–147. CEUR-WS.org.

[44] Derler, P., Lee, E. A., and Sangiovanni-Vincentelli, A. L. (2012). Modeling cyber-physical systems. *Proc. IEEE*, 100(1):13–28.

[45] Di Pompeo, D., Tucci, M., Celi, A., and Eramo, R. (2019). A microservice reference case study for design-runtime interaction in MDE. In Bagnato, A., Brunelière, H., Burgueño, L., Eramo, R., and Gómez, A., editors, *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019), Eindhoven, The Netherlands, July 15 - 19, 2019*, volume 2405 of *CEUR Workshop Proceedings*, pages 23–32. CEUR-WS.org.

[46] Drusinsky, D. (2006). *Modeling and verification using UML statecharts - a working guide to reactive system design, runtime monitoring and execution-based model checking*. Elsevier.

[47] Düllmann, T. F. and van Hoorn, A. (2017). Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. In Binder, W., Cortellessa, V., Koziolek, A., Smirni, E., and Poess, M., editors, *Companion Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 171–172. ACM.

[48] Eramo, R., Cortellessa, V., Pierantonio, A., and Tucci, M. (2012). Performance-driven architectural refactoring through bidirectional model transformations. In Grassi, V., Mirandola, R., Buhnova, B., and Vallecillo, A., editors, *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures, QoSA 2012, part of Comparch '12 Federated Events on Component-Based Software Engineering and Software Architecture, Bertinoro, Italy, June 25-28, 2012*, pages 55–60. ACM.

[49] Eramo, R., Pierantonio, A., and Rosa, G. (2015). Managing uncertainty in bidirectional model transformations. In Paige, R. F., Di Ruscio, D., and Völter, M., editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, pages 49–58. ACM.

[50] Eramo, R., Pierantonio, A., and Tucci, M. (2018). Improved traceability for bidirectional model transformations. In Hebig, R. and Berger, T., editors, *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018*, volume 2245 of *CEUR Workshop Proceedings*, pages 306–315. CEUR-WS.org.

[51] Eysholdt, M. and Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way. In Cook, W. R., Clarke, S., and Rinard, M. C., editors, *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 307–309. ACM.

[52] Fabro, M. D. D. and Valduriez, P. (2009). Towards the efficient development of model transformations using model weaving and matching transformations. *Softw. Syst. Model.*, 8(3):305–324.

[53] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. (2007). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17.

[54] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. A. and Bowen, K. A., editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press.

[55] Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. In Gabriel, R. P., Bacon, D. F., Lopes, C. V., and Jr., G. L. S., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 57–76. ACM.

[56] Geraci, A., Katki, F., McMonegal, L., Meyer, B., Lane, J., Wilson, P., Radatz, J., Yee, M., Porteous, H., and Springsteel, F. (1991). *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press.

[57] Gomaa, H. and Albassam, E. (2017). Run-time software architectural models for adaptation, recovery and evolution. In Burgueño, L., Corley, J., Bencomo, N., Clarke, P. J., Collet, P., Famelis, M., Ghosh, S., Gogolla, M., Greenyer, J., Guerra, E., Kokaly, S., Pierantonio, A., Rubin, J., and Di Ruscio, D., editors, *Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017*, volume 2019 of *CEUR Workshop Proceedings*, pages 193–200. CEUR-WS.org.

[58] Gómez, A., Joubert, C., and Merseguer, J. (2016). A tool for assessing performance requirements of data-intensive applications. In *Proceedings of the XXIV National Conference of Concurrency and Distributed Systems (JCDS 2016)*, pages 159–169.

[59] Gómez, A., Smith, C. U., Spellmann, A. C., and Cabot, J. (2018). Enabling performance modeling for the masses: Initial experiences. In Khendek, F. and Gotzhein, R., editors, *System Analysis and Modeling. Languages, Methods, and Tools for Systems Engineering - 10th International Conference, SAM 2018, Copenhagen, Denmark, October 15-16, 2018, Proceedings*, volume 11150 of *Lecture Notes in Computer Science*, pages 105–126. Springer.

[60] Goseva-Popstojanova, K. and Trivedi, K. S. (2000). Stochastic modeling formalisms for dependability, performance and performability. In Haring, G., Lindemann, C., and Reiser, M., editors, *Performance Evaluation: Origins and Directions*, volume 1769 of *Lecture Notes in Computer Science*, pages 403–422. Springer.

[61] Happe, J., Becker, S., Rathfelder, C., Friedrich, H., and Reussner, R. H. (2010). Parametric performance completions for model-driven performance prediction. *Perform. Eval.*, 67(8):694–716.

[62] Heinrich, R. (2020). Architectural runtime models for integrating runtime observations and component-based models. *J. Syst. Softw.*, 169:110722.

[63] Hettel, T., Lawley, M., and Raymond, K. (2008). Model synchronisation: Definitions for round-trip engineering. In Vallecillo, A., Gray, J., and Pierantonio, A., editors, *Theory and Practice of Model Transformations - 1st International Conference, ICMT@TOOLS 2008, Zurich, Switzerland, July 1-2, 2008, Proceedings*, volume 5063 of *Lecture Notes in Computer Science*, pages 31–45. Springer.

[64] Hidaka, S., Hu, Z., Inaba, K., Kato, H., and Nakano, K. (2011). GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. In Alexander, P., Pasareanu, C. S., and Hosking, J. G., editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 480–483. IEEE Computer Society.

[65] Hidaka, S., Tisi, M., Cabot, J., and Hu, Z. (2016). Feature-based classification of bidirectional transformation approaches. *Softw. Syst. Model.*, 15(3):907–928.

