Automatic Derivation of Dependability and Fault Tolerance Analysis Models from Software Architecture

by

Naif Abdullah M. Alzahrani

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

Complex distributed software systems are increasingly employed to support and operate critical applications, so these systems should be highly dependable. The dependability of a software system is the ability to deliver trusted service and to avoid service failure that is more frequent and more severe than acceptable. Dependability attributes encompass a set of Non-Functional Properties (NFP), such as reliability and availability. The goal of this thesis is to develop a framework for integrating model-based dependability assessment techniques in the early stages of the software development process. This will support designers in taking the right design decisions and avoiding costly corrective actions later on, after the implementation and deployment have been completed.

The first objective of the thesis is to introduce an aspect-based modeling approach for representing the erroneous behavior of UML components and for capturing failure propagation between connected components. This approach, called Component Erroneous Behavior Aspect Modeling approach (CeBAM), supports the definition of aspects representing component erroneous behavior and composes the aspects automatically with the normal component behavior represented as a state-machine. It also enables the compatibility verification between interacting components and conformance verification of their internal behavior with the corresponding ports protocol behavior.

The next objective is to provide an automated transformation chain for deriving a Stochastic Reward Net (SRN) reliability analysis model from the software model in four phases: a) In-Place transformation to automate CeBAM approach, b) model-to-model transformation from UML software model extended with dependability annotations to SRN model, c) intermediate model-to-model transformation to build a CSPL model, and

d) model-to-text transformation from the generated CSPL model to a C-based SRN Programming Language (CSPL) specification. The derived SRN model is used first to verify the conformance and compatibility of the involved components in the selected scenario. Once it passes the verification, the CSPL code is generated and used to obtain the required reliability analysis results. These results are fed back to the designer, to support the selection of proper software fault tolerance mechanisms for the software. To provide the modeler with quantitative data for an easy comparison between different fault tolerance tactics in order to select the best solution for a given system, we developed the Single Version Fault Tolerance Aspect Modeling approach (SvFTAM), which captures architectural and behavioral models of single version fault tolerance tactics into a generic reusable aspect model.

The state space explosion is a well-known problem of state-based analysis models such as SRN. To address this problem, we use decomposition and reduction techniques of the derived SRN model to compute approximate system reliability measures. The SRN decomposition is guided by the component architecture of the software model and the protocol state machines describing the component interaction.

Acknowledgements

All praises are due to God (ALLAH), the most merciful and most compassionate. I am so grateful to ALLAH for his uncountable bounties on me including the ability to perform and write this thesis.

First of all, my deep and sincere appreciation goes to my thesis supervisor, *Professor Dorina C. Petriu,* for continued support, kind cooperation, and many long hours of discussions. This thesis would not have been possible without her guidance and assistance. I will never forget the first interview meeting when she said that: "I'm here to support and help students;" this statement is a source of motivation. Thank you for everything you provided me. You will be my model to follow in my academic life when I return to my country.

I am infinitely grateful to my parents and my lovely family. My father is the source of inspiration, and his unlimited support and advice helped me face any challenge. My mother is the source of love and her prayers are lightening my path to achieve my goals. My wife and my kids always supported and encouraged me to accomplish this achievement and other goals. I am thankful to all of them and GOD bless them.

I would also like to acknowledge the academic research support that I have received from Albaha University, the ministry of education, and their representatives in the Saudi bureau office here in Ottawa, Canada.

To my Grandmother Maytha Alghamdi

To my parents: Prof. Abdullah M. Alzahrani & Alya H. Alghamdi

To my beautiful wife: Huda A. Alzahrani

To my lovely kids:

Reema, Norah, Omar, Haneen,

and to anyone joining my lovely family in the future

Table of Contents

List of Tables

List of Illustrations

List of Acronyms

AMO	Aspect-Oriented Modeling
ASRs	Architectural Service Routes(s)
BSM	Behavioral State Machine
CBD	Component Based Development
CBSS	Component Based Software System
CDG	Component Dependency Graph
CeBAM	Component Erroneous Behavior Aspect Modeling
CFT	Component Fault Tree
CSPL	C-based SRN Programming Language
CTMC	Continues Time Markov Chain
DAM	Dependability Analysis and Modeling profile
DFT	Dynamic Fault Tree
DSPN	Deterministic Stochastic Petri Net
DTMC	Discrete Time Markov Chain
ECRA	Early Component-based Reliability Assessment
EESM	Extended External State Machine
EMS	Emergency Monitoring System
ERC	External Reliability Contract
FMS	Field Monitoring System
FPTA	Failure Propagation and Transformation Analysis
FPTC	Fault Propagation and Transformation Calculus
FSM	Finite State Machine
FTA	Fault Tree Analysis
GSPN	Generalized Stochastic Petri Net
MC	Markov Chain
MDA	Model Driven Architecture
MDD	Model Driven Development
MM	Mathematical Model
MMT	Model to Model Transformation
MTL	Model to Text Language

NFR	Non-Functional Requirement
PCM	Palladio Component Model
PN	Petri Net
PoSM	Port State Machine
PSM	Protocol state machine
OCL	Object Constraint Language
QVT-O	Query View Transformation - Operational
RADL	Rich Architecture Definition Language
RBD	Reliability Block Diagram
SAAM	Software Architecture Analysis Method
SBRA	Scenario-Based Reliability Analysis Approach
SDL	Specification and Description Language
SEFT	State Event Fault Tree
SISO	Single Input Single Output
SM	State Machine
SOA	Service-Oriented Architecture
SPN	Stochastic Petri Net
SPNP	Stochastic Petri Net Package
SRN	Stochastic Reward Net
SvFTAM	Single Version Fault Tolerance Aspect Modeling
SV-FT	Single Version-Fault Tolerance
TMR	Triple Model Redundancy
TPN	Timed Petri Net
UML	Unified Modeling Language
VTS	Vehicle Tracking System

Chapter 1: Introduction

The dependability of a software system is the ability to deliver trusted service and to avoid service failure that is more frequent and more severe than acceptable [1]. Dependability attributes encompass a set of Non-Functional Properties (NFP) such as reliability and availability. Software reliability is defined as the probability that the software system will operate correctly for a period of time, while software availability is the probability that the software system will be operational at a specific time. Fault tolerance plays a crucial role in achieving and attaining dependability requirements. In fact, including fault tolerance in a software system helps dealing with unpredictable situations and ensures that the external behavior of the system remains acceptable during operation. However, adding a fault tolerance mechanism may improve the system reliability, but the effects are non-trivial and depend on the context [2].

The Unified Modeling Language (UML) is a well known modeling language that is used to model software systems during the development phase [3]. It helps to capture the software system's structure, as well as its behavior. UML offers extension mechanisms for defining profiles that extend the language for different domains. For instance, Dependability Analysis and Modeling profile (DAM) [4] is defined to help modelers to capture software system dependability requirements such as reliability and availability. It depicts also dependability threats that affect the system's components as well as error propagation. DAM profile supports modeling redundancy, but it does not fully support fault tolerance architecture modeling. Quantitative and qualitative information can be specified using tagged values.

The Model Driven Development (MDD) is a promising approach for software development that changes the focus from code to models. This change of focus facilitates the analysis of different NFP using formal analysis models obtained by model transformations from the software models. The present work is concerned with the analysis of dependability attributes (such as reliability and availability) using analysis models automatically generated from software models extended with dependability annotations using DAM profile. Another software development paradigm of interest is Component Based Development (CBD) [5], which applies the "divide and conquer" principle to manage system complexity. Each component is a unit of composition that interacts with other components through predefined interfaces. CBD is a reuse-based approach that has an impact on the development time and system dependability attributes. Combining MDD and CBD is an appealing approach to the development of real-time embedded systems, as it reduces the complexity, time and cost. In addition, MDD and CBD help to integrate dependability modeling and analysis during the design phase [6].

Software architecture is the process of designing the global organization of software system by identifying its components and specifying component interactions through interfaces. Software architecture is utilized to reason about the system since it influences its quality attributes [7]. For instance, the reliability analysis of a software system is based on its architecture and helps to identify reliability hot spots [8]. It was shown that including error propagation between the software components in reliability assessments has a significant impact on the accuracy of the analysis results [9] and it helps to select proper fault tolerance mechanism and to place error detection [10].

