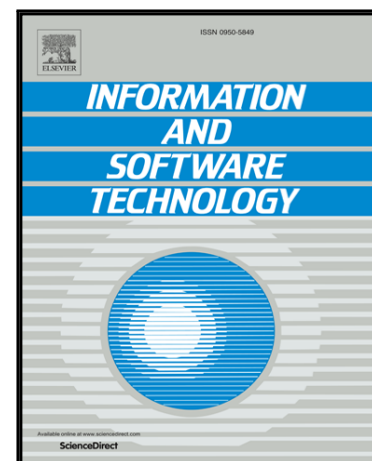


Journal Pre-proof

From Software Architecture to Analysis Models and Back:
Model-Driven Refactoring Aimed at Availability Improvement

Vittorio Cortellessa, Romina Eramo, Michele Tucci

PII: S0950-5849(20)30130-0
DOI: <https://doi.org/10.1016/j.infsof.2020.106362>
Reference: INFOSOF 106362



To appear in: *Information and Software Technology*

Received date: 21 October 2019
Revised date: 24 April 2020
Accepted date: 30 May 2020

Please cite this article as: Vittorio Cortellessa, Romina Eramo, Michele Tucci, From Software Architecture to Analysis Models and Back: Model-Driven Refactoring Aimed at Availability Improvement, *Information and Software Technology* (2020), doi: <https://doi.org/10.1016/j.infsof.2020.106362>

This is a PDF file of an article that has undergone enhancements after acceptance, such as the addition of a cover page and metadata, and formatting for readability, but it is not yet the definitive version of record. This version will undergo additional copyediting, typesetting and review before it is published in its final form, but we are providing this version to give early visibility of the article. Please note that, during the production process, errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

© 2020 Published by Elsevier B.V.

From Software Architecture to Analysis Models and Back: Model-Driven Refactoring Aimed at Availability Improvement

Vittorio Cortellessa, Romina Eramo, Michele Tucci

*Department of Information Engineering, Computer Science and Mathematics
University of L'Aquila, Italy
{vittorio.cortellessa, romina.eramo, michele.tucci}@univaq.it*

Abstract

Context: With the ever-increasing evolution of software systems, their architecture is subject to frequent changes due to multiple reasons, such as new requirements. Appropriate architectural changes driven by non-functional requirements are particularly challenging to identify because they concern quantitative analyses that are usually carried out with specific languages and tools. A considerable number of approaches have been proposed in the last decades to derive non-functional analysis models from architectural ones. However, there is an evident lack of automation in the backward path that brings the analysis results back to the software architecture.

Objective: In this paper, we propose a model-driven approach to support designers in improving the availability of their software systems through refactoring actions.

Method: The proposed framework makes use of bidirectional model transformations to map UML models onto Generalized Stochastic Petri Nets (GSPN) analysis models and vice versa. In particular, after availability analysis, our approach enables the application of model refactoring, possibly based on well-known fault tolerance patterns, aimed at improving the availability of the architectural model.

Results: We validated the effectiveness of our approach on an Environmental Control System. Our results show that the approach can generate: (i) an analyzable availability model from a software architecture description, and (ii) valid software architecture models back from availability models. Finally, our results highlight that the application of fault tolerance patterns significantly improves the availability in each considered

scenario.

Conclusion: The approach integrates bidirectional model transformation and fault tolerance techniques to support the availability-driven refactoring of architectural models. The results of our experiment showed the effectiveness of the approach in improving the software availability of the system.

Keywords: Software architecture, Availability, Bidirectional Model Transformation, Refactoring.

1. Introduction

In order to succeed in new market segments, organizations have constantly been increasing the use of software in systems over the last decades. Nowadays, due to continuous evolution, software architecture is subject to changes induced by decisions taken
 5 along the overall software lifecycle [1]. Indeed, as the earliest artifact that evolves along the process, a software architecture model can support different tasks, such as test case generation [2], traceability [3], and non-functional validation [4].

Appropriate architectural changes driven by non-functional requirements are particularly challenging to identify, mainly because non-functional analysis is based on
 10 specific languages and tools (e.g., Petri Nets, Markov Models) that are different from typical software architecture notations like Architecture Description Languages (e.g., ACME [5]). In fact, very few ADLs embed constructs that enable the specification of non-functional attributes (e.g., AADL [6]) and even fewer ones are equipped with solvers leading non-functional indices out of an architecture specification (e.g., Palladio [7]).
 15 Hence, even in cases where the analysis tools help to identify suitable architectural changes that may overcome non-functional problems, these changes need to be brought back within the architecture description language and environment. This step may prove to be particularly complex, as it subsumes a change of notation that might alter the semantics of identified architectural changes.

20 With the introduction of Model Driven Engineering (MDE) [8] techniques in the software lifecycle, the analysis of quality attributes has become more effective by means of automated transformations from software artifacts to analysis models [9].

Hence, in order to validate non-functional requirements on a software architecture, a number of approaches, mostly based on model transformations, have been proposed in the last decades to generate non-functional models from software architectural descriptions [10, 11]. There is instead a clear lack of automation in the backward path that basically consists in the interpretation of the analysis results and the generation of architectural feedback to be propagated back to the software architecture.

The goal of this paper is to introduce 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 ¹ 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.

The main contributions of this paper are:

- the automated transformation of a software architecture (modeled in UML) into a GSPN analysis model,
- the refactoring of GSPN models (possibly based on well-known fault tolerance patterns), and
- the propagation of the changes performed on the GSPN models back to UML models.

In a previous paper [13], we presented a bidirectional model transformation between UML State Machines (SMs), annotated with availability parameters, and GSPN.

¹The transformation engine is based on JTL [12]

50 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 paper is an extension of our previous paper stemming from the realization that it was restricted to consider only SMs. In fact, a deeper semantic comprehension of an UML model can be achieved if the
 55 dynamic behavior is modeled by using the Sequence Diagrams (SDs) in addition to SMs [14]. In fact, 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
 60 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.

Furthermore, in this paper we introduce a catalog of refactoring patterns with the
 65 aim to drive the designers in their process. In particular, the patterns for fault tolerance presented in [15] 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 implemented as a dedicated framework implemented within Eclipse ².

70 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.

75 The rest of the paper is organized as follows: Section 2 sets the background for this research work along with its contributions and relations with the authors' previous work. Section 3 describes the JASA methodology and its implementation. Section 4 illustrates the application of JASA to the Environmental Control System (ECS) exam-

²Eclipse Platform: <https://projects.eclipse.org/projects/eclipse.platform>

ple application. Section 5 provides the evaluation of the results obtained by applying
 80 JASA. Section 6 describes related approaches, and finally Section 7 concludes the paper.

2. 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
 85 of JASA. Also, we detail the contributions presented in this paper and put them in relation to the authors' previous work.

2.1. 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
 90 decades to generate *non-functional models* from *software architectural descriptions* [10, 11]. This generation step is also called *forward path*, and it is represented by the topmost steps of Fig. 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
 95 (hopefully) ends up when satisfactory indices are produced, and it is represented by the rightmost step of Fig. 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 Fig. 1, also called *backward path*. However, analysis
 100 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 paper, we consider this general round-trip
 105 non-functional analysis process in the availability analysis context.

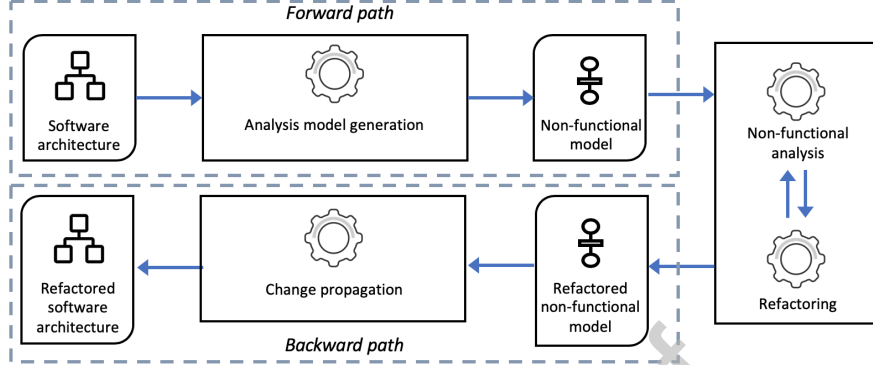


Figure 1: Round-trip non-functional analysis process

2.2. 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 [16]. In particular, the steady state availability can be expressed as the ratio
 110 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 [11]. In this paper, we consider an extension of SPN, called Generalized Stochastic Petri Nets (GSPN) [17]. 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
 115 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,
 120 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 work, availability analysis is conducted on a GSPN derived from a software
 125 architecture modeled in UML [18].

Since UML does not natively provide support for availability modeling, we rely on the “Dependability Analysis and Modeling” (DAM) profile [19] to enhance UML models with availability annotations. DAM was designed on top of the standard MARTE profile [20], 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.

2.3. 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 [15], 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 3.3.

160 3. The JASA 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, 165 JASA leverages the interplay of UML and GSPN and provides automation for their mapping by means of a bidirectional model transformation mechanism [12]. In fact, the bidirectional engine provides the possibility to automate round-trip process by applying the transformation rules in both ways, from right to left domains and vice versa. In addition, JASA provides a set of refactoring actions that can be used by the designer to 170 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 175 online ³.

3.1. Using model driven techniques

Model Driven Engineering (MDE) [8] leverages domain knowledge and business logic from source code into high-level specifications enabling more accurate analyses. In general, an application domain is consistently analyzed and engineered by means of a 180 *metamodel*, i.e., a coherent set of interrelated concepts. A model is said to *conform* to a metamodel: it is expressed by the concepts encoded in the metamodel. Constraints are expressed at the meta-level, and model transformations are based on source and

³JASA: <https://github.com/SEALABQualityGroup/JASA>

target metamodels. With the introduction of model-driven techniques in the software lifecycle, the analysis of quality attributes has become effective by means of automated transformations from software artifacts to analysis models [10].

The proposed approach makes use of bidirectional model transformations [21] to map architectural models and analysis models in both forward and backward directions. In contrast to unidirectional languages, bidirectional approaches allow describing both forward and backward transformations simultaneously, so that the consistency of the transformation can be guaranteed by construction [22].

3.2. From availability assessment to architecture improvements

The proposed approach is realized on top of JTL (Janus Transformation Language) [12, 23], that is a constraint-based model transformation framework specifically tailored to support bidirectionality and change propagation⁴. JASA has been implemented within the Eclipse framework and mainly exploits the Eclipse Modeling Framework (EMF)⁵. As a consequence, the environment supports any language defined as a metamodel conforming to Ecore (i.e., the EMF metamodel). In this work, 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 paper, we use UML^{JASA} to refer to the considered UML diagrams, that are UMLSM, UML^{SD} and UML^{CD}.

The JASA overall approach is reported in Fig. 2. As said, the *Bidirectional engine* relies on JTL to enable the execution of bidirectional model transformations in both forward and backward directions. The UML^{JASA}-GSPN bidirectional transformation maps UML models to GSPN and vice versa. In particular, in order to execute the 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

⁴JTL: <http://jtl.univaq.it/>

⁵Eclipse Modeling Framework: <https://www.eclipse.org/modeling/emf/>

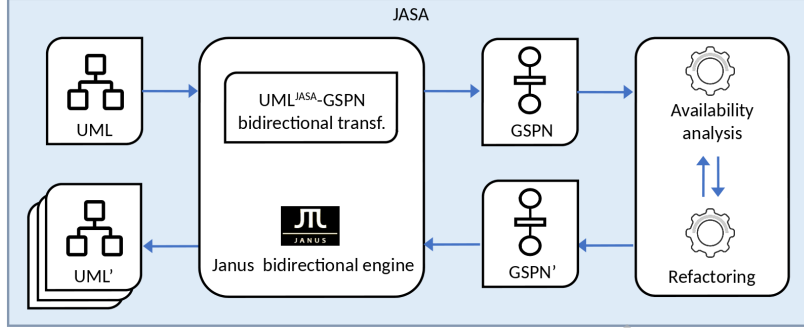


Figure 2: The JASA overall approach

210 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
 215 DAM-annotated UML model representing the software architecture that embeds the changes made on the GSPN to solve arisen availability problems.