[66] Huszerl, G., Majzik, I., Pataricza, A., Kosmidis, K., and Cin, M. D. (2002). Quantitative analysis of UML statechart models of dependable systems. *Comput. J.*, 45(3):260–277.

[67] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39.

[68] Jouault, F. and Kurtev, I. (2005). Transforming models with ATL. In Bruel, J., editor, *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer.

[69] Jouault, F., Vanhooff, B., Brunelière, H., Doux, G., Berbers, Y., and Bézivin, J. (2010). Inter-DSL coordination support by combining megamodeling and model weaving. In Shin, S. Y., Ossowski, S., Schumacher, M., Palakal, M. J., and Hung, C., editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 2011–2018. ACM.

[70] Kalibera, T. and Jones, R. E. (2013). Rigorous benchmarking in reasonable time. In Cheng, P. and Petrank, E., editors, *International Symposium on Memory Management, ISMM 2013, Seattle, WA, USA, June 20, 2013*, pages 63–74. ACM.

[71] Kalibera, T. and Jones, R. E. (2020). Quantifying performance changes with effect size confidence intervals. *CoRR*, abs/2007.10899.

[72] Kemme, B. (2018). Replication for availability and fault tolerance. In Liu, L. and Özsu, M. T., editors, *Encyclopedia of Database Systems, Second Edition*. Springer.

[73] Ko, H., Zan, T., and Hu, Z. (2016). BiGUL: a formally verified core language for putback-based bidirectional programming. In Erwig, M. and Rompf, T., editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 61–72. ACM.

[74] Kolovos, D. S. (2009). Establishing correspondences between models with the Epsilon comparison language. In Paige, R. F., Hartman, A., and Rensink, A., editors, *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*, volume 5562 of *Lecture Notes in Computer Science*, pages 146–157. Springer.

[75] Kolovos, D. S., Paige, R. F., and Polack, F. (2008). The Epsilon transformation language. In Vallecillo, A., Gray, J., and Pierantonio, A., editors, *Theory and Practice of Model Transformations - 1st International Conference, ICMT@TOOLS 2008, Zurich, Switzerland, July 1-2, 2008, Proceedings*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer.

[76] Kolovos, Dimitris and Rose, Louis and Paige, Richard and Garcıa-Domınguez, Antonio (2010). *The EPSILON book*. Structure.

[77] Kushary, D. (2000). Bootstrap methods and their application. *Technometrics*, 42(2):216–217.

[78] Laaber, C., Scheuner, J., and Leitner, P. (2019). Software microbenchmarking in the cloud. how bad is it really? *Empir. Softw. Eng.*, 24(4):2469–2508.

[79] Laaber, C., Würsten, S., Gall, H. C., and Leitner, P. (2020). Dynamically reconfiguring software microbenchmarks: reducing execution time without sacrificing result quality. In Devanbu, P., Cohen, M. B., and Zimmermann, T., editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 989–1001. ACM.

[80] Lauder, M., Anjorin, A., Varró, G., and Schürr, A. (2012). Efficient model synchronization with precedence triple graph grammars. In Ehrig, H., Engels, G., Kreowski, H., and Rozenberg, G., editors, *Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings*, volume 7562 of *Lecture Notes in Computer Science*, pages 401–415. Springer.

[81] Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C. (1984). *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc.

[82] Lecomte, T., Burdy, L., and Leuschel, M. (2012). Formally checking large data sets in the railways. *CoRR*, abs/1210.6815.

[83] Lecomte, T., Déharbe, D., Prun, É., and Mottin, E. (2020). Applying a formal method in industry: a 25-year trajectory. *CoRR*, abs/2005.07190.

[84] Lehmann, G., Blumendorf, M., Trollmann, F., and Albayrak, S. (2010). Meta-modeling runtime models. In Dingel, J. and Solberg, A., editors, *Models in Software Engineering - Workshops and Symposia at MODELS 2010, Oslo, Norway, October 2-8, 2010, Reports and Revised Selected Papers*, volume 6627 of *Lecture Notes in Computer Science*, pages 209–223. Springer.

[85] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562.

[86] Macedo, N. and Cunha, A. (2013). Implementing QVT-R bidirectional model transformations using alloy. In Cortellessa, V. and Varró, D., editors, *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7793 of *Lecture Notes in Computer Science*, pages 297–311. Springer.

[87] Marsan, M. A., Balbo, G., Conte, G., Donatelli, S., and Franceschinis, G. (1998). Modelling with generalized stochastic Petri nets. *SIGMETRICS Perform. Evaluation Rev.*, 26(2):2.

[88] Marsan, M. A., Conte, G., and Balbo, G. (1984). A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.*, 2(2):93–122.

[89] Martens, A., Koziolek, H., Becker, S., and Reussner, R. H. (2010). Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In Adamson, A., Bondi, A. B., Juiz, C., and Squillante, M. S., editors, *Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering, San Jose, California, USA, January 28-30, 2010*, pages 105–116. ACM.

[90] Mazkatli, M., Monschein, D., Grohmann, J., and Koziolek, A. (2020). Incremental calibration of architectural performance models with parametric dependencies. In *2020 IEEE International Conference on Software Architecture, ICSA 2020, Salvador, Brazil, March 16-20, 2020*, pages 23–34. IEEE.

[91] Mitchell, N., Sevitsky, G., and Srinivasan, H. (2006). Modeling runtime behavior in framework-based applications. In Thomas, D., editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 429–451. Springer.

[92] Mkaouer, M. W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., and Ouni, A. (2020). Many-objective software remodularization using NSGA-III. *CoRR*, abs/2005.06510.

[93] Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.

[94] Mustafiz, S., Sun, X., Kienzle, J., and Vangheluwe, H. (2008). Model-driven assessment of system dependability. *Softw. Syst. Model.*, 7(4):487–502.

[95] Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc., 1st edition.