The Aspect-Oriented Modeling (AOM) approach allows modelers to treat crosscutting concerns separately. Modeling erroneous behavior and error propagation of software components along with normal behavior tends to be complex and hard to read and maintain. AOM is utilized in our work to model component erroneous behavior through the Component Erroneous Behavior Aspect Modeling approach (CeBAM). Therefore, the reliability analysis model will be automatically derived from software models that include components' normal behavior, as well as erroneous behavior.

The compatibility verification between interacting components and the conformance verification of their internal behavior with the corresponding ports protocol behavior are crucial steps for the early identification of unexpected messages between components. In fact, components are not designed to handle properly unexpected messages, and this will negatively impact the system reliability. Since we have selected Stochastic Reward Net (SRN) for reliability analysis model, we also utilize SRN to verify the conformance and compatibility between software components. In this thesis, we present transformation rules for deriving SRN from CeBAM representations that can be analyzed to identify conformance and compatibility issues.

The Fault tolerance mechanism is one of the means for improving the system reliability and availability. Each fault tolerance technique needs to be customized and tailored to the application that is using it. In fact, increasing redundancy by identical replication is a common approach for fault tolerance in hardware. According to [11] such an approach is not applicable to software, which is usually deterministic and thus each replica receives and processes the same data; design diversity needs to be used instead. However, the work in [12], [13] introduces a new thinking in software fault tolerance

based on environmental diversity as opposed to design diversity. We introduce the Single Version Fault Tolerance Aspect Modeling (SvFTAM) approach that provides the software modeler with a quantitative approach to compare different fault tolerance tactics. It captures architectural and behavioral models of single fault tolerance tactics as generic reusable aspects annotated with formal dependability attributes.

The state space explosion is a well known problem for state-based analysis models such as SRN. In our case, as the number of the involved components or services in the critical scenario under analysis grows up, the number of underlying states of the derived SRN model grows exponentially. As a result, the SRN models do not scale up for larger systems. Therefore, we looked for an approach to solve larger models, even at the price of obtaining approximate measures for reliability. In the literature, different techniques are proposed to address this issue; among them is the structure-based reduction technique [14]-[17]. In our approach, we adopted and extended such techniques, integrating them in our automated dependability analysis framework. We provide a decomposition and reduction techniques for the derived SRN model to compute approximate system reliability measures.

1.1 Motivation

In modern software systems, reliability and availability are considered among the most critical NFPs. Therefore, an analysis of such NFPs should be performed in the early stages of the architectural design in order to reduce the cost and time of development and to increase the dependability of the software system. However, generating dependability analysis models during the design phase is challenging due to the gap between the software model and the analysis model. Utilizing the MDD approach allows obtaining

formal analysis models form software models that have been extended with dependability annotations by applying model transformation rules.

The existing software architecture-based reliability prediction suffers from different drawbacks and limitation that impact its applicability and result accuracy. For instance, most of the approaches do not include component erroneous behavior and failure propagation during the reliability analysis process. Indeed, due to the lack of a practical modeling approach, capturing component erroneous behavior along with normal behavior increases the complexity of the software model.

In CBD, interacting components may be developed by different teams; therefore, it is possible to unintentionally violate the interface agreement by sending an unexpected message to the other components. Other kind of unexpected messages are due to component failure. Unexpected messages are not handled properly, which affects the software reliability. In our opinion, performing conformance and compatibility verification for software architecture prior to generating analysis models is a mandatory step. To the best of our knowledge, all existing approaches for dependability analysis assume that the components are compatible with each other and all messages are handled correctly.

Fault tolerance mechanisms are applied to software architecture in order to improve the overall software system reliability. In fact, adding a fault tolerance mechanism may improve the system reliability, but the effects are non-trivial and depend on the context [2]. However, most of the existing architectural-based software reliability assessment approaches do not help the developers to have a practical approach that allows them to

trade off between different fault tolerance styles and select the best style based on quantitative data.

Performing analytical analysis using the SRN model remains challenging, due to the state space size which grows exponentially for large systems with many components. This problem is well known and many approaches have already been proposed to address it, such as in [14], [15], [17]. The software component architecture model can be used to guide the decomposition of the derived SRN model into smaller subnets that can be aggregated. Also, Petri Net (PN) reduction rules from literature were adapted to build an approximate analysis models based on software architecture and SRN composition transformation rules. These models are solved iteratively to compute approximate reliability measures for large systems.

The motivation for the thesis research is: (I) Simplify the task of building dependability models by deriving them automatically from software models; (II) Include error propagation in reliability analysis to get more accurate predictions. (III) Simplify the task of composing normal behavior with erroneous behavior/error propagation by using AOM; (IV) Verify component compatibility and conformance to detect and eliminate bugs before solving the dependability analysis models. (V) Support the selection of the most appropriate fault tolerance styles based on quantitative data; (VI) Use the software architecture to guide the decomposition and reduction of the PN model in order to mitigate the state space explosion for large models.

1.2 Thesis Objectives

The ultimate goal of our research is to design a framework based on standard modeling languages such as UML, that will help developers to evaluate quantitatively

dependability attributes during the design phase. To automate the framework process, we built a transformation chain based on standard model transformation languages (QVT operational and Acceleo), which bridges the gap between the software model and analysis model. The proposed framework integrates the analysis into the development process by generating analysis models and getting results transparently, without involving the designers in the details and complexity of model generation.

Our approach of predicting reliability based on software architecture is taking into consideration component erroneous behavior and error propagation. We believe that including component error propagation in dependability analysis and prediction will help developers to take the right design decisions, such as selecting proper fault tolerance mechanisms, placing error detection and using suitable recovery approaches. The findings of [9], [10] support this belief, showing that error propagation may have a significant impact on reliability prediction. Thus, in our approach, the evaluation of dependability attributes is based on complete components behavior, both normal and erroneous.

In CBD the components have two views: internal and external. The internal view describes the component's private properties realizing the provided services, while the external view shows the public properties in terms of required and provided services. A component's port is an interaction point with other connected components. Provided and required services are associated with the component ports, therefore incoming and outgoing messages can be precisely modeled at the port. We use a behavior state machine to describe the component's internal view and we extend the UML concept of protocol state machines in order to capture all sent and received messages through the ports.

In order to model the dependability threats and to capture the fault origin as well as failure propagation without increasing the model complexity, we employ the aspect-oriented modeling (AOM) approach. More specifically, we separately model the erroneous behavior of each component as an aspect, using the same modeling language and tools as for the normal behavior models. Once both behaviors are modeled, we compose automatically the erroneous behavior aspects with the normal behavior, which is the base model.

Verifying the conformance and compatibility of components helps avoiding unexpected messages, which may negatively affect the reliability of the system. Such verification can be done with the help of the SRN model generated for reliability analysis. More specifically, the automated transformation rules for generating the SRN model are also used for conformance and compatibility verification, performed by studying SRN properties, such as deadlocks and dangling transitions or places.

We have selected the Stochastic Reward Net (SRN) from different existing alternatives, since it allows for modeling complex specifications in a convenient way [18], [19]. The tool used for solving the SRN models, called Stochastic Petri Net Package (SPNP) [20], has a specific input language C-based SRN Programming Language (CSPL), which is generated in a transformation chain developed in this thesis, composed of four automated transformation engines:

1. Aspect-oriented model transformation to compose a component normal behavior with its erroneous behavior aspects;
2. Model-to-model transformation to generate a SRN model from a software model and verify conformance and compatibility;

3. Intermediate model-to-model transformation that builds a CSPL model based on the derived SRN model;
4. Model-to-text transformation to generate SPNP solver input code (CSPL) from the generated CSPL model.

The generated CSPL code is passed to the SPNP solver to obtain the required analysis results. These results will guide the designers in selecting design alternatives that will improve the system dependability.

As mentioned earlier, applying fault tolerance styles does not always improve the reliability of the software system. We aim to provide the developer with a practical approach to compare the reliability effects of different fault tolerance styles based on quantitative data during the design phase. Moreover, we intend to enrich the application of fault tolerance techniques by building libraries of Single Version Fault Tolerance (SV-VT) styles to allow the developer to select a proper style for refactoring the software architecture and evaluating how it affects the overall reliability of the system.

The state space explosion is a well known problem of state-based analysis methods, which makes impossible to analyze large systems analytically, only by simulation. An approach for alleviating this problem is to apply decomposition and reduction techniques to build a set of reduced SRN models that can be solved iteratively using an analytical solver. In the decomposition process we take advantage of the software component architecture to construct a set of SRN submodels for each component. The reduction process uses our SRN composition transformation rules to systematically identify the candidate SRN subnets for reduction. Each service execution path is a candidate subnet for reduction without losing the original behavior.