3.3. 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
 220 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 automa-
 225 tion. 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.

230 Once applied on the GSPN model, the refactoring patterns will be propagated back-
wards through the transformation that will be presented in Section 3.4.

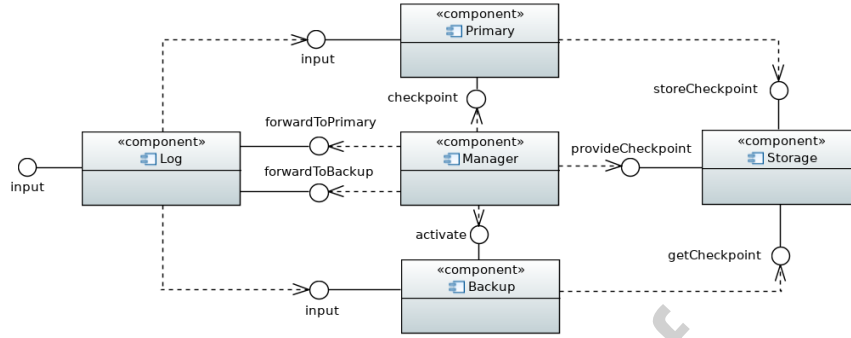
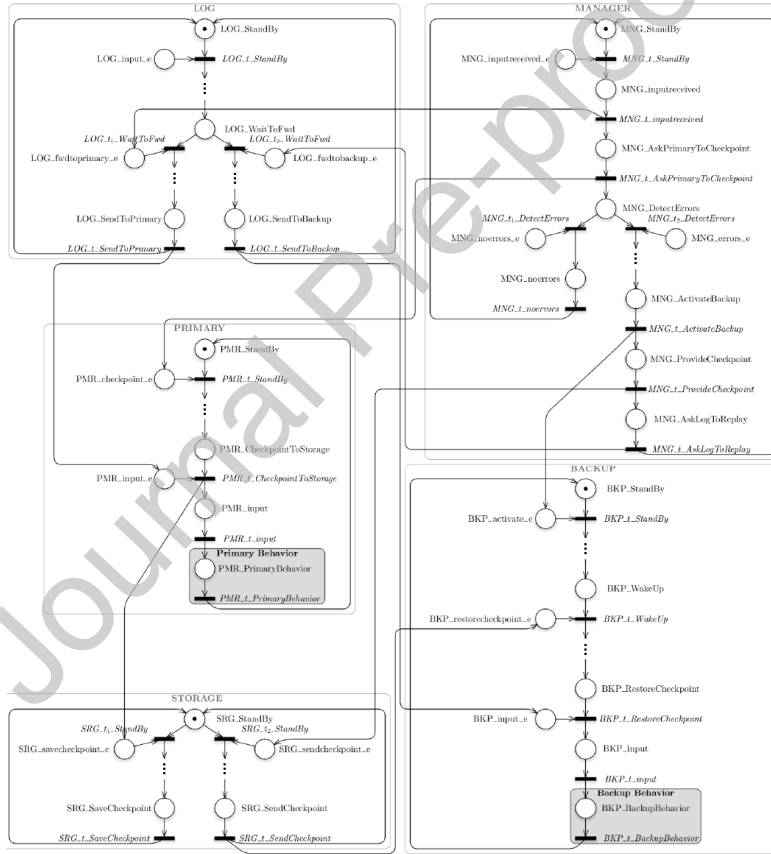
3.3.1. Passive Replication

In this pattern, an error is masked by saving the state of a system component (a check-
point) before it starts processing the input. If an error is detected, a backup replica
235 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 3a,
where components represent roles in the pattern and interfaces are used to depict the
actions that are required for coordination. Figure 3b shows the implementation of this
240 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 sur-
rounding 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.

245 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
250 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,
255 (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.

(a) Component Diagram of the *Passive Replication* pattern(b) GSPN of the *Passive Replication* patternFigure 3: The *Passive Replication* pattern

3.3.2. Semi-Passive Replication

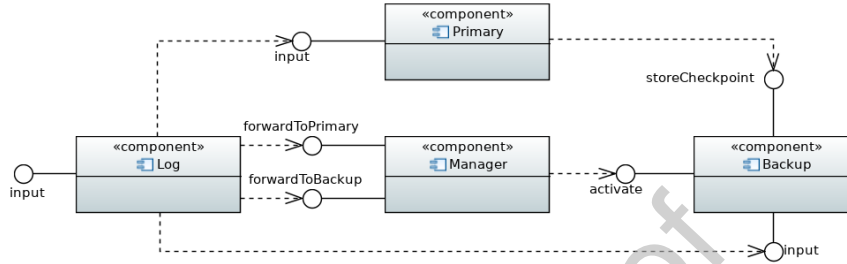
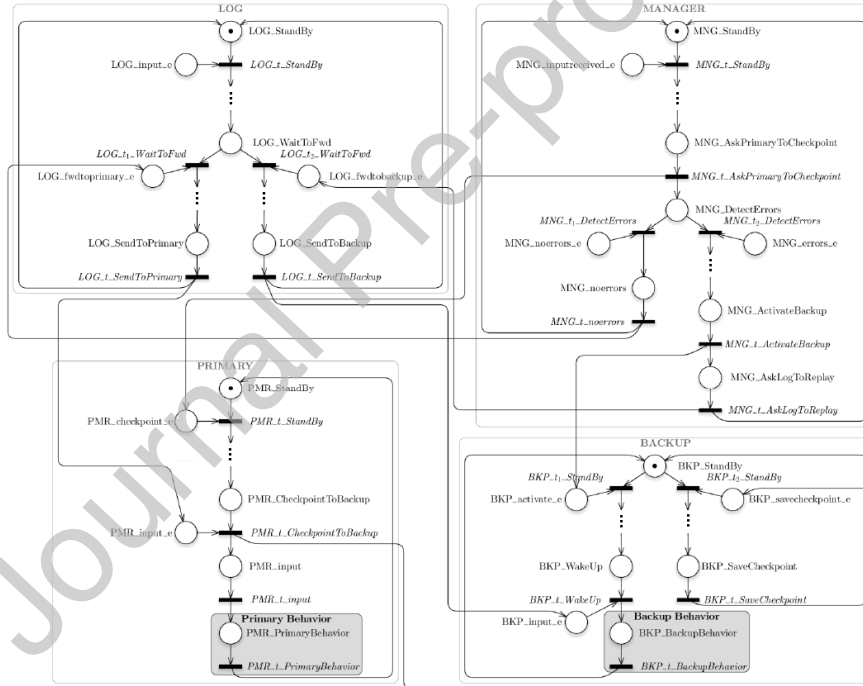
The *Semi-Passive Replication* pattern is able to mask errors in a similar way to the *Passive Replication* pattern, but without requiring a storage dedicated to checkpoints. The Component Diagram of this pattern is shown in Figure 4a. The *Primary* component saves the checkpoint by sending it directly to the *Backup*. *Log* and *Manager* components are still required to implement the pattern. The *Log* stores and forward 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.

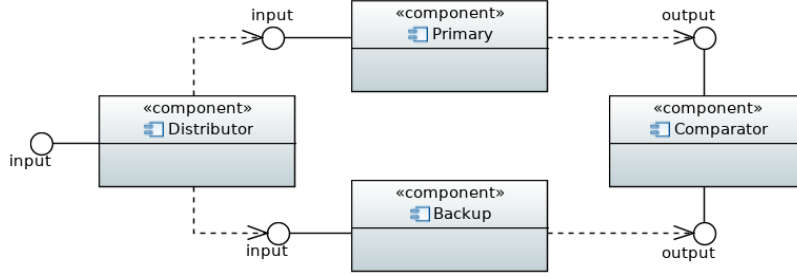
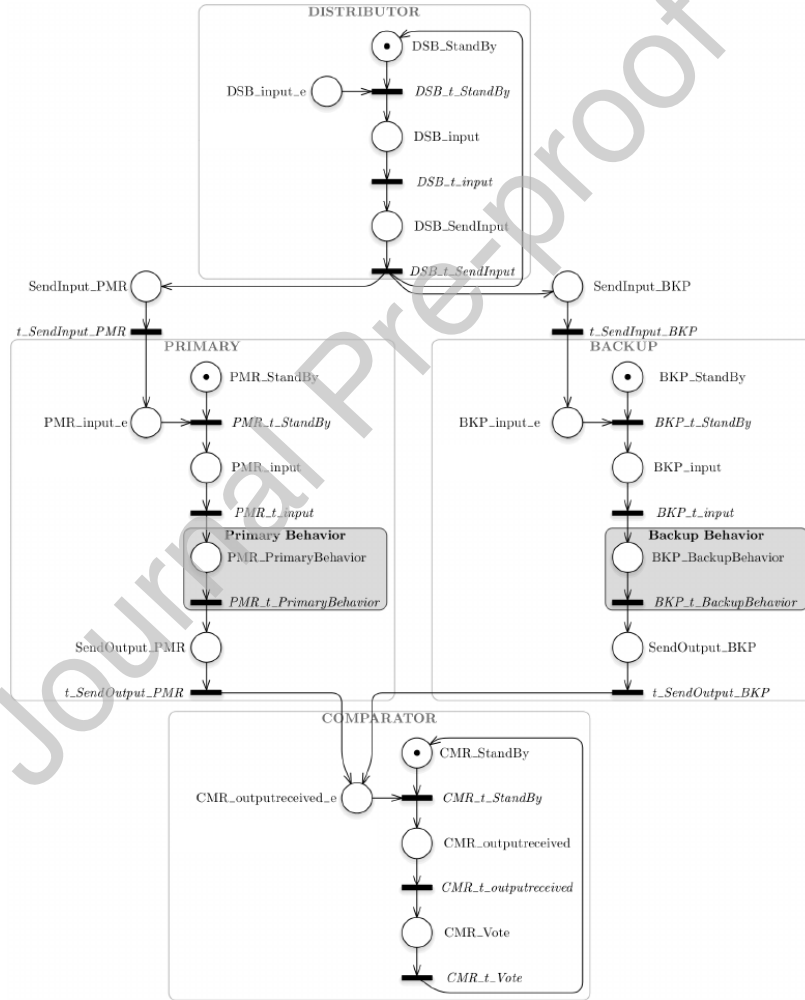
3.3.3. Active Replication