[96] Object Management Group (2008). A UML profile for MARTE: modeling and analysis of real-time embedded systems. Object Management Group.

[97] Object Management Group (2015). Unified modeling language. Object Management Group. Version 2.5.

[98] Otero, M. C. and Dolado, J. J. (2004). Evaluation of the comprehension of the dynamic modeling in UML. *Inf. Softw. Technol.*, 46(1):35–53.

[99] Pacheco, H., Zan, T., and Hu, Z. (2014). BiFluX: A bidirectional functional update language for XML. In Chitil, O., King, A., and Danvy, O., editors, *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pages 147–158. ACM.

[100] Paige, R. F., Drivalos, N., Kolovos, D. S., Fernandes, K. J., Power, C., Olsen, G. K., and Zschaler, S. (2011). Rigorous identification and encoding of trace-links in model-driven engineering. *Softw. Syst. Model.*, 10(4):469–487.

[101] Paige, R. F., Kolovos, D. S., Rose, L. M., Drivalos, N., and Polack, F. A. C. (2009). The design of a conceptual framework and technical infrastructure for model management language engineering. In *14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009, Potsdam, Germany, 2-4 June 2009*, pages 162–171. IEEE Computer Society.

[102] Popova-Zeugmann, L. (2013). *Time and Petri Nets*. Springer.

[103] Porter, J., Menascé, D. A., and Gomaa, H. (2016). DeSARM: A decentralized mechanism for discovering software architecture models at runtime in distributed systems. In Götz, S., Bencomo, N., Bellman, K. L., and Blair, G. S., editors, *Proceedings of the 11th International Workshop on Models@run.time co-located with 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint Malo, France, October 4, 2016*, volume 1742 of *CEUR Workshop Proceedings*, pages 43–51. CEUR-WS.org.

[104] Powell, D., Bey, I., and Leuridan, J., editors (1991). *Delta Four: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, Berlin, Heidelberg.

[105] Rademacher, F., Sachweh, S., and Zündorf, A. (2017). Differences between model-driven development of service-oriented and microservice architecture. In *2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017*, pages 38–45. IEEE Computer Society.

[106] Reiser, M. and Lavenberg, S. S. (1981). Corrigendum: "mean-value analysis of closed multichain queuing networks". *J. ACM*, 28(3):629.

[107] Ren, S., Lai, H., Tong, W., Aminzadeh, M., Hou, X., and Lai, S. (2010). Nonparametric bootstrapping for hierarchical data. *Journal of Applied Statistics*, 37(9):1487–1498.

[108] Sampaio, A. R., Rubin, J., Beschastnikh, I., and Rosa, N. S. (2019). Improving microservice-based applications with runtime placement adaptation. *J. Internet Serv. Appl.*, 10(1):4:1–4:30.

[109] Saridakis, T. (2002). A system of patterns for fault tolerance. In O'Callaghan, A., Eckstein, J., and Schwanninger, C., editors, *Proceedings of the 7th European Conference on Pattern Languages of Programms (EuroPLoP '2002), Irsee, Germany, July 3-7, 2002*, pages 535–582. UVK - Universitaetsverlag Konstanz.

[110] Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31.

[111] Schürr, A. (1994). Specification of graph translators with triple graph grammars. In Mayr, E. W., Schmidt, G., and Tinhofer, G., editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer.

[112] Smith, C. U. (2007). Introduction to software performance engineering: Origins and outstanding problems. In *Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures*, pages 395–428.

[113] Smith, C. U. and Williams, L. G. (2002). New software performance antipatterns: More ways to shoot yourself in the foot. In *28th International Computer Measurement Group Conference, December 8-13, 2002, Reno, Nevada, USA, Proceedings*, pages 667–674. Computer Measurement Group.

[114] Stevens, P. (2007a). Bidirectional model transformations in QVT: semantic issues and open questions. In Engels, G., Opdyke, B., Schmidt, D. C., and Weil, F., editors, *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer.

[115] Stevens, P. (2007b). A landscape of bidirectional model transformations. In Lämmel, R., Visser, J., and Saraiva, J., editors, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer.

[116] Stevens, P. (2014). Bidirectionally tolerating inconsistency: Partial transformations. In Gnesi, S. and Rensink, A., editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8411 of *Lecture Notes in Computer Science*, pages 32–46. Springer.

[117] Tatibouet, J., Cuccuru, A., Gérard, S., and Terrier, F. (2013). Principles for the realization of an open simulation framework based on fUML (WIP). In Wainer, G. A., Mosterman, P. J., Barros, F. J., and Zacharewicz, G., editors, *2013 Spring Simulation Multiconference, SpringSim '13, San Diego, CA, USA, April 7-10, 2013, Proceedings of the Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, page 4. ACM.

[118] Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2010). *Software Architecture - Foundations, Theory, and Practice*. Wiley.

[119] Toeroe, M. and Tam, F. (2012). *Service Availability: Principles and Practice*. Wiley.

[120] Traini, L., Di Pompeo, D., Tucci, M., Lin, B., Scalabrino, S., Bavota, G., Lanza, M., Oliveto, R., and Cortellessa, V. (2020). How software refactoring impacts execution time. *ACM Transactions on Software Engineering and Methodology*. Submitted.

[121] Trubiani, C., Bran, A., van Hoorn, A., Avritzer, A., and Knoche, H. (2018). Exploiting load testing and profiling for performance antipattern detection. *Inf. Softw. Technol.*, 95:329–345.

[122] Tsantalis, N. and Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *IEEE Trans. Software Eng.*, 35(3):347–367.

[123] Tsantalis, N., Ketkar, A., and Dig, D. (2020). RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*.

[124] Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In Chaudron, M., Crnkovic, I., Chechik, M., and Harman, M., editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 483–494. ACM.