1.3 Thesis Contributions

The contributions of the thesis are summarized as follows:

1. Develop the Component Erroneous Behavior Aspect Modeling (CeBAM) approach for modeling component erroneous behavior and capturing failure propagation between connected components, based on aspect-oriented modeling techniques in annotated UML and OCL.
2. Define a set of model mappings and compositional rules to derive the SRN model for dependability analysis from annotated software models that conform to the CeBAM approach. Mapping rules show the derivation of SRN subnet for each model element of the behavioral models, while the compositional rules are responsible for composing the derived SRN subnets to build a single SRN model.
3. Develop a compatibility and conformance verification approach to detect unexpected messages between components. The compatibility verification between interacting components and the conformance verification of their internal behavior with the corresponding ports protocol behavior are integrated with the composition of the derived SRN subnets for components.
4. Design and implement a QVT-O and Acceleo based transformation chain to automate the proposed dependability analysis framework. It consists of the following transformation engines:
    i. *CebamWeaver* is a model-to-model in-place transformation engine which composes the component normal behavior with the erroneous behavior developed as aspect models according to the CeBAM approach.
    ii. *sm2srn* is a model-to-model transformation engine that derives and

composes the SRN model as well as automates the process of conformance and compatibility verification.

iii. *srn2cspl* is an intermediate model-to-model transformation engine that derives the CSPL model and adds other SPNP configuration settings.

iv. *cspl2text* is a model-to-text transformation engine based on Acceleo that generates the CSPL executable code.

5. Design the Single Version Fault Tolerance Aspect Modeling (SvFTAM) approach that captures the architectural and behavioral models of single fault tolerance tactics as generic reusable aspects annotated with formal dependability attributes.
6. Provide a decomposition and reduction technique for the derived SRN model to compute approximate system reliability measures.

The following papers are the outcomes of this research:

1. N. A. Mokhayesh Alzahrani and D. C. Petriu, "Modeling Component Erroneous Behavior and Error Propagation for Dependability Analysis," Proceedings of the 16th International System Design Languages Forum Model-driven dependability engineering (SDL 2013), pp. 124–143, Apr. 2013.
2. N. A. Mokhayesh Alzahrani and D. C. Petriu, "Derivation of Stochastic Reward Nets from Component Erroneous Behavior Model for Compatibility and Conformance Verification" Technical Report SCE-13-02, Carleton University, Dept. of Systems and Computer Engineering, 42 pages, May 2013.
3. N. A. Mokhayesh Alzahrani and D. C. Petriu, "Derivation of Stochastic Reward Net for Compatibility and Conformance Verification of Component Erroneous Behavior Model", 19th IEEE Pacific Rim International Symposium on

Dependable Computing (PRDC 2013).

4. Naif A. Mokhayesh Alzahrani and Dorina C. Petriu; “Modeling Fault Tolerance Tactics with Reusable Aspects”; In Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA '15).

1.4 Thesis Content

This thesis is organized as follows: Chapter 2 surveys the fundamental concepts of software dependability and the state of the art related to the research area. Chapter 3 presents a high-level overview of the proposed approach. Chapter 4 presents the CeBAM approach and an illustrative case study that will be used as running example throughout the thesis to explain the transformation chain and associated approaches. Chapter 5 shows the model transformation rules to generate a SRN model from a software model, as well as the compositional rules. It also presents the conformance and compatibility verification approach. Chapters 6 and 7 present the design, implementation, and testing of the transformation chain, as well as samples of the output model and log files that capture the transformation progress and modeling errors. Chapter 8 shows the SvFTAM approach for modeling single version fault tolerance as a reusable aspect. The SRN decomposition and reduction technique is presented in Chapter 9. The last chapter summarizes the contributions and limitations of the thesis and provides a list of future research directions.

Chapter 2: Background and State of the Art

This chapter provides an overview of the background and state of the art related to the thesis research area.

2.1 Fundamental Concepts of Software Dependability

This section presents basic concepts, definitions, and terminologies relevant to software dependability. We rely on a widely accepted taxonomy presented by Laprie et al. [1]. Discussions and comparisons between different dependability definitions and viewpoints can be found in [21]. In the literature, dependability concepts are explained in more depth in [11], [22], [23]. Our objective in this section is to present only a brief introduction to the main concepts of software dependability as a point of reference for the next sections.

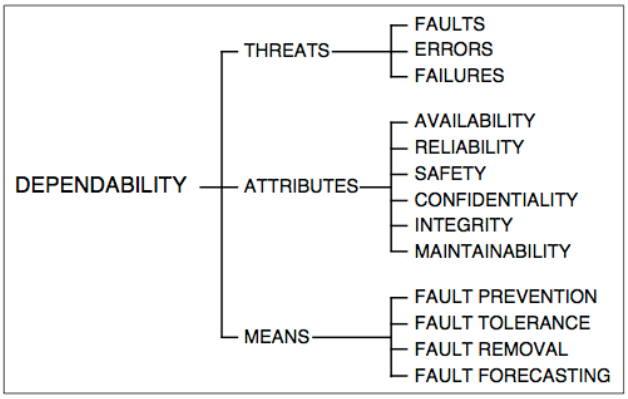


Figure 2.1: The Dependability Tree [1]

Dependability is defined as *"the ability to avoid service failures that are more frequent and more severe than acceptable"* [1]. Figure 2.1 shows the dependability tree that has three components: attributes, threats and means. We notice from this definition that the use of word "acceptable" means that dependability is subjective, since we cannot

develop a system that never fails. Moreover, dependability is defined as an average and it depends on the operating context [11].

2.1.1 Attributes

According to [1] dependability is an integrative concept that has the following basic set of attributes:

- Reliability: continuity of correct service;
- Availability: readiness for correct service;
- Safety: absence of catastrophic consequences on the uses(s) and the environment;
- Integrity: absence of improper system alternations;
- Confidentiality: absence of unauthorized disclosure of information.
- Maintainability: ability to undergo modifications and repairs.

Dependability attributes are Non-Functional Properties (NFP) which help setting the dependability requirements by providing a framework for defining the type of acceptable service failure and its rate [11]. For instance, we quantitatively define reliability and availability requirements in terms of probabilities, which are measurable values and represent a clear target that can be verified in later development phases (e.g., design, testing). In this thesis we are interested in the early prediction of reliability and availability of software systems at the design phase, so we are going to briefly explain these two attributes.

According to [1], reliability is defined as: ”the *probability that the system will operate correctly in a specific operating environment up until time t,*” while availability

is " the *probability that the system will be operational at time t*". Software reliability assessment approaches are classified as follows [24]:

- Black box reliability analysis approach that collects and observes failure data of the software in testing or operation and uses it for reliability estimation;
- Software metric-based reliability approach based on static analysis of the software such as complexity, number of code lines, and developer experience;
- Architecture-based reliability analysis approach using component reliabilities and system structure for the evaluation of the system properties. In fact, this approach is further classified into state-based and path-based approach. The framework developed in this thesis uses this approach and more details will be presented in the next sections.

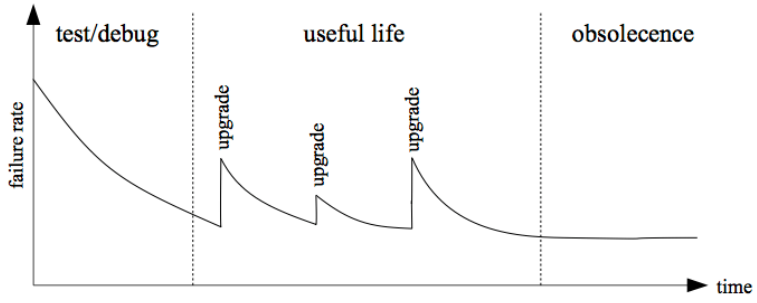


Figure 2.2: A Rough Sketch of Software Failure Rate Over Life Time [24]

The reliabilities of hardware and software are basically similar, but deal with different failure rates during operation. In hardware, once the components are deployed to the operational environment, no further updates or changes are applied to it and the failure rate will increase after a long time of operation (wear-out phase). However, software components can be changed overtime long after deployment. In fact, a software system is always under improvement and/or bug fixing by introducing new versions or

upgrades. As a consequence, the failure rate may be increased after each upgrade and the wear-out phase is not applicable in software systems, as shown in Figure 2.2 [24].