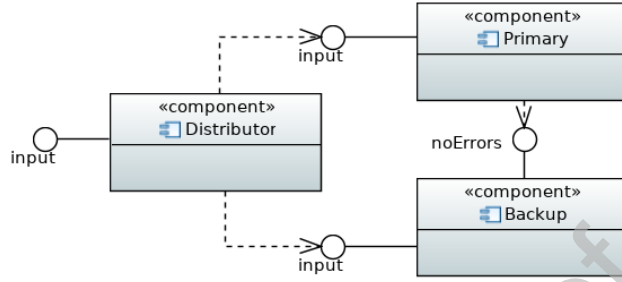
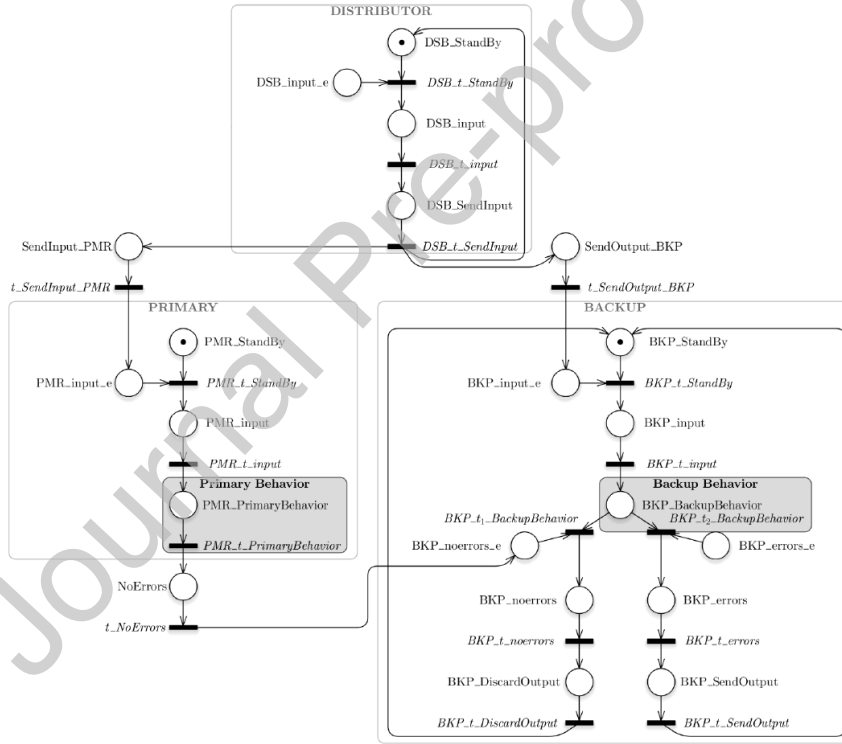
The *Active Replication* pattern is considered the most effective error masking technique but also the most expensive. This pattern employs a group of replicas actively receiving and processing every input intended for the component of which we want to mask errors. According to this pattern, we need 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 (by majority voting) what will be the final output of the system.

Figure 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.

(a) Component Diagram of the *Semi-Passive Replication* pattern(b) GSPN of the *Semi-Passive Replication* patternFigure 4: The *Semi-Passive Replication* pattern

(a) Component Diagram of the *Active Replication* pattern(b) GSPN of the *Active Replication* patternFigure 5: The *Active Replication* pattern

(a) Component Diagram of the *Semi-Active Replication* pattern(b) GSPN of the *Semi-Active Replication* patternFigure 6: The *Semi-Active Replication* pattern

285 3.3.4. *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 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* takes over the responsibility to deliver the output. *Primary*, *Backup* must be deployed to different units of failure.

3.4. The UML^{JASA}-GSPN bidirectional transformation

JTL adopts a textual syntax (that is inspired to QVT-R [24]) and allows a declarative specification of relationships between MOF models. The mechanism of transformation is rule-based. The language supports object pattern matching, and implicitly creates trace instances to record what occurred during a transformation execution. A transformation between candidate models is specified as a set of *relations* that must hold for the transformation to be successful: in particular, it is defined by two *domains* and includes a pair of *when* and *where* predicates that specify the pre- and post- conditions that must be satisfied by elements of the candidate models. When a bidirectional transformation is invoked for the enforcement, it is executed in a specific direction by selecting one of the candidate models as the target by means of a run configuration. The implementation relies on the Answer Set Programming (ASP) [25], which 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.

The JTL engine finds and generates, in a single execution, all possible models that are consistent with the transformation rules by a deductive process.

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

- Mapping UMLSM to GSPN models and vice versa (UMLSM-GSPN),

- 315 • 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 [18]. 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 10, the *getTemperatureData* 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 online ⁶.

3.4.1. 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

⁶ UML^{JASA} -GSPN: <https://github.com/SEALABQualityGroup/JASA/blob/master/JTL/transformations/UMLGSPN.jtl>

in GSPN are generated and vice versa. Such implementation considers the formal definition of the unidirectional translation of UMLSM in GSPN provided in [26]. Starting from the latter, the relationships between UMLSM 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 UMLSM-GSPN bidirectional transformation implemented via JTL is depicted in Listing 1. In the following listings, three dots are used in place of repetitive sections of code.

```

1 transformation UMLGSPN (uml:umlsm, pn: 28 objects=s1:ptnet::Transition {
  ptnet) { 29 transitionKind="immediate"
2 ... 30 }
3 top relation StateMachine2PetriNet { 31 };
355 4 name: String; 38532 enforce domain pn p:ptnet::Page {
5 enforce domain uml statemachine:umlsm 33 objects=s2:ptnet::Arc {
  ::StateMachine { 34 source=s:ptnet::Place {}
6 name=name, 35 target=s1:ptnet::Transition {}
7 region=r:umlsm::Region {} 36 }
360 8 }; 39037 };
9 enforce domain pn petrinet:ptnet:: 38 where {
  PetriNet { 39 s.doActivity.ocIsUndefined()
10 id=name, 40 }
11 pages=p:ptnet::Page {} 41 }
365 12 }; 39542
13 where { 43 relation StateActivity2Pattern {
14 State2Pattern(r, p); 44 enforce domain uml r:umlsm::Region {
15 StateActivity2Pattern(r, p); 45 subvertex=s:umlsm::State {
16 Transition2Pattern(r, p); 46 doActivity=a:umlsm::Activity {}
370 17 } 40047 }
18 } 48 };
19 49 enforce domain pn p:ptnet::Page { ...
20 relation State2Pattern { 50 };
21 enforce domain uml r:umlsm::Region { 50 enforce domain pn p:ptnet::Page { ...
375 22 subvertex=s:umlsm::State {} 405 };
23 }; 51 enforce domain pn p:ptnet::Page { ...
24 enforce domain pn p:ptnet::Page { 52 };
25 objects=s:ptnet::Place {} 52 enforce domain pn p:ptnet::Page {
26 }; 53 objects=s3:ptnet::Place {}
380 27 enforce domain pn p:ptnet::Page { 41054 };

```

```

55  enforce domain pn p:ptnet::Page { 100  enforce domain pn page:ptnet::Page {
56  objects=s4:ptnet::Arc { 101  objects=t4:ptnet::Place {}
57  source=s1:ptnet::Transition {} 102  };
58  target=s3:ptnet::Place {} 460 3  enforce domain pn page:ptnet::Page {
415 59  } 104  objects=t5:ptnet::Arc {
60  }; 105  source=s1:ptnet::Transition {}
61  enforce domain pn p:ptnet::Page { 106  target=t4:ptnet::Place {}
62  objects=s5:ptnet::Transition { 107  }
63  transitionKind="exponential" 465 8  };
420 64  } 109  enforce domain pn page:ptnet::Page {
65  }; 110  objects=t6:ptnet::Transition {
66  enforce domain pn p:ptnet::Page { 111  transitionKind="immediate"
67  objects=s6:ptnet::Arc { 112  }
68  source=s3:ptnet::Place {} 470 13  };
425 69  target=s5:ptnet::Transition {} 114  enforce domain pn page:ptnet::Page {
70  } 115  objects=t7:ptnet::Arc {
71  }; 116  source=t4:ptnet::Place {}
72  } 117  target=t6:ptnet::Transition {}
73  475 18  };
430 74  relation Transition2Pattern { 119  };
75  enforce domain uml region:umlsm:: 120  }
    Region { 121  }
76  transition=t:umlsm::Transition { 122  relation TransitionDaStep2Pattern {
77  source=s:umlsm::State {} 480 23  name:String;
435 78  } 124  prob:String;
79  }; 125  enforce domain uml region:umlsm::
    Region {
80  enforce domain pn page:ptnet::Page { 126  transition=t:umlsm::DaStep {
81  objects=t:ptnet::Place {} 127  name=name,
82  }; 485 27  occurrenceProb=prob,
440 83  enforce domain pn page:ptnet::Page { 128  source=s:umlsm::State {}
84  objects=t1:ptnet::Transition { 129  }
85  transitionKind="immediate" 130  };
86  } 131  };
87  }; 490 32  enforce domain pn page:ptnet::Page {
445 88  enforce domain pn page:ptnet::Page { 133  ... };
89  objects=t2:ptnet::Arc { 134  enforce domain pn page:ptnet::Page {
90  source=t:ptnet::Place {} 135  objects=t1:ptnet::Transition {
91  target=t1:ptnet::Transition {} 136  id=name,
92  } 495 36  weight=prob
450 93  }; 137  transitionKind="immediate"
94  enforce domain pn page:ptnet::Page { 138  }
95  objects=t3:ptnet::Arc { 139  };
96  source=t:ptnet::Place {} 140  enforce domain pn page:ptnet::Page {
97  target=s1:ptnet::Transition {} 500  ... };
455 98  } 141  enforce domain pn page:ptnet::Page {
99  };  ... };

```

```

142 enforce domain pn page:ptnet::Page { ... };
... };
505 143 enforce domain pn page:ptnet::Page { 145 enforce domain pn page:ptnet::Page {
... };
144 144 enforce domain pn page:ptnet::Page { 147 ...

```

Listing 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 UMLSM metamodel and the variable `pn` that matches models conforming to the GSPN metamodel (based on the standard Petri Net Markup Language (PNML) [27]). 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 UMLSM domain induces the creation of a list of elements in the GSPN domain, the relation enforces multiple patterns. In particular, for each UMLSM `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. 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 `UMLSM 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 `UMLSM 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.

3.4.2. GSPN subnets composition

The `UMLSM GSPN` transformation in the previous section generates a separate GSPN for each `UMLSM`. 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 `UMLSD`.