[125] Vögele, C., van Hoorn, A., Schulz, E., Hasselbring, W., and Krcmar, H. (2018). WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction - a model-driven approach for session-based application systems. *Softw. Syst. Model.*, 17(2):443–477.

[126] Weber, M. and Kindler, E. (2003). The Petri net markup language. In *Petri Net Technology for Communication-Based Systems - Advances in Petri Nets*, pages 124–144.

[127] Westfechtel, B. (2015). A case study for evaluating bidirectional transformations in QVT relations. In Filipe, J. and Maciaszek, L. A., editors, *ENASE 2015 - Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering, Barcelona, Spain, 29-30 April, 2015*, pages 141–155. SciTePress.

[128] Weyns, D. (2019). Software engineering of self-adaptive systems. In Cha, S., Taylor, R. N., and Kang, K. C., editors, *Handbook of Software Engineering*, pages 399–443. Springer.

[129] Zan, T., Pacheco, H., and Hu, Z. (2014). Writing bidirectional model transformations as intentional updates. In Jalote, P., Briand, L. C., and van der Hoek, A., editors, *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, pages 488–491. ACM.

[130] Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Liu, D., Xiang, Q., and He, C. (2019). Latent error prediction and fault localization for microservice applications by learning from system trace logs. In Dumas, M., Pfahl, D., Apel, S., and Russo, A., editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 683–694. ACM.

[131] Zhou, X., Peng, X., Xie, T., Sun, J., Li, W., Ji, C., and Ding, D. (2018a). Delta debugging microservice systems. In Huchard, M., Kästner, C., and Fraser, G., editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 802–807. ACM.

[132] Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., and Zhao, W. (2018b). Benchmarking microservice systems for software engineering research. In Chaudron, M., Crnkovic, I., Chechik, M., and Harman, M., editors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 323–324. ACM.

[133] Zúñiga-Prieto, M., Insfran, E., Abrahão, S., and Cano-Genoves, C. (2017). Automation of the incremental integration of microservices architectures. In Goluchowski, J., Pankowska, M., Linger, H., Barry, C., Lang, M., and Schneider, C., editors, *Complexity in Information Systems Development*, pages 51–68, Cham. Springer International Publishing.

# Appendix A

# Model transformations

This appendix contains the full code of transformations that are not otherwise included in the rest of the thesis.