Reliability and availability are closely related concepts and usually mixed with each other. However, they are different and their importance depends on the application context. For instance, an airplane control system must perform failure-free during the flight time; therefore, this kind of system must be highly reliable. On the other hand, telecom system services must be available for customers whenever they request it even if the system fails regularly, but they have a short repair time. Moreover, when the system's reliability is improved, the availability is improved accordingly, but the opposite is not always true [25].

2.1.2 Faults, Error and Failure Propagation

A fault is the adjudged or hypothesized cause of an error [1]. In other words, it is a dormant or latent defect in a software system that may cause an erroneous state if it is activated. Faults could be permanent after activation leading to service failure, or transient which may affect the system state temporally without causing a service failure [11], [22], [26]. Faults are introduced in the system because of incorrect requirements, incorrect design, or coding defects [25]. According to [11], faults can be categorized into three types: degradation, design and Byzantine. Degradation faults are related to hardware and are activated when something brakes. Most software faults are design faults. If fault effects are seen differently by different system components, then they are called Byzantine faults.

An error is a part of the system's state that may lead to a system failure [1]. Multiple faults may create the same error. An error may propagate to produce another

error. Detecting an erroneous state and starting a recovery process prevents further propagation that may cause service failure [11]. According to [25], errors can be categorized into two types: value and timing. For instance, two connected components may exchange unexpected messages due to the incompatibility between provided and required services or to messages sent at inappropriate times.

A system failure is an event that occurs when the delivered service deviates from the correct service [1]. System failures can be further categorized into failure modes such as fail-silent, crash failure, and fail-stop. If the failed component does not deliver incorrect results that may cause severe consequences, then we consider it fail-silent or fail-safe mode. A crash failure mode occurs after a first fail-silent mode and after the component stops working. The main difference between crash failure and fail-stop is that the latter is visible to the rest of the system and detectable, while the former may remain undetected [25].

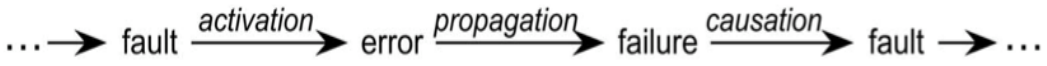


Figure 2.3: The Fundamental Chain of Dependability Threats [1]

The relation between fault, error, and failure is a cause and effect relationship (see Figure 2.3). An activated fault will cause an erroneous state that may propagate to cause another error or failure manifestation. In [25] are presented different examples illustrating the differences between fault, error, and failure and how they are related to each other. This chain of dependability threats between software components affects the overall reliability and availability of the system. As shown in Figure 2.4, if component B uses a provided services of component A, then the manifested failure of component A

propagates to all dependent components, such as component B [1]. Including fault tolerance helps detecting erroneous states and starting the recovery process to avoid service failure.

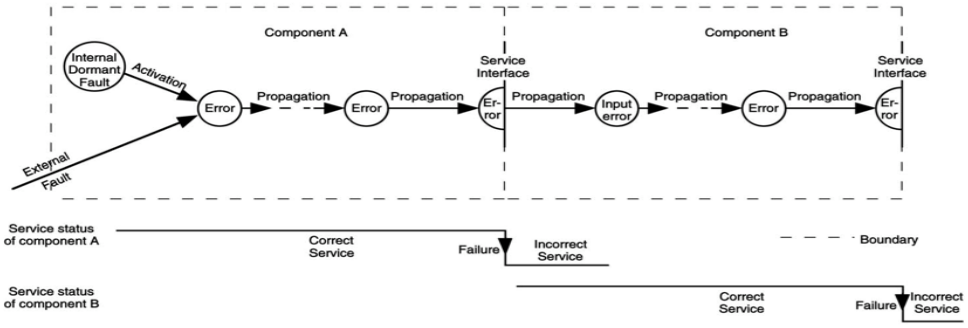


Figure 2.4: Error Propagation [1]

2.1.3 Means to Attain Dependability

Faults are the main issue in dependability. Therefore, in order to improve the dependability of a software system we have to deal with faults during the development and operation phases [11]. Accordingly, means to achieve dependability requirements are classified into: fault prevention, fault removal, fault tolerance, and fault forecasting [1]. This classification can be determined by the development process to which they can be applied. Applying multiple means will positively affect the dependability [21].

Fault prevention aims to develop a system without certain types of faults. From a software perspective, techniques for avoiding faults can be applied in all the development phases. For example, applying a certain design pattern and following a systematic development approach such as COMET [27] helps to avoid certain design faults. During the implementation phase, utilizing correctness by construction approaches also prevents design faults. More details about techniques that can be applied to avoid faults are discussed in [11].

Fault elimination and/or removal complements fault prevention since it identifies existing faults that cannot be avoided during the design and implementation phases. Fault removal can be applied during the system development and operation. Software verification and validation processes are examples of fault removal during development and preventive maintenance techniques [21].

If fault avoidance and removal cannot correct all faults, fault tolerance techniques could help to mitigate some of the remaining faults during operation. Fault tolerance plays a crucial role toward achieving dependability. In fact, including fault tolerance helps dealing with unpredictable situations, ensuring that the external behavior of the system remains acceptable during operation. An objective of this thesis is how to select the best software fault tolerance techniques to improve the overall reliability of the system based on quantitative data. The following sections will discuss it in more details.

Fault forecasting predicts the effects of the faults on the system, helping to decide whether or not the system will meet the dependability requirements [11]. It is a proactive approach since it estimates the present occurrences of faults and applies quantitative techniques to assess dependability attributes such as reliability. In this thesis, we are proposing a framework to derive dependability analysis model from the software model that captures dependability requirements in order to predict the overall system reliability at the design phase. Dependability attributes forecasting reduces cost and development time, helping the developer to take the right decision based on quantitative data.

2.1.4 Dependability Analysis

Dependability analysis is conducted by anticipating all possible faults in the system and by studying the consequences of failure. Deductive methods assume the system's

failure in order to identify which system modes or behavior caused that failure. Examples of such analysis approaches are Fault Tree Analysis (FTA) and Reliability Block Diagram (RBD) [28]. Other analysis approaches are the inductive methods that start by assuming fault activation and then finding out the effects on the system. These approaches help to identify possible failure states. Failure Mode Effects and Criticality Analysis (FMECA) and Hazard and Operability Analysis (HazOp) are example of inductive methods [11].

Dependability modeling techniques can be categorized into combinatorial and state-based approaches. In order to simplify the mathematical solution, combinatorial approaches assume that the failure events of the system's components are independent. In fact, the assumption of independent failure component behavior is not the best approach in complex safety critical applications [29]. Another limitation of combinatorial approaches is the inability to model repair dependencies among system components that share the same repair facility [30]. FTA and FMECA are examples of combinatorial approaches that do not consider time information. A recent survey [31] provides a review of analysis approaches and identifies in-depth its limitations. For instance, building FMECA and FTA depends on the analyst’s skills and captures only a static representation of the system.

FMECA is a forward analysis technique that determines the effects of potential component failures on the system. It utilizes the experience of engineers to identify possible failures of each component in the system and then, elaborates failure semantics and identifies the consequences on the system. This process is recorded in a tabular format [32], [33].

RBD is a success oriented analysis technique used to evaluate the system reliability and availability. It has several blocks connected in parallel or in series configuration. The blocks represent the smallest entities of the system that cannot be further divided. The connection between components depends on the logical interaction and their effect on the system. Each RBD has one input node and one output node and a set of blocks in between connected in parallel or in series. If there is a path between the input and output node, then the system is available. A structure formula can be obtained by identifying all minimal cut sets [34]. SHARPE is an example of tool that can be used to model and solve RBD [35].

Static FTA is a failure-oriented analysis that determines all events that lead to a system failure. It is a backward search technique [11], [28]. It starts by identifying the top event that represents the system failure and then, identifies the immediate failure events. For each identified failure event, another list of caused failure events must be identified. Logical gates such as AND, OR are used to connect these events. FMECA results can be used as input for FTA in order to identify all possible events that lead to system failure. FTA provides a framework for both quantitative and qualitative analysis of the system reliability.

Dynamic Fault Tree (DFT) is an extension of FTA. It was introduced to model functional dependencies, sequence dependencies and spares in fault tolerance systems. New gate types are defined such as Functional Dependency Gate, Spare Gate, Priority AND gate [36], [37].