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 7a, we need

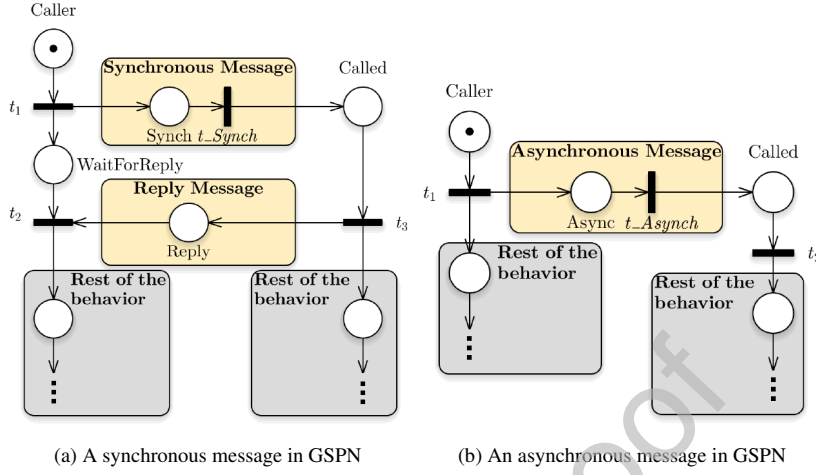


Figure 7: GSPN subnets composition

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 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 [28]. 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 2.


```

1 ... 41
2 relation MessageSynch2Pattern { 42
585 3 enforce domain uml m:umlsm::Message { 43
4 messageSort="synchCall" 44
5 }; 45
6 enforce domain pn p:ptnet::Page { 46
7 objects=s:ptnet::Place {} 47
590 8 }; 48
9 enforce domain pn p:ptnet::Page { 49
10 objects=s1:ptnet::Transition {} 50
11 transitionKind="immediate" 51
12 } 52
595 13 }; 53
14 enforce domain pn p:ptnet::Page { 54
15 objects=s2:ptnet::Arc { 55
16 source=s:ptnet::Place {} 56
17 target=s1:ptnet::Transition {} 57
600 18 } 58
19 }; 59
20 enforce domain pn p:ptnet::Page { 60
21 objects=s3:ptnet::Arc { 61
22 source=c:ptnet::Transition {} 62
605 23 target=s:ptnet::Place {} 63
24 } 64
25 }; 65
26 enforce domain pn p:ptnet::Page { 66
27 objects=s4:ptnet::Arc { 67
610 28 source=s1:ptnet::Transition {} 68
29 target=r:ptnet::Place {} 69
30 } 70
31 }; 71
32 when { 72
615 33 State2Pattern(c, c); 73
34 State2Pattern(r, r); 74
35 } 75
36 } 76
37 } 77
620 38 relation MessageAsynch2Pattern { 78
39 enforce domain uml m:umlsm::Message { 79
40 messageSort="asynchCall" 80

```

Listing 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 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 1) in the `when` clause. In the opposite direction, for each occurrence of the described GSPN pattern a correspondent reply Message is generated;

3.4.3. 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
 695 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 3.

```

1 ... 30 };
2 relation Component2Page { 31 enforce domain pn p:ptnet::Page {
3   name:String; 32   objects=s1:ptnet::Transition {
700 4   enforce domain uml c:umlsm::Component { 33     transitionKind="immediate"
5     { 34   }
6     name=name 35 };
7   }; 36 enforce domain pn p:ptnet::Page {
705 8   checkonly domain pn p:ptnet::Page { 37   objects=s2:ptnet::Arc {
9     id=name 740 38     source=s:ptnet::Place {}
10   }; 39     target=s1:ptnet::Transition {}
11   } 40   }
12   }; 41 };
12 relation Interface2Pattern { 42 enforce domain pn p1:ptnet::Page {
710 13   enforce domain uml i:umlsm::Interface { 43   objects=s3:ptnet::Arc {
14     { 44     source=c:ptnet::Transition {}
15     ownedOperation=o:umlsm::Operation {} 45     target=s:ptnet::Place {}
16   }; 46   }
17   enforce domain uml receiver:umlsm:: 47 };
715   Component { 750 48   enforce domain pn p2:ptnet::Page {
18     interfaceRealization=ir:umlsm:: 49   objects=s4:ptnet::Arc {
19     InterfaceRealization { 50     source=s1:ptnet::Transition {}
20     supplier=i:umlsm::Interface {}, 51     target=r:ptnet::Place {}
21     contract=i:umlsm::Interface {} 52   }
720 20   } 755 53 };
21 }; 54 when {
22   enforce domain uml caller:umlsm:: 55   Component2Page(caller, p1);
23   Component { 56   Component2Page(receiver, p3);
24   packagedElement=d:umlsm::Dependency 57 }
725 25   { 760 58   where {
26     client=caller:umlsm::Component {}, 59     s3.source.containerPage.id <>
27     supplier=i:umlsm::Interface {} 60     s3.target.containerPage.id;
28   } 61     s4.source.containerPage.id <>
29   }; 62     s4.target.containerPage.id;
730 28   enforce domain pn p:ptnet::Page { 765 63   }
29   objects=s:ptnet::Place {} 64 ...

```

Listing 3: A fragment of the UML^{JASA}-GSPN bidirectional transformation

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

- 770
 • `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.
- 775
 • `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
 780
 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
 785
 Pages.

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

4. JASA at work

In this section, we present the approach in practice with the aim of illustrating the
 790
 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 4.1). First, the system has been modelled by means of UML annotated with the MARTE DAM profile (as described in
 795
 Section 2). 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^{JASA}-GSPN bidirectional transformation

has been run in the forward direction to generate the GSPN models (as described in Section 4.2); after performing the GSPN analysis, a set of refactoring actions have
 800 been performed on the GSPNs on the basis of the obtained results (as described in Section 4.3). Finally, the UML^{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 4.4).

4.1. Environmental Control System (ECS) modeling

805 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
 810 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 *Soil-HumiditySensor* respectively measure air temperature, air humidity and soil humidity;
 815 *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
 820 UML Sequence Diagrams: *Monitoring Conditions*, in Figure 9a, in which a timer periodically activates a procedure to check environment conditions, *Remote Monitoring*, in Figure 9b, in which the air humidity is continuously monitored and the *Greenhouse-Controller* notifies the user when the value exceeds the corresponding threshold, and *Managing Temperature*, in Figure 9c, that defines the procedure for the activation of
 825 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.

830 The resulting State Machine diagram is shown in Figure 8.

UML *Transition* and *Message* elements that may fail are annotated with the *DaStep* stereotype from DAM, as depicted in Figures 8 and 9, 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
835 *failure* can be used to set the failure probability as the *occurrenceProb* real value.

The considered models, specified in UML, are available online ⁷.

4.2. Analysis model generation

The first operational step of our approach consists in the execution (in the forward direction) of the transformation presented in 3.4 (UML^{JASA}-GSPN) within the JTL
840 framework. For each scenario, from a Sequence Diagram and the set of involved State Machines, this execution generates a GSPN. The transformation UMLSM-GSPN in Section 3.4.1 creates a GSPN subnet for each State Machine. As an example, Figure 11 shows a fragment of the GSPN obtained for the *Managing Temperature* scenario (Figure 9c). The GSPN subnets visible in the figure are generated from the *TemperatureSensor*, *AirConditioner* and *GreenhouseController* State Machines in Figure 10,
845 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 3.4.2.

In general, the composition of subnets obtained from this step is based on interactions
850 among components, as appearing in Sequence Diagrams. In particular, synchronous and asynchronous messages are mapped to the corresponding patterns presented in Section 3.4.2⁸.

⁷ECS UML models: <https://github.com/SEALABQualityGroup/JASA/tree/master/UML>

⁸The GSPNs generated for each scenario are available at <https://github.com/SEALABQualityGroup/JASA/tree/master/GSPN>.

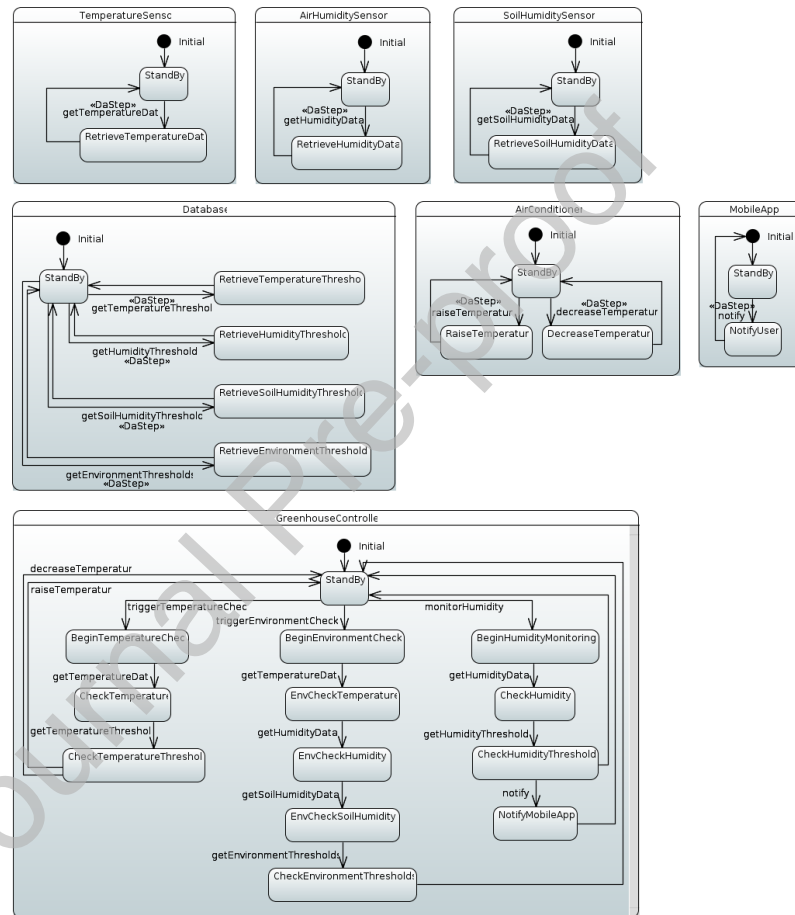


Figure 8: UML State Machine Diagram of the ECS components

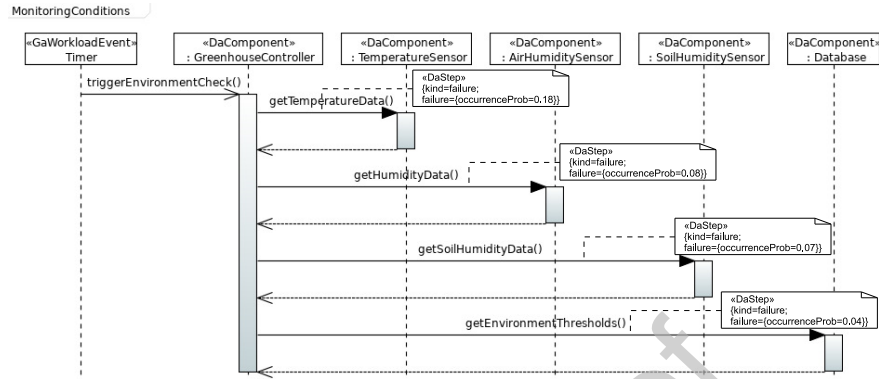
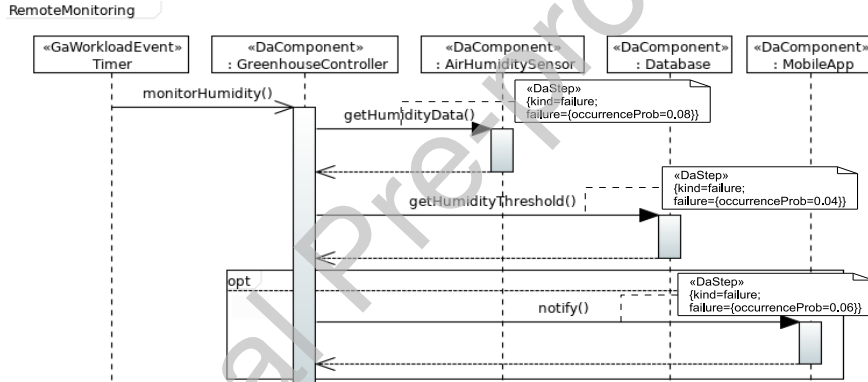
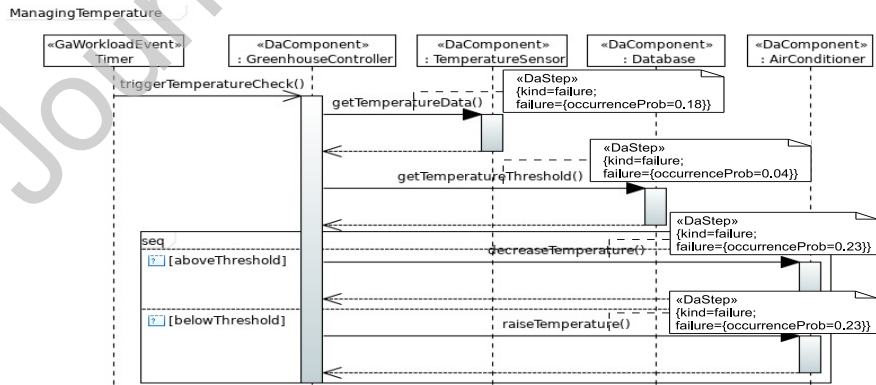
(a) The *Monitoring Conditions* scenario(b) The *Remote Monitoring* scenario(c) The *Managing Temperature* scenario