## A.1 Hierarchical State Machines to State Machines

JTL transformation for the Hierarchical State Machines to State Machines benchmark presented in Chapter 3.

```
1 transformation hsm2sm(hsm : HSM, sm : SM) {
2
3   top relation StateMachine2StateMachine {
4
5     enforce domain hsm hm : HSM::StateMachine {
6     };
7
8     enforce domain sm m : SM::StateMachine {
9     };
10
11    where {
12      ownedState2ownedState(hm,m);
13      ownedCompositeState2ownedState(hm,m);
14      ownedInitialState2ownedInitialState(hm,m);
15      ownedFinalState2ownedFinalState(hm,m);
16      ownedTransition2ownedTransition(hm,m);
17    }
18  }
19
20  relation ownedState2ownedState {
21
```

```
22      enforce domain hsm hm : HSM::StateMachine {
23        ownedState = s : HSM::State { }
24      };
25
26      enforce domain sm m : SM::StateMachine {
27        ownedState = s : SM::State { }
28      };
29      where {
30        State2State(s,s);
31      }
32    }
33
34    relation ownedCompositeState2ownedState {
35
36      enforce domain hsm hm : HSM::StateMachine {
37        ownedState = s : HSM::CompositeState { }
38      };
39
40      enforce domain sm m : SM::StateMachine {
41        ownedState = s : SM::State { }
42      };
43
44      where {
45        CompositeState2State(s,s);
46      }
47    }
48
49    relation State2State {
50
51      varName : String;
52
53      enforce domain hsm hs : HSM::State {
54        name = varName
55      };
56
57      enforce domain sm s : SM::State {
58        name = varName
59      };
60
61    }
62
63    relation CompositeState2State {
64
65      varName : String;
```

```
66
67      enforce domain hsm hs : HSM::CompositeState {
68          name = varName
69      };
70
71      enforce domain sm s : SM::State {
72          name = varName
73      };
74    }
75
76    relation ownedInitialState2ownedInitialState {
77
78      enforce domain hsm hm : HSM::StateMachine {
79        ownedState = s : HSM::InitialState { }
80      };
81
82      enforce domain sm m : SM::StateMachine {
83        ownedState = s : SM::InitialState { }
84      };
85
86      where {
87        InitialState2InitialState(s,s);
88      }
89    }
90
91    relation ownedFinalState2ownedFinalState {
92
93      enforce domain hsm hm : HSM::StateMachine {
94        ownedState = s : HSM::FinalState { }
95      };
96
97      enforce domain sm m : SM::StateMachine {
98        ownedState = s : SM::FinalState { }
99      };
100
101     where {
102       FinalState2FinalState(s,s);
103     }
104   }
105
106   relation InitialState2InitialState {
107
108     varName : String;
109
```

```
110     enforce domain hsm hs : HSM::InitialState {
111       name = varName
112     };
113
114     enforce domain sm s : SM::InitialState {
115       name = varName
116     };
117
118   }
119
120   relation FinalState2FinalState {
121
122     varName : String;
123
124     enforce domain hsm hs : HSM::FinalState {
125       name = varName
126     };
127
128     enforce domain sm s : SM::FinalState {
129       name = varName
130     };
131   }
132
133   relation ownedTransition2ownedTransition {
134
135     enforce domain hsm hm : HSM::StateMachine {
136       ownedTransition = t : HSM::Transition { }
137     };
138
139     enforce domain sm m : SM::StateMachine {
140       ownedTransition = t : SM::Transition { }
141     };
142
143     where {
144       Transition2Transition(t,t);
145     }
146   }
147
148   relation Transition2Transition {
149
150     varTrigger : String;
151     varEffect : String;
152
153     enforce domain hsm ht : HSM::Transition {
```

```
154       trigger = varTrigger,
155       effect = varEffect
156     };
157
158     enforce domain sm t : SM::Transition {
159       trigger = varTrigger,
160       effect = varEffect
161     };
162
163     where {
164       TransitionSource2TransitionSource(ht, t);
165       TransitionTarget2TransitionTarget(ht, t);
166       TransitionInitial2TransitionInitial(ht, t);
167       TransitionFinal2TransitionFinal(ht, t);
168       TransitionSourceComposite2TransitionSource(ht, t);
169       TransitionTargetComposite2TransitionTarget(ht, t);
170       TransitionIntoComposite2TransitionTargetComposite(ht, t);
171       TransitionIntoComposite2TransitionSourceComposite(ht, t);
172     }
173
174   }
175
176   relation TransitionSource2TransitionSource {
177
178     enforce domain hsm ht : HSM::Transition {
179       source = s : HSM::State {
180           }
181     };
182
183     enforce domain sm t : SM::Transition {
184       source = s : SM::State {
185       }
186     };
187     when {
188       State2State(s, s);
189     }
190   }
191
192   relation TransitionTarget2TransitionTarget {
193
194     enforce domain hsm ht : HSM::Transition {
195        target = hs : HSM::State {
196       }
197     };
```

```
198
199     enforce domain sm t : SM::Transition {
200       target = s : SM::State {
201       }
202     };
203     when {
204       State2State(hs, s);
205     }
206   }
207
208   relation TransitionInitial2TransitionInitial {
209
210     enforce domain hsm ht : HSM::Transition {
211       source = hs : HSM::InitialState {
212       }
213     };
214
215     enforce domain sm t : SM::Transition {
216       source = s : SM::InitialState {
217       }
218     };
219     when {
220       InitialState2InitialState(hs, s);
221     }
222   }
223
224   relation TransitionFinal2TransitionFinal {
225
226     enforce domain hsm ht : HSM::Transition {
227       target = hs : HSM::FinalState {
228       }
229     };
230
231     enforce domain sm t : SM::Transition {
232       target = s : SM::FinalState {
233       }
234     };
235     when {
236       FinalState2FinalState(hs, s);
237     }
238   }
239
240   relation TransitionSourceComposite2TransitionSource {
241
```

```
242     enforce domain hsm ht : HSM::Transition {
243       source = hcs : HSM::CompositeState {
244       }
245     };
246
247     enforce domain sm t : SM::Transition {
248       source = s : SM::State {
249       }
250     };
251
252     when {
253       CompositeState2State(hcs, s);
254     }
255   }
256
257   relation TransitionTargetComposite2TransitionTarget {
258
259     enforce domain hsm ht : HSM::Transition {
260       target = hcs : HSM::CompositeState {
261       }
262     };
263
264     enforce domain sm t : SM::Transition {
265       target = s : SM::State {
266       }
267     };
268
269     when {
270       CompositeState2State(hcs, s);
271     }
272   }
273
274   relation TransitionIntoComposite2TransitionTargetComposite {
275
276     enforce domain hsm ht : HSM::Transition {
277       target = hs : HSM::State {
278         owningCompositeState = s : HSM::CompositeState {
279         }
280       }
281     };
282
283     enforce domain sm t: SM::Transition {
284       target = s : SM::State {
285       }
```

```
286     };
287
288     when {
289       CompositeState2State(s,s);
290     }
291   }
292
293   relation TransitionIntoComposite2TransitionSourceComposite {
294
295     enforce domain hsm ht : HSM::Transition {
296       source = hs : HSM::State {
297         owningCompositeState = s : HSM::CompositeState {
298         }
299       }
300     };
301
302     enforce domain sm t: SM::Transition {
303       source = s : SM::State {
304       }
305     };
306
307     when {
308       CompositeState2State(s,s);
309     }
310   }
311 }
```

Listing A.1 The HSM2SM transformation.