Component Fault Tree (CFT) [38] and State Event Fault Tree (SEFT) [39] are compositional extensions of static fault tree. A component in CFT is defined according to

the technical component of the system; therefore, CFT overcomes the limitation of reusability of FTA modules. Each component in CFT has an input and output port connected to each other according to the system architecture. CFT is used to describe the failure event of each component. The top failure event is connected to the output port (output propagation), while the basic fault event of the component is connected to the input port if it is initiated by another component failure event (incoming propagation), otherwise will be the same as in classical FTA. SEFT further extends CFT by a probabilistic finite state model. The component internal behavior is described by a deterministic state machine, further translated into deterministic and stochastic Petri Nets (DSPN). The state port along with the standard event-based failure port can be used to examine architectural elements in a specific state such as the output port state.

Component based models such as FTA, DFT and RBD work if failure events are stochastically independent. In fact, this assumption is not always applicable since failure may affect several components. For instance, if two components use the same power source, the failure or the repair process of these components is statistically dependent. Relaxation of this assumption makes the reliability computation hard. As a result, a stochastic process is needed [18], [40]. Markov Chain (MC) [41] and Petri Net (PN) [42] are examples of state based dependability models that can represent failure states of the system and are capable of capturing different kinds of dependencies. The main disadvantage of these techniques --PN and MC-- is the fact that the number of states will increase exponentially for large systems [18], [31].

Several approaches in the literature propose an automatic derivation and synthesis of dependability analysis model from the system model. In [30] are presented algorithms

or converting FTA to equivalent Generalized Stochastic Petri Net (GSPN) and Stochastic Reward Net (SRN). Using a case study, it shows how SRN reduces the complexity of the model specification, but the effort and complexity to solve GSPN and SRN remain the same. Automatic synthesis of DFT from UML is introduced in [43]. It starts by extending UML by defining new stereotypes and tagged values to model fault tolerance systems; then, algorithm for deriving DFT are developed. In [44] a framework to automatically generate FTA from SysML system models is presented; in [45] FTA is generated from an AADL Model. A modeling approach focused on reusability and automatic FTA generation from a UML profile that captures fault propagation and containments is presented in [46]. Several approaches in the literature derive state based analysis models (SPN, GSPN, and SRN) from UML models such as [47]-[51].

A comprehensive comparison between dependability analysis models and their limitations are presented in [31]. From these approaches, we selected Stochastic Reward Net (SRN) [18], [52] which extends GSPN. Marking dependency is the main feature of SRN which allows modeling complex specifications in a more convenient way than GSPN. One of the objectives of this thesis is to automate the derivation of SRN models from software models to predict the reliability/availability of the systems.

2.2 Software Architecture-Based Reliability Analysis

A component is a basic independent entity and workhorse abstraction of software architecture. With clearly defined interfaces (provided/required) its internal behavior can be designed and developed independently. On the other hand, software architecture defines the global organization of a software system including how components interact with each other [7]. In order to predict the reliability of an application, the information

about its architecture must be considered. Architecture-based reliability models help to understand how system reliability depends on the component's interactions and on individual component's reliability.

In [53] are presented different qualitative and heuristic approaches for evaluating software architecture. One of these approaches is Scenario-based Software Architecture Analysis Method (SAAM) that guides the inspection of software architecture to find potential issues such as requirement conflicts. SAAM approach has been extended and refined to support specific goals in architecture evaluation. In [54], [55] many of the architecture evaluation approaches in the literature are surveyed. Our approach focuses only on quantitative based techniques to predict and evaluate software architecture reliability.

According to [56], architectural-based software reliability approaches are classified into: additive, path-based, and state-based. Additive models are mathematical models meant to compute failure intensity of the system, which is calculated from the component failure intensities. Since the software architecture is not explicitly examined in additive models, we do not cover it in this thesis; we just focus on path-based and state-based approaches. Surveys such as [8], [55]-[60] review and compare many approaches in the literature. In the next section we will briefly present only the most recent approaches related to our solution.

2.2.1 Path-based Analysis

In these models the system reliability is computed by examining possible execution paths. First, path reliability is computed by multiplying the reliability of each component

involved in that path. Second, the average of path reliability of overall paths is calculated to get the system reliability [56], [57].

In [61] a Bayesian framework integrated with UML is proposed. Use cases and sequence diagrams are annotated for the purpose of reliability assessment. For instance, use case diagrams show the probability that certain actors request certain system functionality, and sequence diagrams show the duration of busy periods. The system reliability is computed by using mathematical equations based on diagram annotations. Moreover, prior information about the component's failure probability must be available. This approach is extended in [62] to include a deployment diagram that is annotated with the failure probability of connectors. Including operational profiles in the system reliability prediction process by annotating UML diagrams makes this approach more user friendly.

A technique called Scenario-Based Reliability Analysis Approach (SBRA) to predict reliability for component-based software is proposed in [63]. Component Dependency Graph (CDG) is a probabilistic directed graph constructed for each scenario. CDG represents the dependency between components in possible execution paths and it is derived from a sequence diagram. Each node in CDG represents a component with its reliability and average execution time. Transitions capture the execution path from one component to another; transition attributes are name, transition reliability and probability. Based on the constructed CDG, an algorithm is used to analyze the reliability of the application as a function of its components. It has the same limitations as [61], [62], since it requires to know component reliability in advance and it does not consider failure dependencies between components [57], [58].

A bottom-up framework for estimating system architecture reliability using different component composition is presented in [64]. Five basic composition mechanisms are defined and mathematical formulas are used to compute reliability estimation for each composition mechanism. An iterative process estimates the overall application reliability for a given system that uses a combination of component composition mechanisms.

2.2.2 State-based Analysis

A state-based analysis approach has several advantages, such as considering different failure modes and the impact of infinite paths resulting from the presence of loops. The architecture of the software can be modeled as Discrete-Time Markov Chain (DTMC), Continues-Time Markov Chain (CTMC), or semi Markov Process [8]. The control flow between components is considered in reliability prediction computation. Control transfer is modeled as a Markov process, service Finite State Machine (FSM) usage profile which makes the assumption that the current behavior of a component is independent of its past behavior. Moreover, most of the approaches assume that the components fail independently.

The reliability prediction approach in [65] allows architects to predict component reliability through compositional analysis of the usage profile and environment component reliability. This approach is based on the Rich Architecture Definition Language (RADL). For each component, there are three types of finite state machines (FSM): provided gate FSM, required gate FSM, and, for each provided service, a service FSM usage profile. The overall reliability is computed analytically, by using a provided gate FSM augmented by a usage profile and the reliability of provided services. Reussner

et al. [65] assume that a service call is modeled as a Markov process. This means that the service failure is independent; therefore, error propagation is not included in this approach.

Hong et al. [66] classify reliability analysis into black box and white box. The black box analysis builds failure behavior based on runtime information, while the white box utilizes software internal information at early stages of development to predict the reliability. Hong et al. developed a white box reliability prediction methodology that uses UML models. This method has several steps. First, a case model is used to identify the operational profile. Second, an activity diagram is developed for each use case and a domain expert assigns activity transition probabilities. Third, the utilization of each component is identified by counting the busy periods in a sequence diagram. The next three steps use the information provided in each UML artifact to calculate the component level reliability, activity level reliability and finally derive the system level reliability. Absence of automatic recovery is assumed. This method helps the developer to identify early the critical component quantitatively.

Bondavalli et al. [47] propose an automated dependability analysis approach based on UML diagrams. Structural UML views such as a deployment diagram is enriched by failure and repair characteristics using an UML profile. The software views are transformed into Timed Petri Net (TPN) for dependability analysis. The transformation takes place in two phases: a) transformation rules are applied to extract the required information from UML structural views in order to build an Intermediate Model (IM); b) transform from IM to TPN using another set of rules. A similar approach using graph transformation techniques to generate IM is presented in [67]. However, we believe that

having IM increases the transformation complexity and it could be done in one phase by generating TPN directly from UML structural view, as we will show in the solution developed in the thesis.

In [68] a user-oriented software reliability model is proposed. In this model, a composite structure of a system is depicted as a control flow graph with a single entry and a single exit. The control transfer between components is described by DTMC. The system reliability is computed as a function of the deterministic properties of the system structure and stochastic properties of the system components, utilization and failures.

Most of the approaches in the reliability prediction assume the reliability of a single component and focus on the system level of reliability. The work of [69] presents an approach to predict the reliability of a single component. It can be used with other approaches by computing the component reliability instead of assuming it. Markov model is used as a stochastic reliability model, based on the dynamic behavior of the component. New failure states, failure transitions, and recovery transitions are added to the model. Hidden Markov Models (HMM) are used to obtain behavioral transition probability. The component reliability is computed by solving the constructed Markov chain model.