Figure 9: UML Sequence Diagrams of the ECS scenarios

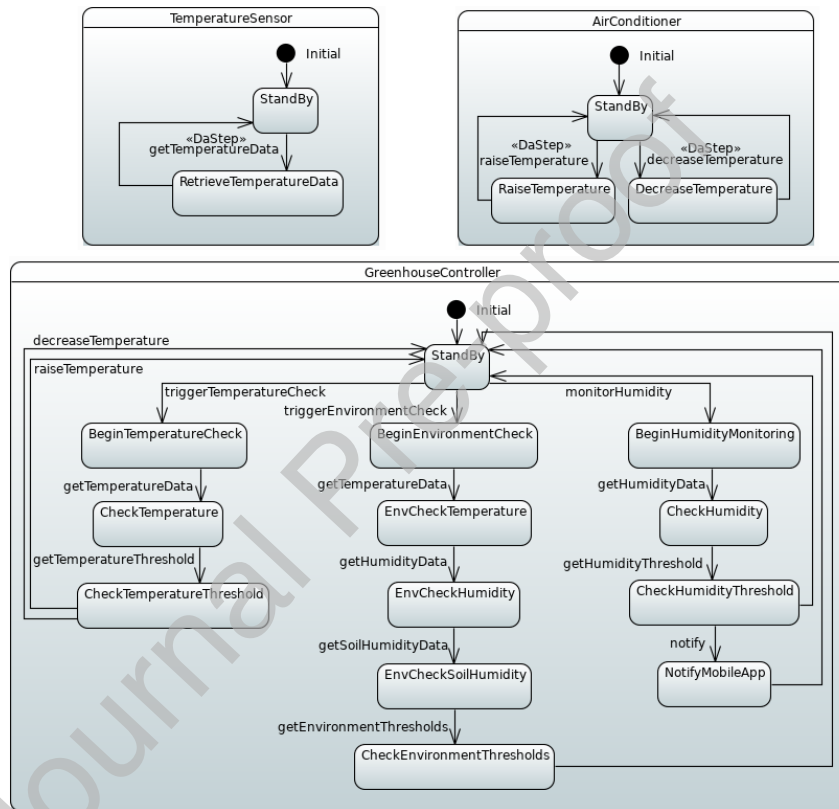


Figure 10: UML State Machines of the *TemperatureSensor*, *AirConditioner* and *GreenhouseController* components

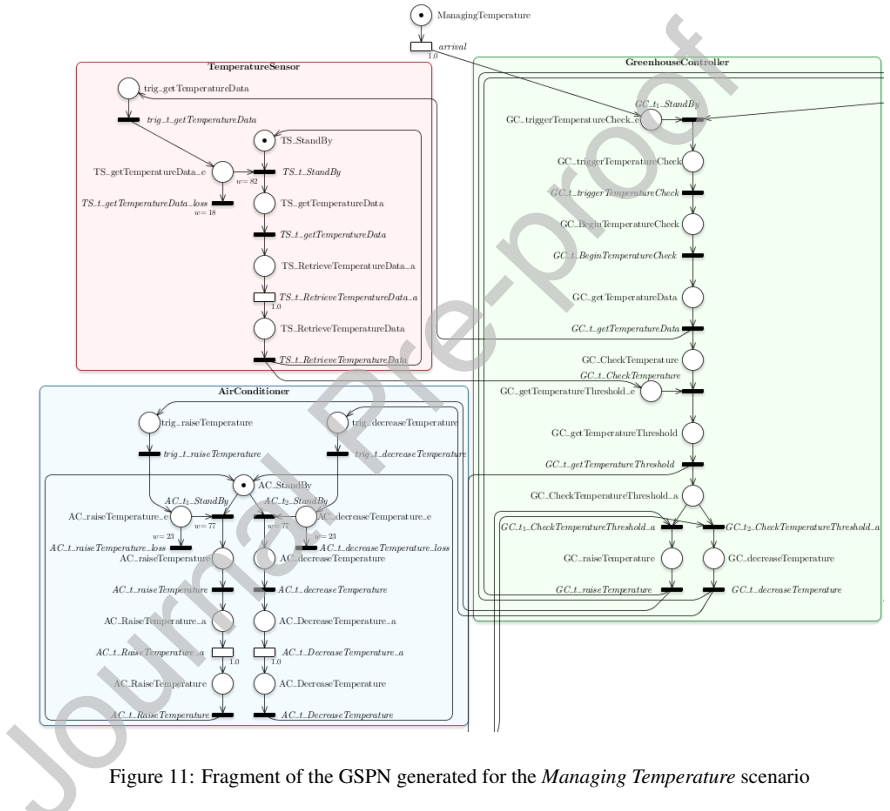


Figure 11: Fragment of the GSPN generated for the *Managing Temperature* scenario

4.3. Analysis results and refactoring

In this section, first we play with a simple case for checking whether patterns induce
 855 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).

860 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
 865 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' .

In general, availability metrics of an GSPN model can be defined as reward
 functions on the reachability graph [29]. Let M^0 be the initial marking, and
 870 $r_M = \{1 \text{ if } M \in O, 0 \text{ if } M \in F\}$ 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 mark-
 ing M at time instant t can be expressed as $\sigma_M(t) = Pr \{X(t) = M\}$. Steady state
 probability can be computed as $\sigma_M = \lim_{t \rightarrow \infty} \sigma_M(t)$, and it represents the probabil-
 875 ity 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_∞ is to be interpreted as
 the percentage of time the system is not in a failure state after running for a sufficiently
 880 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.

885 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.

The GreatSPN solver [30] is used to derive the reachability graph of markings in the net and to compute the corresponding values of σ_M . In the initial marking of the net, a
 890 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. Since we assume that the operations belonging to the same component fail with the same probability, we report in Table 1 the initial failure probabilities of every component in ECS.

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

Table 1: Initial failure probabilities of components in ECS

895 The steady state availability index can be computed by considering both the aforementioned initial marking of the GSPN and the failure probabilities. The resulting indices for the three scenarios we considered are reported in Table 2.

Monitoring Conditions	0.985392
Remote Monitoring	0.991672
Managing Temperature	0.977984

Table 2: Steady state availability of execution scenarios

In order to establish the effectiveness of the fault tolerance patterns presented in Section 2.3, we apply each of them on the *TemperatureSensor* component in the *Managing*
 900 *Temperature* scenario. The steady state availability resulting in each case is reported in Table 3. The results show, as expected, that the application of the fault tolerance pat-

terms 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*) [31].

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

Table 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 [32]. Since the *Active Replication* pattern is widely used in practice to deploy high availability databases [33], we apply it to the *Database* component in each scenario. The results obtained from this refactoring are discussed in Section 5.1.3. 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.

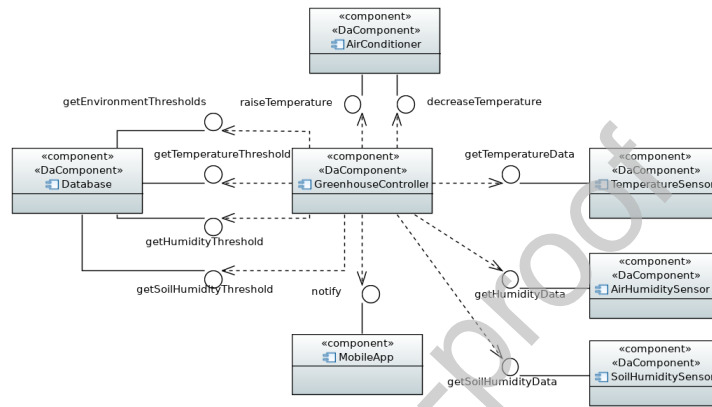
4.4. Change propagation

After the analysis and refactoring step, the UML^{JASA}-GSPN bidirectional transformation is applied in backward direction on the refactored GSPN model. In particular,

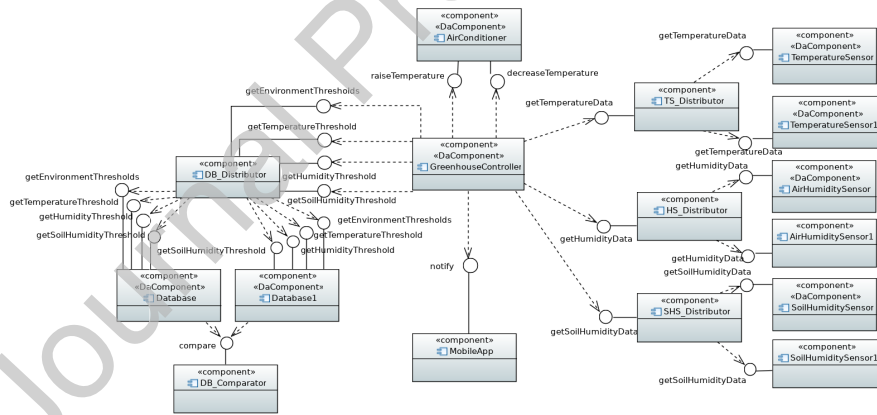
the refactored UML Sequence and State Machine Diagrams are generated for each
 925 scenario. These new diagrams contain the changes applied to the GSPN during the
 refactoring step and propagated back by the execution of the transformation.
 Moreover, the back propagation of changes generates additional software components.
 The updated Component Diagram is reported in Figure 12b. In particular:

- *Monitoring Conditions*: the components *TS_Distributor*, *TemperatureSensor1*,
 930 *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
 935 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*,
 940 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*;

945 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 without any modification. In addition, each state machine corresponding to replicas in the *Semi-Active Replication* pattern (i.e., all sensors' replicas) includes a
 950 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



(a) Initial Component Diagram of ECS



(b) Refactored Component Diagram of ECS

Figure 12: Component Diagrams before and after the change propagation

TemperatureSensor1) and restored (*TemperatureSensor*) from the *Semi-Active Replication* pattern on the *TemperatureSensor* component are included in Figure 13.

955 The refactored UML Sequence Diagrams for the scenarios *Monitoring Conditions*, *Remote Monitoring*, and *Managing Temperature* are shown in Figures 14, 15, and 16, 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.

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

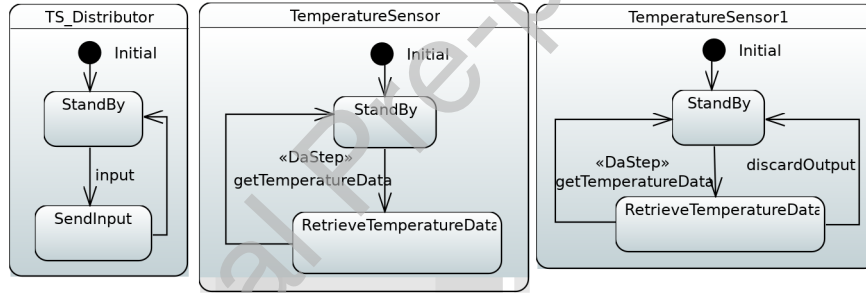


Figure 13: UML State Machines generated from the back propagation of the *Semi-Active Replication* pattern on *TemperatureSensor*

965 5. Results evaluation

In this section we 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?