## A.2 Families to Persons

JTL transformation for the Families to Persons benchmark presented in Chapter 3.

```
1 transformation Families2Persons(family:Families, person:Persons)
      {
2
3   top relation FamilyRegister2PersonRegister {
4
5     enforce domain family fr:Families::FamilyRegister {
6       };
7
8       enforce domain person pr:Persons::PersonRegister{
9       };
```

```
10
11      where {
12        Father2Male(fr,pr);
13        Mother2Female(fr,pr);
14        Son2Male(fr,pr);
15        Daughter2Female(fr,pr);
16      }
17  }
18
19  relation FamilyMember2Male {
20    enforce domain family m:Families::FamilyMember{
21      name = n
22      };
23
24      enforce domain person m:Persons::Male {
25        name = n
26      };
27  }
28
29  relation FamilyMember2Female {
30    enforce domain family m:Families::FamilyMember{
31      name = n
32      };
33
34      enforce domain person m:Persons::Female {
35        name = n
36      };
37  }
38
39  relation Father2Male {
40    n: String;
41    sn: String;
42    enforce domain family fr:Families::FamilyRegister {
43      families = f:Families::Family {
44        name = sn,
45        father = m:Families::FamilyMember {
46          name = n
47        }
48      }
49      };
50
51      enforce domain person pr:Persons::PersonRegister {
52        persons = m:Persons::Male {
53          name = n,
```

```
54            surname = sn
55          }
56        };
57
58        where {
59          FamilyMember2Male(m,m);
60        }
61    }
62
63    relation Mother2Female {
64      n: String;
65      sn: String;
66      enforce domain family fr:Families::FamilyRegister {
67        families = f:Families::Family {
68          name = sn,
69          mother = m:Families::FamilyMember {
70            name = n
71          }
72        }
73        };
74
75        enforce domain person pr:Persons::PersonRegister {
76          persons = m:Persons::Female {
77            name = n,
78            surname = sn
79          }
80        };
81
82        where {
83          FamilyMember2Female(m,m);
84        }
85    }
86
87    relation Son2Male {
88      n: String;
89      sn: String;
90      enforce domain family fr:Families::FamilyRegister {
91        families = f:Families::Family {
92          name = sn,
93          sons = m:Families::FamilyMember{
94            name = n
95          }
96        }
97        };
```

```
98
99        enforce domain person pr:Persons::PersonRegister {
100         persons = m:Persons::Male {
101           name = n,
102           surname = sn
103         }
104       };
105
106       where {
107         FamilyMember2Male(m,m);
108       }
109   }
110
111   relation Daughter2Female {
112     n: String;
113     sn: String;
114     enforce domain family fr:Families::FamilyRegister {
115       families = f:Families::Family {
116         name = sn,
117         daughters = m:Families::FamilyMember {
118           name = n
119         }
120       }
121       };
122
123       enforce domain person pr:Persons::PersonRegister{
124         persons = m:Persons::Female {
125           name = n,
126           surname = sn
127         }
128       };
129
130       where {
131         FamilyMember2Female(m,m);
132       }
133   }
134 }
```

Listing A.2 The Families2Persons transformation.