2.2.3 Limitations and Challenges

Assessing software system dependability during the early design phase helps designers to take the right design decisions and tread-offs between different architectural styles. However, the information required to predict the system dependability attributes such as reliability, is not complete and not clearly identified. Thus, prediction and

assessment become more challenging. Several surveys review the existing approaches for predicting software system reliability from its architecture [8], [59], [70]-[72].

In [54]several limitations of architecture-based reliability prediction approaches are identified. For instance, using MC for modeling software architecture is not the best choice because the execution of some applications depends on history information, which is not supported by MC. Moreover, in order to simplify the mathematical solution, approaches that use MC assume that the failures of the components occur independently. Another limitation is that the concurrent execution behavior of a system component is not considered, assuming that the components are executed sequentially. In order to overcome this limitation, some approaches use high-level specification mechanisms such as GSPN and SRN.

Reliability ingredients are introduced in [71]. The thirteen ingredients (parameters) can be categorized/classified into: failure information, operational profile, and recovery information. Complete component behavior can be extracted from architectural models, which include functional and non-functional properties. This behavior is the source of constructing the failure model. For instance, failure free behavior includes critical services which help to identify failure severity and failure impact of these critical services. In contrast, failure probabilities cannot be derived from architectural models since they depend on other factors, such as deployment. The work in [9], [10] shows more accurate reliability prediction results when failure behavior is included. A few state-based approaches such as [49], [73] consider failure behavior and failure propagation in reliability prediction. In fact most of the surveyed approaches do not include complete

component erroneous behavior and failure propagation due to complexity and lack of practical modeling approach.

Operational profiles that describes the system usage is the second category of reliability ingredients [71]. It is necessary to estimate the effects of failures. In order to define operational information about the execution frequency of system services, the probability of user input, and operational context are required. Musa [74] provides a systematic approach to define the operational profile. The work of [75] explicitly models usage profile, but most of other approaches do not consider it.

Determining recovery information, such as the likelihood to recover and time to recovery are other challenges, since they cannot be determined from the architectural model. For example, in case of automated recovery mechanisms, the time it takes to bring the system back to the correct state depends on other factors, such as resource availability [71]. On the other hand, the recovery mechanisms and recovery process can be specified in the architectural model. For instance, including fault tolerance style in the architectural model will precisely capture the system behavior once the erroneous state is detected. In order to simplify the mathematical calculation, most of the reviewed path-based approaches assume that the system has no recovery process.

The work in [8] summarizes the limitations of the architecture-based software reliability analysis approaches. The identified limitations are categorized into: modeling limitations, analysis limitations, parameter estimation limitations, validation limitations, and optimization limitations. For each category the author suggests some possible modeling techniques, tools, and analysis techniques to overcome the identified limitations. For instance, some approaches assume sequential execution. This means that

only one component is running at a time. In fact, an application developed using object-oriented programming is executed concurrently. SRN [30] can be used to represent the architecture of concurrent applications. Most of the approaches assume that the components fail independently therefore, error propagation between components is not considered.

In [72] are listed eight dependability concerns that should be captured in software models in order to derive a dependability analysis model using the model driven development approach (MDD). Among these concerns is modeling components dependencies that allows capturing failure propagation between components. This means that the assumption of independent component's failure is no longer valid. Moreover, a component may have different incorrect behaviors according to different kinds of threats; so the complete component failure model must capture a different kind of fault, error and failure modes. Moreover, in order to automatically derive dependability models that describe fault tolerance details, a proper definition and modeling of fault tolerance structure is needed. For instance, in the triple model redundancy (TMR) the fault tolerance structure, the roles of voter and spare should be defined and assigned to components at UML level.

Most of the existing approaches have common limitations, such as their lack of scalability and ignoring failure propagation. In state-based approaches, the number of states in the analysis model exponentially increases with the number of software components. Most of the reviewed approaches use small case studies for the approach verification, but they do not test their approach using industrial applications. Appropriate decomposition and aggregation strategies would help to solve state space explosion of the

analysis model. Another limitation is the fact that some approaches ignore the deployment environment and its failure behavior. Actually, hardware and network failure impact the software operation. Ignoring the erroneous behavior of the deployment environment and the way it propagates to software has a negative impact on the accuracy of the reliability analysis results. Additionally, some approaches assume that the reliability of individual components is available and fixed regardless of the deployment environments [70].

To the best of our knowledge, the existing state-based approaches focus on deriving and building analysis models, but do not adequately address how the computed reliability and availability results can be interpreted and reflected in the software model. Feedback is not explicitly addressed in the literature to show how analysis results could be translated and used to improve the software model. Verifying components' compatibility is important to avoid any unexpected messages that could affect the reliability of the system. Most of the surveyed works assume that the components are compatible and all messages will be handled correctly, which is not always the case if different components are developed by different teams.

2.3 Failure Behavior and Propagation Analysis Approaches

The previous sections present a list of selected approaches that focus on predicting and evaluating the system reliability assuming that the components fail independently. It is also assumed that if one component fails, then the whole system will fail. Indeed, these assumptions are not reflecting the actual behavior of software systems. One of our objectives is to include erroneous behavior and error propagation in the analysis model; therefore, the following subsections will focus on surveying approaches that share the

same goal. Several recent surveys collect and compare proposed approaches in the literature, such as [31], [55], [58], [59].

2.3.1 Analytical Approaches

In [76] the error propagation is studied as an architectural attribute. This analytical approach is based on the architectural specification and helps to estimate the error propagation behavior of architecture. Abdelmoez et al. [76] introduce a set of formulas for estimating error propagation between components. Initially a formula for error propagation probability between two connected components is defined. In a series of mathematical derivations, a formula of estimating error propagation probability is derived and validated in a case study. The analytical results are validated by an empirical approach based on fault injection experiments. In our opinion, even after validating the derived formula for estimating error propagation from architectural specification, it is still hard to utilize it for large applications.

An approach to examine error propagation and how it affects the software reliability is proposed in [77]. A tool called Propagation Analysis Environment (PROPANE) is developed to experiment with injecting faults into running software and to analyze their effects. The challenges of error propagation between functions in a software system by comparing and contrasting different proposed error propagation analysis approaches is presented in [78]. The Interface Propagation Analysis approach (IPA) is a fault-injection based technique developed in [79] to observe and analyze how the failure propagates between components.

Failure propagation is analyzed using Architectural Service Routes (ASRs) in [80]. ASRs are a sequence of components that are connected through interfaces. In fact, the

proposed technique views the system's architecture as a set of ASRs. According to this view, a set of architectural attributes (e.g., number of ASRs, shortest and longest ASR) are used to compute failure propagation and the way it affects the reliability of Component Based Software System (CBSS). This approach is a path-based approach that proposes a set of mathematical formulas to compute the reliability of CBSS, considering failure propagation. However, it does not consider the origin of the error and ignores the component's internal behavior. In [81] the ASR idea is used to determine separately the effects of each kind of failure (i.e. silent, performance, erratic), integrating them in the analysis to get more accurate reliability prediction results. Automating these approaches and utilizing them in industry level applications may need extra efforts. In our proposed approach, we consider the failure propagation, a complete internal behavior of the component (normal and erroneous), and all messages that pass through the component interface.

The works in [9], [82] are among the first path-based approaches that show the impact of including error propagation in the reliability computation. They extend Bayesian reliability prediction of the component-based systems [61] by introducing error propagation probability into the model. Network failure probability is considered as well, but the availability of deployment hardware is ignored. The error propagation model proposed in [83] is adopted to extend the Bayesian reliability prediction model. Actually, [76] follows the same approach to derive a set of mathematical formulas to compute the system reliability. In extending the Bayesian model, the independences assumption is removed in order to introduce the effect of failure raised in one component and then propagated to other dependent components. An interesting part of this work is the use of

UML artifacts with annotated parameters. For instance, in the use case diagram two parameters are used: probability that an actor *i* interacts with the system and the probability of actor *i* using a specific use case. More annotated parameters are added in the sequence diagram and deployment diagram. Early Component-based Reliability Assessment (ECRA) is a tool developed early in [82] and extended by adding error propagation into the reliability prediction. The authors compare the results of two scenarios, with and without error propagation, to find differences in the system reliability prediction results. The conclusion is that including error propagation gives more accurate results. This work inspired some aspects of our proposed approach.