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

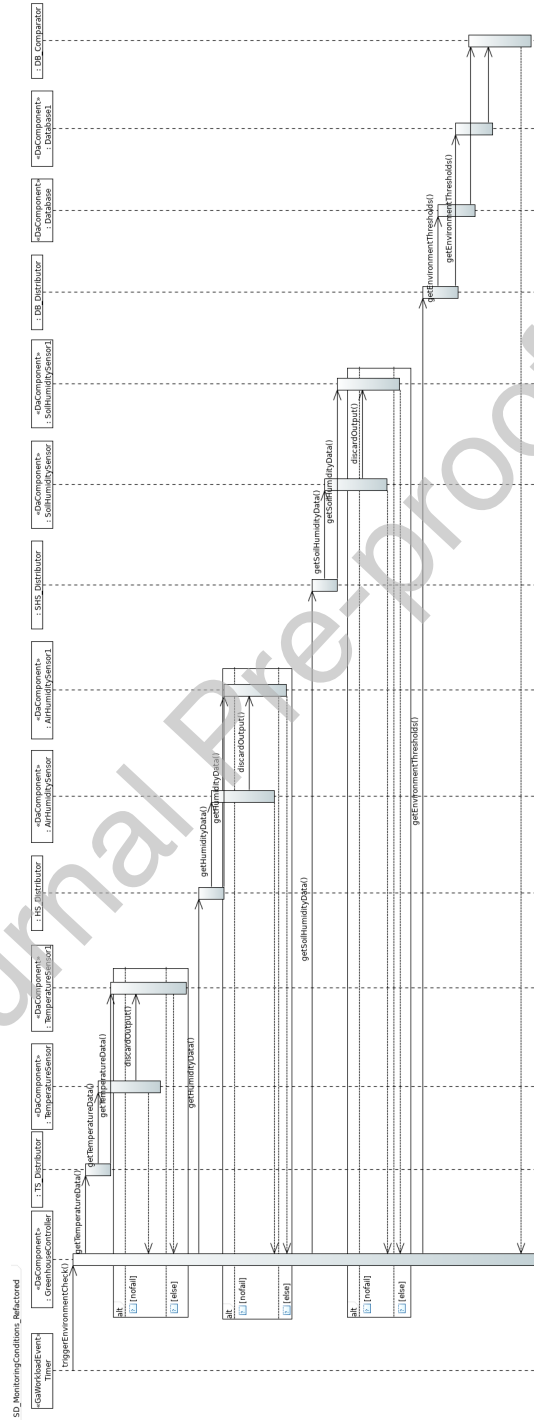


Figure 14: UML Sequence Diagram of the Monitoring Conditions scenario

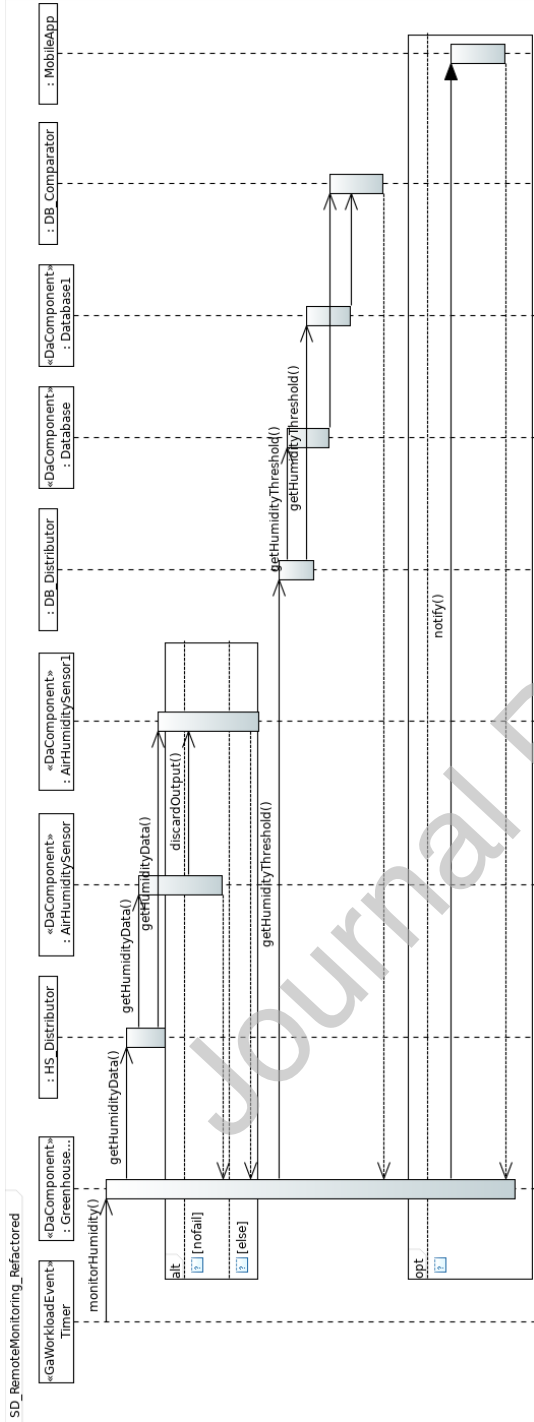


Figure 15: UML Sequence Diagram of the Remote Monitoring scenario

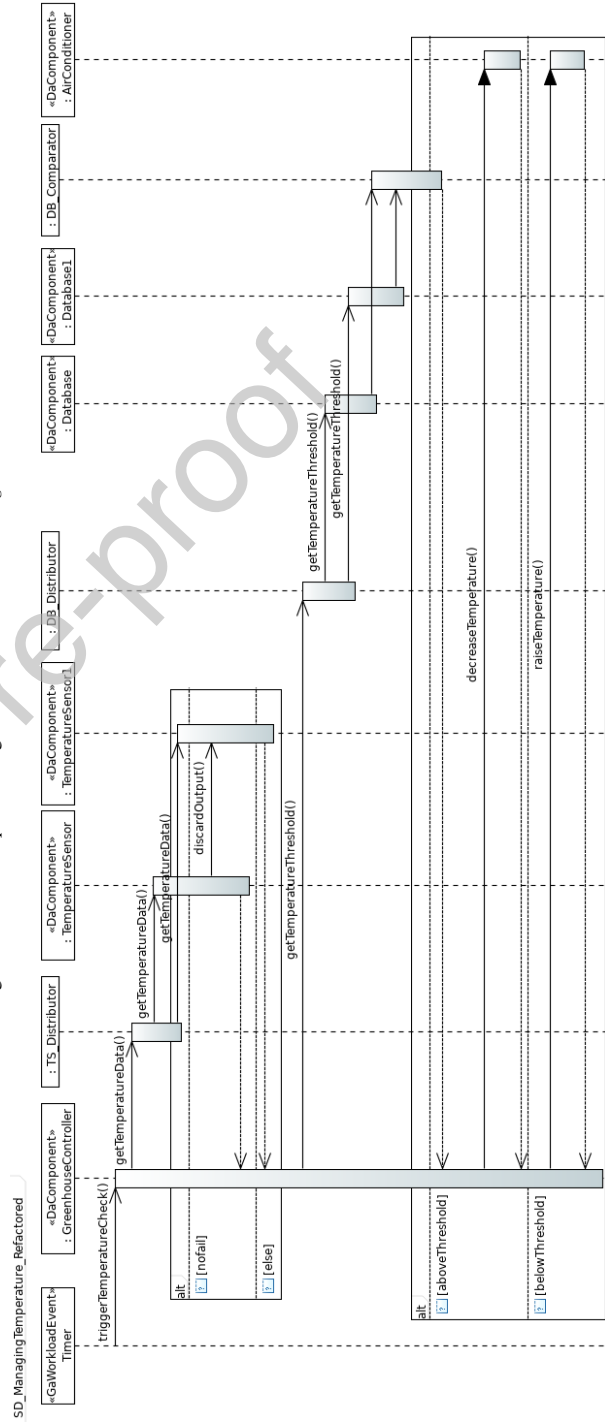


Figure 16: UML Sequence Diagram of the Managing Temperature scenario

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

The evaluation has been conducted by applying the approach to the Environmental Control System (ECS) example application, described in Section 4. The software design has been modeled by means of UML diagrams; then, for each scenario the following process has been applied:

- the UML^{JASA}-GSPN bidirectional transformation has been executed in forward direction: thus, a Sequence diagram and a set of State Machine diagrams have been given as input and the corresponding GPSN has been obtained as output;
- the resulting GPSN has been analyzed to obtain the steady state availability index, and it has been then refactored on the basis of the fault tolerance patterns defined in Section 2.3;
- the UML^{JASA}-GSPN bidirectional transformation has been executed in backward direction: thus, the changes performed on the GPSN have been propagated to the UML model.

5.1. Insights on Research Questions

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.

5.1.1. *RQ1: Analyzability 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 GPSN models can support our analysis, we refer to a set of basic behavioral properties (as introduced in [34]) discussed as following.

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.

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, we report the cardinality of the reachability set $RS(M_0)$ for each scenario. In particular, the *Initial* values refer to the GSPN models obtained by applying the UML^{JASA}-GSPN bidirectional transformation, whereas the *Refactored* values refer to the GSPN models after the refactoring described in Section 4.3. 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: 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 [35].

By considering that (i) a GSPN is bounded if and only if its reachability graph is finite [36], and (ii) we showed in Table 4 that finite reachability sets can be computed

before and after the refactoring, we can assert that all the GSPNs (i.e., initial and refac-
 1025 tored 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.

1030 *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.

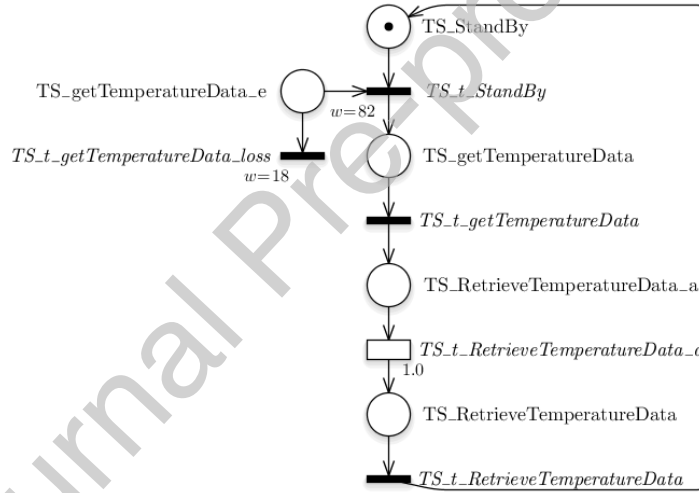


Figure 17: GSPN subnet of the *Temperature Sensor*

In our experiment, we can observe that all the GSPNs (both initial and refactored)
 1035 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 *LI-live*, as they
 can be fired at least once in some firing sequence starting from the initial mark-
 ing M_0 . As an example, the transition $TS.t_getTemperatureData_loss$ in Figure 17
 is a *LI-live* one, as it can potentially fire only once, when at least one token is
 1040 in the place $TS_getTemperatureData_e$. All the other transitions in the GSPN are

L3-live, since they can fire infinitely, as well as all transitions in Figure 17 except *TS.t.getTemperatureData_loss*. In general, liveness of obtained GSPNs can be checked by the GreatSPN solver that we have adopted.

5.1.2. RQ2: Validity of the refactored architecture

1045 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 [37].