## A.3 UML to Generalized Stochastic Petri Nets

JTL transformation presented in Chapter 4, from UML annotated with MARTE-DAM to Generalized Stochastic Petri Nets.

```
transformation UMLGSPN (uml:umlsm, pn:ptnet) {

  -- State Machines

  top relation StateMachine2PetriNet {

    name: String;

    enforce domain uml statemachine:umlsm::StateMachine {
      name = name,
      region = r : umlsm::Region {}
    };

    enforce domain pn petrinet : ptnet::PetriNet {
      id = name,
      pages = p : ptnet::Page {}
    };

    where {
      State2Pattern(r, p);
      StateActivity2Pattern(r, p);
      Transition2Pattern(r, p);
    }
  }

  relation State2Pattern {

    enforce domain uml r : umlsm::Region {
      subvertex = s : umlsm::State {}
    };

    enforce domain pn p : ptnet::Page {
      objects = s : ptnet::Place {}
    };
    enforce domain pn p : ptnet::Page {
      objects = s1 : ptnet::Transition {
        transitionKind = "immediate"
      }
    };
```

```
40      enforce domain pn p : ptnet::Page {
41        objects = s2 : ptnet::Arc {
42          source = s : ptnet::Place {}
43          target = s1 : ptnet::Transition {}
44        }
45      };
46
47      where {
48        s.doActivity.oclIsUndefined()
49      }
50    }
51
52    relation StateActivity2Pattern {
53
54      enforce domain uml r : umlsm::Region {
55        subvertex = s : umlsm::State {
56          doActivity = a : umlsm::Activity {}
57        }
58      };
59
60      enforce domain pn p : ptnet::Page {
61        objects = s : ptnet::Place {}
62      };
63      enforce domain pn p : ptnet::Page {
64        objects = s1 : ptnet::Transition {
65          transitionKind = "immediate"
66        }
67      };
68      enforce domain pn p : ptnet::Page {
69        objects = s2 : ptnet::Arc {
70          source = s : ptnet::Place {}
71          target = s1 : ptnet::Transition {}
72        }
73      };
74      enforce domain pn p : ptnet::Page {
75        objects = s3 : ptnet::Place {}
76      };
77      enforce domain pn p : ptnet::Page {
78        objects = s4 : ptnet::Arc {
79          source = s1 : ptnet::Transition {}
80          target = s3 : ptnet::Place {}
81        }
82      };
83      enforce domain pn p : ptnet::Page {
```

```
84        objects = s5 : ptnet::Transition {
85          transitionKind = "exponential"
86        }
87      };
88      enforce domain pn p : ptnet::Page {
89        objects = s6 : ptnet::Arc {
90          source = s3 : ptnet::Place {}
91          target = s5 : ptnet::Transition {}
92        }
93      };
94    }
95
96    -- Transition leaving a state WITHOUT activity
97    relation Transition2Pattern {
98
99      enforce domain uml region : umlsm::Region {
100       transition = t : umlsm::Transition {
101         source = s : umlsm::State {}
102       }
103     };
104
105     enforce domain pn page : ptnet::Page {
106       objects = t : ptnet::Place {}
107     };
108     enforce domain pn page : ptnet::Page {
109       objects = t1 : ptnet::Transition {
110         transitionKind = "immediate"
111       }
112     };
113     enforce domain pn page : ptnet::Page {
114       objects = t2 : ptnet::Arc {
115         source = t : ptnet::Place {}
116         target = t1 : ptnet::Transition {}
117       }
118     };
119     enforce domain pn page : ptnet::Page {
120       objects = t3 : ptnet::Arc {
121         source = t : ptnet::Place {}
122         target = s1 : ptnet::Transition {}
123       }
124     };
125     enforce domain pn page : ptnet::Page {
126       objects = t4 : ptnet::Place {}
127     };
```

```
128     enforce domain pn page : ptnet::Page {
129       objects = t5 : ptnet::Arc {
130         source = s1 : ptnet::Transition {}
131         target = t4 : ptnet::Place {}
132       }
133     };
134     enforce domain pn page : ptnet::Page {
135       objects = t6 : ptnet::Transition {
136         transitionKind = "immediate"
137       }
138     };
139     enforce domain pn page : ptnet::Page {
140       objects = t7 : ptnet::Arc {
141         source = t4 : ptnet::Place {}
142         target = t6 : ptnet::Transition {}
143       }
144     };
145
146     when {
147       State2Pattern(s, s1);
148     }
149   }
150
151   -- Transition leaving a state WITH activity
152   relation TransitionActivity2Pattern {
153
154     enforce domain uml region : umlsm::Region {
155       transition = t : umlsm::Transition {
156         source = s : umlsm::State {}
157       }
158     };
159
160     enforce domain pn page : ptnet::Page {
161       objects = t : ptnet::Place {}
162     };
163     enforce domain pn page : ptnet::Page {
164       objects = t1 : ptnet::Transition {
165         transitionKind = "immediate"
166       }
167     };
168     enforce domain pn page : ptnet::Page {
169       objects = t2 : ptnet::Arc {
170         source = t : ptnet::Place {}
171         target = t1 : ptnet::Transition {}
```

```
172       }
173     };
174     enforce domain pn page : ptnet::Page {
175       objects = t3 : ptnet::Arc {
176         source = t : ptnet::Place {}
177         target = s1 : ptnet::Transition {}
178       }
179     };
180     enforce domain pn page : ptnet::Page {
181       objects = t4 : ptnet::Place {}
182     };
183     enforce domain pn page : ptnet::Page {
184       objects = t5 : ptnet::Arc {
185         source = s1 : ptnet::Transition {}
186         target = t4 : ptnet::Place {}
187       }
188     };
189     enforce domain pn page : ptnet::Page {
190       objects = t6 : ptnet::Transition {
191         transitionKind = "immediate"
192       }
193     };
194     enforce domain pn page : ptnet::Page {
195       objects = t7 : ptnet::Arc {
196         source = t4 : ptnet::Place {}
197         target = t6 : ptnet::Transition {}
198       }
199     };
200
201     when {
202       StateActivity2Pattern(s, s1);
203     }
204   }
205
206   -- DaStep (Transition) leaving a state WITHOUT activity
207   relation TransitionDaStep2Pattern {
208
209     name : String;
210     prob : String;
211
212     enforce domain uml region : umlsm::Region {
213       transition = t : umlsm::DaStep {
214         name = name,
215         occurrenceProb = prob,
```

```
216         source = s : umlsm::State {}
217       }
218     };
219
220     enforce domain pn page : ptnet::Page {
221       objects = t : ptnet::Place {}
222     };
223     enforce domain pn page : ptnet::Page {
224       objects = t1 : ptnet::Transition {
225         id = name,
226         weight = prob
227         transitionKind = "immediate"
228       }
229     };
230     enforce domain pn page : ptnet::Page {
231       objects = t2 : ptnet::Arc {
232         source = t : ptnet::Place {}
233         target = t1 : ptnet::Transition {}
234       }
235     };
236     enforce domain pn page : ptnet::Page {
237       objects = t3 : ptnet::Arc {
238         source = t : ptnet::Place {}
239         target = s1 : ptnet::Transition {}
240       }
241     };
242     enforce domain pn page : ptnet::Page {
243       objects = t4 : ptnet::Place {}
244     };
245     enforce domain pn page : ptnet::Page {
246       objects = t5 : ptnet::Arc {
247         source = s1 : ptnet::Transition {}
248         target = t4 : ptnet::Place {}
249       }
250     };
251     enforce domain pn page : ptnet::Page {
252       objects = t6 : ptnet::Transition {
253         transitionKind = "immediate"
254       }
255     };
256     enforce domain pn page : ptnet::Page {
257       objects = t7 : ptnet::Arc {