Cortellessa et al. [10] propose a modeling approach to analyze the impact of error propagation on the reliability prediction of a component based system. The paper claims that the approach is providing useful support for placing error detection and recovery mechanisms and helps to focus on the critical component(s) of the system. However, fault tolerance is not considered in this approach. In the same manner as the previous approaches described in this section, the analytical expressions for reliability computation are derived with respect to error propagation and failure probability. The authors compare their approach with Popic et al. [9], criticizing the main assumption from [9] that each component's failure causes the system to fail, and at the same time propagates it to other components. In fact, some component failure may affect only a subset of the system services, while other services may keep running. It depends on the software architecture design whether the component coupling and dependencies are reduced or not. Although fault tolerance is not considered in [10], the results of the case study show the importance of including failure propagation in reliability analysis.

Failure Propagation Transformation Notation (FPTN) is a simple and modular notation technique for the specification of failure behavior of components, which supports qualitative evaluation. The work in [84] includes the failure propagation in safety analysis. It integrates failure analysis with probabilistic model checking using the PRISM tool. Failure Propagation and Transformation Calculus (FPTC) is a technique used for automatically calculating the failure behavior of the whole system from the failure behavior of its individual components. In FPTC, each component is analyzed separately with the consideration of the response of potential failure stimuli. The work in [85] integrates FPTC to the CHESS project by developing a plug-in, called CHESS-FPTC, as a part of CHESS tool-set [49]. A comparative study in [86] compares different model-driven safety evaluation techniques such as FMEA, FTPN,CFT and SETF with AADL Error annex.

Cortellessa and Potena [87] propose a path-based error propagation analysis for Service-Oriented Architecture (SOA). They prove that failure propagation is a key factor for the trustworthiness of the reliability prediction in the SOA context [87]. A new error classification and failure modes are introduced in [88]. It proposes a framework to enable compositional reasoning of error models in CBSS, with focus on reliability and timing.

2.3.2 Model-Based Approaches

A development methodology based on the MDA approach for component based systems that supports multiple analysis techniques during the design is proposed in [49]. CHESS ML is a modeling language defined on a subset of UML, SysML and MARTE. A system modeled with CHESS ML is transformed into Intermediate Dependability Model (IDM) [89]. IDM captured failure modes and errors propagation between components.

The concept of “separation of concerns” is used in order to provides the system designers with multiple views, such as a functional, extraFunctional and analysis view. For instance, in the analysis view, the error model allows the analyst to model just the components’ erroneous behavior and the failure propagation between components. In a second transformation phase using QVT operational [90] an analysis model is generated. This methodology supports multiple analysis models, such as FTA and GSPN. Dependability analysis is performed using GSPN since it shows the system in different states with respect to time and it captures complex interaction between components. In fact, for the same reason we will use state-based analysis model in our proposed approach. With regard to the fault tolerance, IDM captures only redundancy while ignoring behavior. This approach is one of the inspiring approaches for our proposed technique, but we have fundamental differences such as we use UML+DAM profile to model the system and we generate the analysis model directly without an intermediate model. Moreover, we consider that the internal behavior of the component should identify the origin of the faults in the component (e.g., the state in which a fault occurs) and we model port behavior to capture all messages that pass through the component's ports. To model the erroneous behavior in a practical way and reduce the model's complexity from the user’s point of view, we use the Aspect-Oriented Modeling approach (AOM).

Pham et al. [91] propose a reliability model for CBS which includes error propagation. The assumption of independent component failure is relaxed in order to model error propagation. Moreover, to model fault tolerance, concurrent execution of components is assumed. The UML activity diagram is extended to describe the

component reliability specifications and the UML component diagram is used to describe the system architecture. For each component, a set of extended activity diagrams describe the component's provided services behavior. This approach is a scenario-based approach; therefore, a collection of activity diagrams that realize a use case is transformed into a Markov model to get reliability prediction results. This approach treats each provided service separately, since it assumes concurrent execution, but in reality a component in the running system may have only one component instance [7] that handles incoming messages sequentially. In our proposed approach we describe the internal component's behavior (normal and erroneous) using UML state machines and at any point in time the component is only in one state.

A framework for modeling system dependability using AADL is proposed in [73]. AADL is extended by the Error Model Annex to describe dependability characteristics. The error model has two levels of description: error model type and error model implementation. Declaring component erroneous states, occurrences and propagation is defined by the error model type, while the error model implementation describes the component's erroneous behavior. Both behaviors (normal and erroneous) are transformed to GSPN for analysis. In the transformation rules, the authors introduce a dependency subnet to model the error propagation between components. In fact, this approach is fundamentally similar to the CHESS approach [49]. For instance, in the dependability analysis the component internal behavior is molded as one state only (healthy state). Another similarity is the modeling of the component's erroneous behavior separately from the normal behavior. (In our opinion, is better to model them together). Including component port behavior and identifying the source of error (which we try to capture in

our proposed approach) are not considered in the AADL [73] and CHESS [49] approaches.

The Palladio Component Model (PCM) is using UML-like modeling notation(s) designed originally for performance modeling and recently extended to support reliability modeling [75], [92]. With a good tool support, the component service behavior model describes the provided services behavior and it is similar to UML activity diagram. In this model, two types of activities are defined: call and internal. Call activity represents a remote call for a provided service of connected components, while local execution is modeled as an internal activity. This approach integrates two important architectural aspects: usage profile and execution environment. The usage model, provided by the domain expert, consists of a set of usage scenarios that describe different usage classes. In this model, the domain expert sets the probability for each service input parameter and then, the parameter dependencies are solved based on the work of [93]. The deployment model shows the allocation of each system component and captures the failure model of hardware and the network connection which is included in the reliability prediction. The PCM model is transformed into Discrete-Time Markov Chain (DTMC) after determining all possible physical states (PSS). The reliability results for each PSS are aggregated to get an overall system reliability. In our opinion, this approach seems comprehensive since it includes the usage profile and deployment environment in the analysis, but it is still focused on provided services without identifying the origin of the fault and ignoring detailed internal component behavior. A similar approach that uses PCM to predict software system reliability considering fault tolerance in software product lines is presented in [2].

Component Fault Trees (CFTs) is an approach that provides better support for hierarchical decomposition [38]. The main feature of CFT is modularization, as every component in the system is represented by a single CFT component. For each component a static fault tree describes its fault events and internal failure is propagated out to the other components through the output port. The input port is associated with each component to capture the incoming failures propagated from the other connected component(s). For identifying possible failure propagation between components, dependencies are examined. In [94] it is shown the application of the model driven development principles to an accepted and well established safety engineering approach (i.e., fault tree). A new UML profile is defined for CFT to model a fault tree system. The software system architecture is described by Architecture DSL (ArchDSL). Since different modeling languages are used to describe the same system, language integration is addressed in this paper. Using refactoring and harmonization, an integration approach is introduced to integrate ArchDSL and CFT DSML. In addition, CFT is quantitatively used to analyze failure propagation throughout the system. For instance, [95] propose an algorithm to automatically generate CFT from a software component diagram that is annotated with dependencies and failure modes.

The work in [48] derives manually static fault tree and GSPN from a software model developed using UML+MARTE+DAM profile. The same profile is used in [96] to derive DFT from a software model. These works show the benefits of using MARTE and DAM profiles in modeling NFP properties and getting the analysis models for reliability, availability and performance. Error propagation from hardware to software is considered

in availability prediction, but failure propagation between software components is not captured.

The next chapter focuses on the proposed approach that includes the failure propagation between software components by capturing dependability threat chains between connected components along with normal behavior. Our approach follows a state-based approach and we aim to automate the process of deriving the analysis model from the software model to make it more practical.

2.4 Comparison of Selected Architecture-Based Reliability Analysis Approaches

A large number of approaches are proposed to analyze the system reliability from its architecture. Several surveys review and compare them [31], [55]-[59]. As mentioned earlier, one of our objectives in this thesis is to reduce the gap between the analysis model and the software model. In fact, the automatic derivation of analysis model from the software model will allow developers to transparently asses the system's reliability and tread-off between the design alternatives, without knowing the details of how to build or solve the analysis models. Another objective of our proposed approach is to model the components' erroneous behavior and failure propagation, and to include them in reliability analysis. As a result, we select some approaches that share our objectives and compare them to our proposed approach. In the comparison criteria, we use some limitations and challenges of the architectural-based reliability prediction that were identified in [8], [71]. Table 2.1 summarizes the comparison criteria.

In some cases we use (#) notation to illustrate that the proposed approach does not adequately and explicitly cover a certain criterion.