Correctness:

1050 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). 1055
- 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 3.3). 1060
- 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 GSPN 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), an additional state machine that contains the 1065

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: $\vec{T} : M \times N \rightarrow N$ and $\overleftarrow{T} : M \times N \rightarrow M$. T is said to be *correct* if for any pair of models $m \in M$ and $n \in N$, $T(m, \vec{T}(m, n))$ and $T(\overleftarrow{T}(m, n), n)$ [38]. The capability of the JTL framework to correctly execute the transformation is discussed in [12, 23]. 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 3.3, 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^{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* [38]. 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^{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 [12, 23].

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^{JASA}-GSPN bidirectional transformation specifies the mapping between UML and GSPN elements by preserving the consistency of names and structure (e.g., in the GSPN 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

1125 traceability engine of JTL is able to preserve the missing information and restore
it, thus avoiding loss of information [39].

5.1.3. 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 5. It is, instead, less obvious
1130 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.

1135 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: $R(\mathfrak{A}_0) \rightarrow \mathfrak{A}'$.
1140

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
1145 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 18 shows how the steady state availability of the *Managing Temperature* scenario (ES^{MT}) is altered when varying the failure probability of the *Temperature-Sensor* component (FP^{TS}) in the interval $[0.01, 0.5]$.
1150

The figure shows the availability $Avail(\mathfrak{A}_i, ES^{MT}, FP_{[0.01, 0.5]}^{TS})$ 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 after applying the refactoring changes mentioned at the end of Section 4.3, 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 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 1. The measures highlight that the

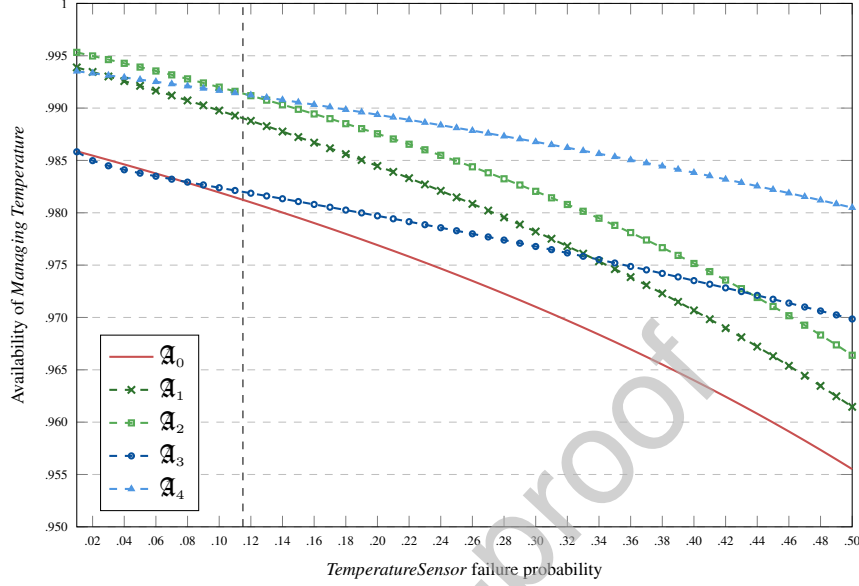


Figure 18: Availability of *Managing Temperature* scenario vs. *TemperatureSensor* failure probability under single refactoring actions

application of the fault tolerance patterns has improved, as expected, the availability in each considered scenario.

	\mathcal{A}_0	\mathcal{A}'
Monitoring Conditions	0.985392	0.990771
Remote Monitoring	0.991672	0.994207
Managing Temperature	0.977984	0.993316

Table 5: Steady state availability computed on the initial (\mathcal{A}_0) and refactored (\mathcal{A}') architecture

1185 The *Managing Temperature* scenario had an improvement of 15.332×10^{-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 5.379×10^{-3} after the application of the *Semi-Active Replication* pattern on *TemperatureSensor*, *HumiditySensor* and *SoilHumiditySensor*,
 1190 and the *Active Replication* pattern on the *Database* component. Finally, the *Remote Monitoring* scenario had an improvement of 2.535×10^{-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 19, 20, and 21 report the results for the *Monitoring Conditions*, *Remote Monitoring*, and *Managing Temperature* scenarios, respectively.

The curve notation is the same as for Figure 18. For example, in Figure 19 we depict $Avail(\mathfrak{A}_0, ES^{MT}, FP_{[0.01, 0.5]}^{TS})$ and $Avail(\mathfrak{A}_0, ES^{MT}, FP_{[0.01, 0.5]}^{DB})$ as solid curves, whereas $Avail(\mathfrak{A}', ES^{MT}, FP_{[0.01, 0.5]}^{TS})$ and $Avail(\mathfrak{A}', ES^{MT}, FP_{[0.01, 0.5]}^{DB})$ 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 19 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.

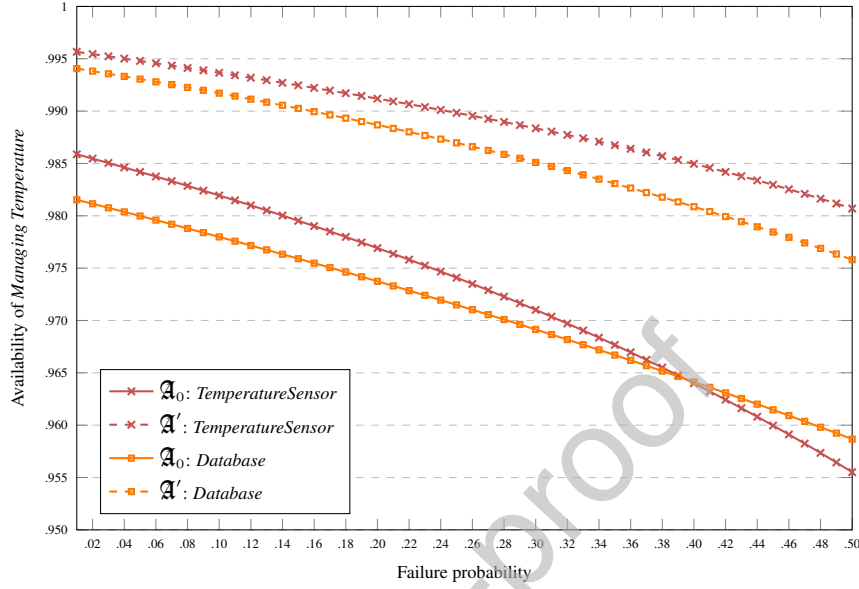


Figure 19: Availability of *Managing Temperature* scenario on the initial (\mathcal{A}_0) and refactored (\mathcal{A}') architecture vs. failure probabilities under combined refactoring actions

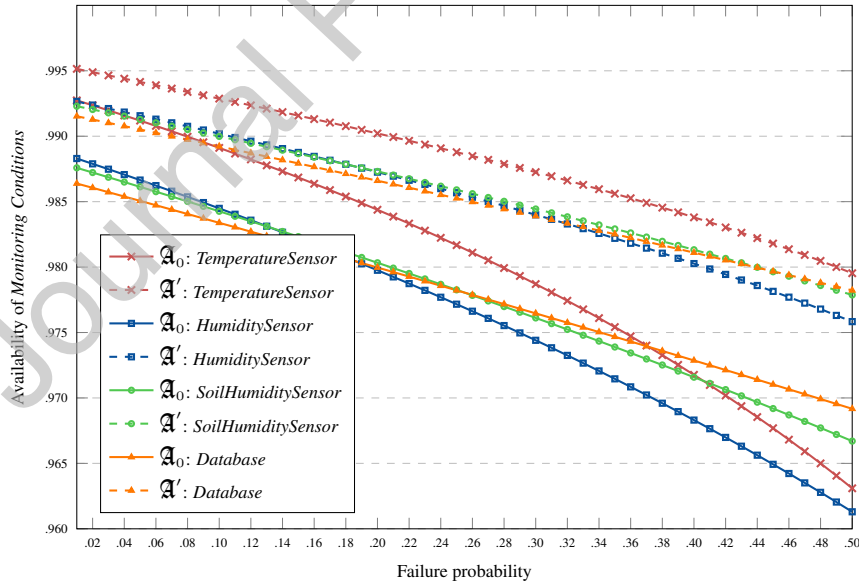


Figure 20: Availability of *Monitoring Conditions* scenario on the initial (\mathcal{A}_0) and refactored (\mathcal{A}') architecture vs. failure probabilities under combined refactoring actions

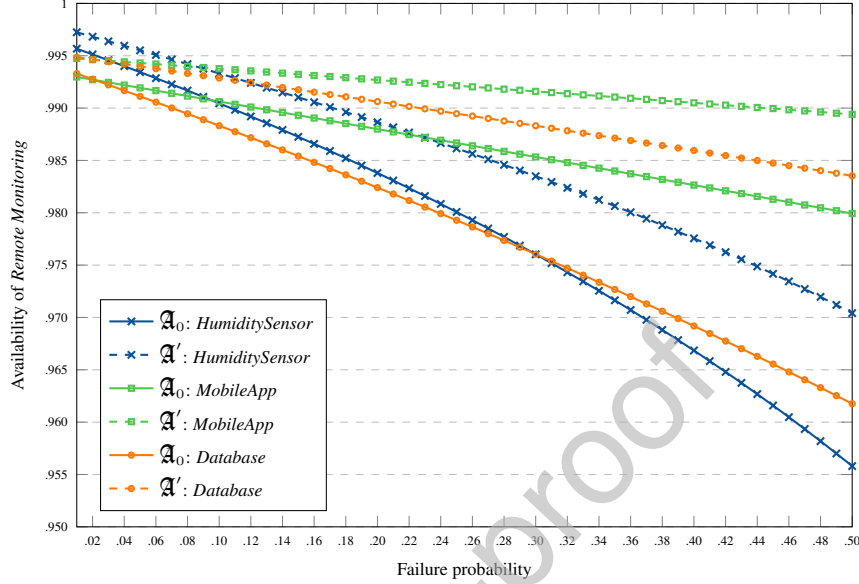


Figure 21: Availability of *Remote Monitoring* scenario on the initial (A_0) and refactored (A') architecture vs. failure probabilities under combined refactoring actions

5.2. 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

1225 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 [28]; ii) by considering already existing fault tolerance patterns [15]; iii) by considering well-established methods for stochastic availability assessment
1230 [29]; iv) by delegating the availability analysis to an external solver [30]. 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

1235 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. 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.

1250 *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.

1255 *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 ⁹, and the experiment can be reproduced locally within the JTL framework.

⁹<https://github.com/SEALABQualityGroup/JASA>

6. Related Work

Several approaches have been introduced in the last few years to derive analysis models from annotated software models. Bondavalli et al. [40] 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. In [41], Huszer et al. propose a transformation of UML 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. [42] 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. [26] 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 in [26], 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.

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, in [43] an evolutionary algorithm is introduced 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 in [43] 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 [44].

In the context of bidirectional model transformations, a round-trip engineering process between models representing different views of the same system is formally defined in [45]. In the performance analysis domain, in a previous paper [9], we have introduced a similar approach to the one presented in this paper. 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. In [46] two methods to tackle the problem of deriving architectural changes from model-based performance analysis results have been compared: (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.

In [47], the authors 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, as pointed in [48, 49].

To the best of our knowledge, this is the first paper proposing an automated propagation of changes performed on an availability model back to an architectural model. Even
 1325 though the scope of this paper 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.