258         source = t4 : ptnet::Place {}
259         target = t6 : ptnet::Transition {}
```

```
260       }
261     };
262
263     when {
264       State2Pattern(s, s1);
265     }
266   }
267
268   -- DaStep (Transition) leaving a state WITH activity
269   relation TransitionDaStepActivity2Pattern {
270
271     enforce domain uml region : umlsm::Region {
272       transition = t : umlsm::DaStep {
273         source = s : umlsm::State {}
274       }
275     };
276
277     enforce domain pn page : ptnet::Page {
278       objects = t : ptnet::Place {}
279     };
280     enforce domain pn page : ptnet::Page {
281       objects = t1 : ptnet::Transition {
282         transitionKind = "immediate"
283       }
284     };
285     enforce domain pn page : ptnet::Page {
286       objects = t2 : ptnet::Arc {
287         source = t : ptnet::Place {}
288         target = t1 : ptnet::Transition {}
289       }
290     };
291     enforce domain pn page : ptnet::Page {
292       objects = t3 : ptnet::Arc {
293         source = t : ptnet::Place {}
294         target = s1 : ptnet::Transition {}
295       }
296     };
297     enforce domain pn page : ptnet::Page {
298       objects = t4 : ptnet::Place {}
299     };
300     enforce domain pn page : ptnet::Page {
301       objects = t5 : ptnet::Arc {
302         source = s1 : ptnet::Transition {}
303         target = t4 : ptnet::Place {}
```

```
304       }
305     };
306     enforce domain pn page : ptnet::Page {
307       objects = t6 : ptnet::Transition {
308         transitionKind = "immediate"
309       }
310     };
311     enforce domain pn page : ptnet::Page {
312       objects = t7 : ptnet::Arc {
313         source = t4 : ptnet::Place {}
314         target = t6 : ptnet::Transition {}
315       }
316     };
317
318     when {
319       StateActivity2Pattern(s, s1);
320     }
321   }
322
323   -- Sequence Diagrams
324
325   relation Lifeline2Page {
326     name : String;
327     enforce domain uml m : umlsm::Lifeline {
328       name = name
329         represents = r : umlsm::Property {
330           type = c : umlsm::Component {
331           name = name
332         }
333       }
334     };
335     checkonly domain pn p : ptnet::Page {
336       id = name
337     };
338     when {
339       Component2Page(c, p);
340     }
341   }
342
343   relation MessageSynch2Pattern {
344     enforce domain uml m : umlsm::Message {
345       messageSort = "synchCall"
346
```

```
347       receiveEvent = re : umlsm::MessageOccurrenceSpecification
              {
348         covered = l1 : umlsm::Lifeline {}
349       }
350
351       sendEvent = se : umlsm::MessageOccurrenceSpecification {
352         covered = l2 : umlsm::Lifeline {}
353       }
354     };
355     enforce domain pn p : ptnet::Page {
356       objects = s : ptnet::Place {}
357     };
358     enforce domain pn p : ptnet::Page {
359       objects = s1 : ptnet::Transition {
360         transitionKind = "immediate"
361       }
362     };
363     enforce domain pn p : ptnet::Page {
364       objects = s2 : ptnet::Arc {
365         source = s : ptnet::Place {}
366         target = s1 : ptnet::Transition {}
367       }
368     };
369     enforce domain pn p : ptnet::Page {
370       objects = s3 : ptnet::Arc {
371         source = c : ptnet::Transition {}
372         target = s : ptnet::Place {}
373       }
374     };
375     enforce domain pn p : ptnet::Page {
376       objects = s4 : ptnet::Arc {
377         source = s1 : ptnet::Transition {}
378         target = r : ptnet::Place {}
379       }
380     };
381     when {
382       State2Pattern(c, c);
383       State2Pattern(r, r);
384       Lifeline2Page(l1, p);
385       Lifeline2Page(l2, p);
386     }
387   }
388
389   relation MessageAsynch2Pattern {
```

```
390     enforce domain uml m : umlsm::Message {
391       messageSort = "asynchCall"
392     };
393     enforce domain pn p : ptnet::Page {
394       objects = s : ptnet::Place {}
395     };
396     enforce domain pn p : ptnet::Page {
397       objects = s1 : ptnet::Transition {
398         transitionKind = "immediate"
399       }
400     };
401     enforce domain pn p : ptnet::Page {
402       objects = s2 : ptnet::Arc {
403         source = s : ptnet::Place {}
404         target = s1 : ptnet::Transition {}
405       }
406     };
407     enforce domain pn p : ptnet::Page {
408       objects = s3 : ptnet::Arc {
409         source = c : ptnet::Transition {}
410         target = s : ptnet::Place {}
411       }
412     };
413     enforce domain pn p : ptnet::Page {
414       objects = s4 : ptnet::Arc {
415         source = s1 : ptnet::Transition {}
416         target = r : ptnet::Place {}
417       }
418     };
419     when {
420       State2Pattern(c, c);
421       State2Pattern(r, r);
422     }
423   }
424
425   relation MessageReply2Pattern {
426     enforce domain uml m : umlsm::Message {
427       messageSort = "reply"
428     };
429     enforce domain pn p : ptnet::Page {
430       objects = s : ptnet::Place {}
431     };
432     enforce domain pn p : ptnet::Page {
433       objects = s1 : ptnet::Arc {
```

```
434         source = r : ptnet::Transition {}
435         target = s : ptnet::Place {}
436       }
437     };
438     enforce domain pn p : ptnet::Page {
439       objects = s2 : ptnet::Arc {
440         source = s1 : ptnet::Transition {}
441         target = c : ptnet::Place {}
442       }
443     };
444     when {
445       State2Pattern(r, r);
446       State2Pattern(c, c);
447     }
448   }
449
450   -- Static component updating
451
452   relation Component2Page {
453     name:String;
454     enforce domain uml c:umlsm::Component {
455       name=name
456     };
457     checkonly domain pn p:ptnet::Page {
458       id=name
459     };
460   }
461
462   relation Interface2Pattern {
463     enforce domain uml i:umlsm::Interface {
464       ownedOperation=o:umlsm::Operation {}
465     };
466     enforce domain uml receiver:umlsm::Component {
467       interfaceRealization=ir:umlsm::InterfaceRealization {
468         supplier=i:umlsm::Interface {},
469         contract=i:umlsm::Interface {}
470       }
471     };
472     enforce domain uml caller:umlsm::Component {
473       packagedElement=d:umlsm::Dependency {
474        client=caller:umlsm::Component {},
475        supplier=i:umlsm::Interface {}
476       }
477     };
```

```
478     enforce domain pn p:ptnet::Page {
479       objects=s:ptnet::Place {}
480     };
481     enforce domain pn p:ptnet::Page {
482       objects=s1:ptnet::Transition {
483        transitionKind="immediate"
484       }
485     };
486     enforce domain pn p:ptnet::Page {
487       objects=s2:ptnet::Arc {
488        source=s:ptnet::Place {}
489        target=s1:ptnet::Transition {}
490       }
491     };
492     enforce domain pn p1:ptnet::Page {
493       objects=s3:ptnet::Arc {
494        source=c:ptnet::Transition {}
495        target=s:ptnet::Place {}
496       }
497     };
498     enforce domain pn p2:ptnet::Page {
499       objects=s4:ptnet::Arc {
500        source=s1:ptnet::Transition {}
501        target=r:ptnet::Place {}
502       }
503     };
504     when {
505       Component2Page(caller, p1);
506       Component2Page(receiver, p3);
507     }
508     where {
509       s3.source.containerPage.id <> s3.target.containerPage.id;
510       s4.source.containerPage.id <> s4.target.containerPage.id;
511     }
512   }
513 }
```

Listing A.3 The UML2GSPN transformation.