7. Conclusion

1330 In this paper, we proposed JASA, a model-driven framework that supports a round-trip availability analysis process based on software architectural refactoring. We used bidirectional model transformations to map software architectures represented by UML models to GSPN analysis models and vice versa. In fact, after the analysis, the obtained GSPN is modified according to a proposed catalog of refactoring based on well-known
 1335 fault tolerance patterns. Finally, the changes are back propagated to the software architecture with the aim of improving the software availability. The effectiveness of our approach has been demonstrated on an Environmental Control System, in terms of ability to generate analyzable availability models from software architectures and valid software architectures from availability models. Also, we showed how to select more
 1340 effective fault-tolerance patterns in different execution scenarios.

Although we considered a set of well-known fault tolerance refactoring techniques, the approach can be extended to support further user-defined refactoring actions and to automate the application of such actions on GSPN models completely. In this respect, the bidirectional model transformation needs to be modified in order to cope with a
 1345 larger set of relationships. As a possible consequence of this modification, the change propagation from analysis to architectural models may result in the generation of multiple architectural alternatives, because a single refactoring action in a GSPN can be mapped in more than one refactored architectural model. This is our main future work direction, where we can still exploit the JTL transformation engine that is able to support
 1350 non-bijective mappings by generating all the alternative solutions according to the

specification. In the same direction, a further challenge for the future is to introduce a human-assisted process for choosing among multiple suggested alternatives. Finally, another line of future investigation encompasses the extension of the proposed methodology to further non-functional requirements such as reliability and safety.

References

- [1] D. Garlan, J. M. Barnes, B. R. Schmerl, O. Celiku, Evolution styles: Foundations and tool support for software architecture evolution, in: WICSA/ECSA, 2009, pp. 131–140. doi:10.1109/WICSA.2009.5290799.
- [2] H. Muccini, A. Bertolino, P. Inverardi, Using software architecture for code testing, IEEE Trans. Software Eng. 30 (3) (2004) 160–171. doi:10.1109/TSE.2004.1271170.
- [3] A. Tang, Y. Jin, J. Han, A rationale-based architecture model for design traceability and reasoning, Journal of Systems and Software 80 (6) (2007) 918–934. doi:10.1016/j.jss.2006.08.040.
- [4] V. Cortellessa, A. D. Marco, P. Inverardi, Non-functional modeling and validation in model-driven architecture, in: WICSA, 2007, p. 25. doi:10.1109/WICSA.2007.30.
- [5] D. Garlan, R. T. Monroe, D. Wile, Acme: an architecture description interchange language, in: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, November 10-13, 1997, Toronto, Ontario, Canada, 1997, p. 7.
URL <https://dl.acm.org/citation.cfm?id=782017>
- [6] P. Feiler, D. Gluch, J. Hudak, The architecture analysis & design language (aadl): An introduction (2006) 145.
- [7] S. Becker, H. Koziolk, R. H. Reussner, The palladio component model for model-driven performance prediction, Journal of Systems and Software 82 (1)

- (2009) 3–22. doi:10.1016/j.jss.2008.03.066.
 URL <https://doi.org/10.1016/j.jss.2008.03.066>
- [8] D. Schmidt, Guest Editor’s Introduction: Model-Driven Engineering, *Computer* 39 (2) (2006) 25–31. doi:<http://doi.ieeecomputersociety.org/10.1109/MC.2006.58>.
 1380
- [9] R. Eramo, V. Cortellessa, A. Pierantonio, M. Tucci, Performance-driven architectural refactoring through bidirectional model transformations, in: *QoSA*, 2012, pp. 55–60. doi:10.1145/2304696.2304707.
- [10] V. Cortellessa, A. D. Marco, P. Inverardi, *Model-Based Software Performance Analysis*, Springer, 2011. doi:10.1007/978-3-642-13621-4.
 1385
- [11] S. Bernardi, J. Merseguer, D. C. Petriu, *Model-Driven Dependability Assessment of Software Systems*, Springer, 2013.
- [12] A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio, JTL: a bidirectional and change propagating transformation language, in: *SLE10*, 2010, pp. 183–202.
 1390
- [13] V. Cortellessa, R. Eramo, M. Tucci, Availability-driven architectural change propagation through bidirectional model transformations between UML and petri net models, in: *IEEE International Conference on Software Architecture, ICSA 2018*, 2018, pp. 125–134. doi:10.1109/ICSA.2018.00022.
 1395 URL <https://doi.org/10.1109/ICSA.2018.00022>
- [14] M. C. Otero, J. J. Dolado, Evaluation of the comprehension of the dynamic modeling in uml, *Information and Software Technology* 46 (1) (2004) 35 – 53. doi:[https://doi.org/10.1016/S0950-5849\(03\)00108-3](https://doi.org/10.1016/S0950-5849(03)00108-3).
- [15] T. Saridakis, A system of patterns for fault tolerance, in: *Proceedings of 2002 EuroPLoP Conference*, 2002.
 1400
- [16] A. Avizienis, J. Laprie, B. Randell, C. E. Landwehr, Basic concepts and taxonomy of dependable and secure computing, *IEEE Trans. Dependable Sec. Comput.* 1 (1)

- (2004) 11–33. doi:10.1109/TDSC.2004.2.
 URL <https://doi.org/10.1109/TDSC.2004.2>
- 1405 [17] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, *Modelling with Generalized Stochastic Petri Nets*, John Wiley & Sons, Inc., 1994.
- [18] Unified modeling language, OMG, version 2.5 (2015).
 URL <http://www.omg.org/spec/UML/2.5/>
- [19] S. Bernardi, J. Merseguer, D. C. Petriu, A dependability profile within MARTE,
 1410 *Software and System Modeling* 10 (3) (2011) 313–336. doi:10.1007/s10270-009-0128-1.
- [20] A UML profile for MARTE: modeling and analysis of real-time embedded systems, OMG (2008).
 URL <http://www.omg.org/omgmarte/>
- 1415 [21] S. Hidaka, M. Tisi, J. Cabot, Z. Hu, Feature-based classification of bidirectional transformation approaches, in: *SOSYM*, 2015, pp. 1–22.
- [22] P. Stevens, A Landscape of Bidirectional Model Transformations, in: *GTTSE*, Springer, 2008, pp. 408–424.
- [23] R. Eramo, A. Pierantonio, G. Rosa, Managing uncertainty in bidirectional model
 1420 transformations, in: *SLE*, 2015, pp. 49–58.
- [24] MOF Query/View/Transformation - QVT, OMG (2016).
- [25] M. Gelfond, V. Lifschitz, The Stable Model Semantics for Logic Programming, in: *ICLP*, 1988, pp. 1070–1080.
- [26] S. Bernardi, J. Merseguer, QoS Assessment via Stochastic Analysis, *IEEE Inter-*
 1425 *net Computing* 10 (2006) 32–42. doi:10.1109/MIC.2006.63.
- [27] M. Weber, E. Kindler, The petri net markup language, in: *Petri Net Technology for Communication-Based Systems - Advances in Petri Nets*, 2003, pp. 124–144.
 doi:10.1007/978-3-540-40022-6_7.

- [28] S. Bernardi, S. Donatelli, J. Merseguer, From UML sequence diagrams and state-charts to analysable petrinet models, in: Workshop on Software and Performance, 2002, pp. 35–45. doi:10.1145/584369.584376.
- [29] K. Goseva-Popstojanova, K. S. Trivedi, Stochastic modeling formalisms for dependability, performance and performability, in: Performance Evaluation: Origins and Directions, 2000, pp. 403–422.
- [30] G. Chiola, G. Franceschinis, R. Gaeta, M. Ribaud, Greatspn 1.7: Graphical editor and analyzer for timed and stochastic petri nets, Performance Evaluation 24 (1-2) (1995) 47–68. doi:10.1016/0166-5316(95)00008-L.
- [31] M. Toeroe, F. Tam, Service Availability: Principles and Practice, Wiley, 2012.
- [32] D. Powell, I. Bey, J. Leuridan (Eds.), Delta Four: A Generic Architecture for Dependable Distributed Computing, Springer-Verlag, Berlin, Heidelberg, 1991.
- [33] B. Kemme, Replication for Availability and Fault Tolerance, Springer, 2018, pp. 1–7. doi:10.1007/978-1-4614-8265-9_80723.
- [34] T. Murata, Petri nets: Properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580. doi:10.1109/5.24143.
- [35] M. Ajmone Marsan, G. Conte, G. Balbo, A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems, ACM Trans. Comput. Syst. 2 (2) (1984) 93–122. doi:10.1145/190.191.
- [36] L. Popova-Zeugmann, Time and Petri Nets, Springer, 2013.
- [37] R. N. Taylor, N. Medvidovic, E. M. Dashofy, Software Architecture - Foundations, Theory, and Practice, Wiley, 2010.
- [38] P. Stevens, Bidirectional model transformations in QVT: semantic issues and open questions, Software and System Modeling 9 (1) (2010) 7–20. doi:10.1007/s10270-008-0109-9.

- [39] R. Eramo, A. Pierantonio, M. Tucci, Improved traceability for bidirectional model transformations, in: *Proceedings of MODELS 2018 Workshops co-located with MODELS 2018*, 2018, pp. 306–315.
- [40] A. Bondavalli, M. D. Cin, D. Latella, I. Majzik, A. Pataricza, G. Savoia, Dependability analysis in the early phases of uml-based system design, *Computer Systems Science And Engineering* 16 (2001) 265 – 275.
- [41] G. Huszerl, K. Kosmidis, M. D. Cin, I. Majzik, A. Pataricza, Quantitative analysis of uml statechart models of dependable systems, *The Computer Journal* 45 (2000) 260–277.
- [42] S. Mustafiz, X. Sun, J. Kienzle, H. Vangheluwe, Model-driven assessment of system dependability, *Software & Systems Modeling* 7 (4) (2008) 487–502. doi:10.1007/s10270-008-0084-1.
- [43] A. Martens, H. Koziolk, S. Becker, R. H. Reussner, Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms, in: A. Adamson, A. B. Bondi, C. Juiz, M. S. Squillante (Eds.), *Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering*, San Jose, California, USA, January 28-30, 2010, ACM, 2010, pp. 105–116. doi:10.1145/1712605.1712624. URL <https://doi.org/10.1145/1712605.1712624>
- [44] V. Cardellini, E. Casalicchio, V. Grassi, F. L. Presti, R. Mirandola, Qos-driven runtime adaptation of service oriented architectures, in: H. van Vliet, V. Issarny (Eds.), *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, ACM, 2009, pp. 131–140. doi:10.1145/1595696.1595718. URL <https://doi.org/10.1145/1595696.1595718>
- [45] T. Hettel, M. Lawley, K. Raymond, Model synchronisation: Definitions for round-trip engineering, in: *Theory and Practice of Model Transformations*, 2008, pp. 31–45.

- [46] D. Arcelli, V. Cortellessa, Software model refactoring based on performance analysis: better working on software or performance side?, in: FESCA, 2013, pp. 33–47. doi:10.4204/EPTCS.108.3.
- [47] J. Tatibouet, A. Cuccuru, S. Gérard, F. Terrier, Principles for the realization of an open simulation framework based on fuml (wip), Vol. 45, 2013.
- [48] L. Berardinelli, A. D. Marco, S. Pace, fuml-driven design and performance analysis of software agents for wireless sensor network, in: Software Architecture - 8th European Conference, ECSA 2014, 2014, pp. 324–339.
- [49] L. Berardinelli, P. Langer, T. Mayerhofer, Combining fuml and profiles for non-functional analysis based on model execution traces, 2013. doi:10.1145/2465478.2465493.