Table 2.1: Comparison Criteria

Comparison criterion	Explanation
Language	What is the modeling and annotation used in architecture description? i.e. UML, AADL.. etc.
Architectural viewpoint	What is the required architectural view to predict reliability and availability?
Analysis model	What kind of model is used to perform reliability analysis?
Automation	Does the proposed approach automate the generating analysis model from the software model and is it supported by tool?
Feedback and traceability	Is the feedback and tractability for analysis results interpretation explicitly considered?
Scalability	Is the approach scaled for industrial application?
Failure propagation	Does the approach capture and include failure propagation in the reliability analysis?
Operational profile	Does the analysis method include operational profile (execution frequency and user input) in the reliability prediction?
Key system scenario	If is does not include operational profile, does it focus on a key system scenario?
(Normal + erroneous) behavior	Is he complete normal behavior captured along with erroneous behavior to identify fault origin?
Recovery information	Does the analysis method support system recovery or is it assumed that the system is unrecoverable?
Deployment environment	Is the hardware and network failure included in the reliability prediction calculation?
Special expertise needed for modeling and analysis	In order to use the proposed approach by the modeler, is it required to learn a new annotation or skill?
Development process inclusion	Is it easy and practical to include the proposed approach into the software development process?

Table 2.2: Selected Approaches Comparison

Approaches		Language	Architectural viewpoint	Analysis model	Automation	Feedback and traceability	Scalability	Operational profile	Key system scenario	(Normal + erroneous) behavior	Recovery information	Deployment environment	Special expertise needed	Development process inclusion
Path-Based	W.Abdelmoez:2004	Semi UML	Architectural model	MM	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
	Mohamed, A:2008	UML	Architectural model	MM	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗
	Popic:2005	UML	Use case, sequence, and deployment diagrams	MM	✗	✗	#	✓	✓	✗	✗	✓	✓	#
	Cortellessa:2007	UML	Architectural model	MM	✗	✗	#	✓	✗	✗	✗	✗	✓	✗
	X Ge:2009	----	Architectural model	FPTA	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
	H. Aysan:2008	----	Architectural model	MM	✗	✗	✗	✗	✗	#	✓	✗	✓	✗
State-Based	Montecchi:2011	CHESS-ML	Architectural model	GSPN	✓	✓	#	✗	✓	#	✓	✓	✓	✓
	Pham:2012	UML	Activity diagram and component diagram	MC	✗	#	✗	✓	✓	✗	✓	✗	✓	#
	Ana_Elens:2007	AADL	AADL architecture model and AADL error model	GSPN	✗	#	#	✗	✗	#	✓	✗	✓	✓
	Brosch:2011	Semi UML	Usage, Architectural, service behavior, and deployment models	MC	✓	#	#	✓	✓	✗	✗	✓	✓	✓
	L. Berardinelli:2009	UML	Use case, state machine, sequence, and deployment diagrams	GSPN	✗	#	#	✗	✓	✗	#	✓	✓	✓
	Proposed approach	UML	(Behavioral + Protocol) state machine and software architecture	SRN	✓	#	✓	#	✓	✓	✓	✓	#	✓

2.5 Software Fault Tolerance

Fault tolerance is defined in [1] as follows: "*fault tolerance means to avoid service failure in the presence of faults*". Applying fault tolerance helps improving the overall system dependability, since it deals with unpredictable situation and ensures that the external behavior of the system remains acceptable during operation [11]. In [19], [97], [98] some fault tolerance mechanisms are modeled in GSPN and SRN, and their effect on improving the overall reliability of the software system is shown.

Increasing redundancy by replication (identical replication) is a common approach for fault tolerance in hardware. In fact, this approach is not applicable in software since all software faults are design faults; as a result design diversity needs to be used [21]. Moreover, two conditions are identified in [11] that must be violated in order to make the replication useful in software fault tolerance:

- Software is deterministic;
- Each replica receives the same data.

If we pass the same input data to each identical replica, each software copy will handle the data in the same way, which is not useful since failure behavior will be also identical. Moreover, if one replica fails, then all of them will fail following the same behavior, because the software is deterministic.

All software fault tolerance techniques can be divided into two groups. The first group is design diversity, which has different replicas (different implementation versions) developed from the same specifications by different teams, different programming languages, and/or different development tools. An example of this approach is the N-version system recovery block. The second group is data diversity, which uses identical

replication, but run each replica with different data generated from the original data by a re-expression from the original data. Examples of data diversity mechanisms are the retry block and N-copy programming [11]. In [99] it is introduced temporal redundancy, which repeats the execution using the same failed software and hardware resource. Somehow this is similar to the backward recovery.

The surveys [100]-[103] review the existing software fault tolerance techniques and categorize them as sequential techniques, independent concurrent systems, and competitive concurrent systems. Moreover, [99] presents a comprehensive list of software fault tolerance techniques and implementations. The classification is based on data diversity and design diversity. In [19] a list of software fault tolerance techniques is presented, but the focus is more on analysis methods and how fault tolerance impacts dependability. A single-version and multi-version fault tolerance techniques are presented by [104]. Targeted fault tolerance techniques for safety-critical applications are presented in [11], such as watchdog timer, application isolation, safety kernel and execution time checking. These techniques are customized since they identify the fault class of interest and then tailor fault tolerance phases around these faults. Software fault tolerance patterns are presented in [25], according to the fault tolerance phases. Some of the patterns need human interaction, while other can be automated.

Utilizing the AOM approach to refactor the software architecture by adding fault tolerance techniques has been suggested by several authors. An approach for modeling and integrating AOM into CBD is presented in [105]. The author also illustrates how AOM can be used to model component dependability aspects separately, by modeling a template for fault tolerance that provides error detection and recovery services. This

template can be customized and composed with base models through a weaving process. A similar approach presented in [106], [107] uses AOM to construct and build fault tolerance systems. New notations are introduced to capture dependability aspects. The author created a library of fault tolerance mechanisms along with its dependability analysis model template. A model weaver is developed to integrate the fault tolerance aspect with the base software model, as well as to perform the model analysis. Moreover, in the AADL paradigm, AOM is also utilized to model the dependability aspect of the component architecture. For instance, a fault tolerance N-Version programming is designed in [108] as an aspect model at the architectural level, which shows how to integrate the AADL component model.

Studying and analyzing the impacts of applying fault tolerance styles is presented in [109]. The authors propose a framework for selecting a fault tolerance mechanism in a specific context. First, they define an abstract fault tolerance style that includes some quality attributes that can be specified at the time of using the selected style. The quality attributes verification is done using model checking. Another study [110] considers how fault tolerance tactics can be implemented in a well known architecture pattern. Moreover, the author investigates in an empirical study the importance of having information about the effects on the architectural design pattern of fault tolerance tactics. The results from this study recommend that the information about the impact of integrating fault tolerance tactics into architectural patterns must be available to the designers, to allow them to select the best fault tolerance tactics. A model driven framework is created in [111] to be used in the development and testing of fault tolerance systems. The authors claim that the framework reduces the gap between the existing

modeling patterns and the practical application of fault tolerance. They propose a N-version programming techniques to illustrate the usage for the proposed framework.

Each fault tolerance technique needs to be customized and tailored to a particular application. One of the objectives of this thesis is to generalize the application of the software fault tolerance architecture by utilizing the AOM approach, including the failure propagation behavior. Moreover, part of the proposed approach is to predict the effects of selected fault tolerance mechanisms in terms of reliability and availability. Indeed this would help the designer to make the right decisions, based on quantitative data and tread-offs between different fault tolerance styles.

2.6 Stochastic Reward Net

According to [112], Petri net is a mathematical modeling language that is widely accepted for the modeling and analysis of distributed systems. It can be represented graphically as a bipartite directed graph with two types of labels: places and transitions. A transition is fired if all of its input places have enough tokens equals to the arc multiplicity. Once the transition is enabled, tokens will be removed from their input places and new tokens will be added to the output places according to their arcs multiplicity. The Petri net state is represented by the so-called “marking”, which indicates the number of tokens in each place at any point in time; the state keeps changing after each transition firing.

Different extensions of PN were defined to model and solve specific applications, for instance Stochastic Petri Net (SPN), Generalized Stochastic Petri Net (GSPN), and Stochastic Reward Net (SRN). In SPN each transition has a firing time, whereas GSPN extends this by introducing immediate transitions. In addition, the marking can be

categorized into tangible and vanishing states. If at least one immediate transition is enabled by a marking, the state is vanishing, otherwise it is tangible. Furthermore, inhibitor arcs are a special kind of arc defined in GSPN that are used to reverse the logic of input places. For example, a transition will not be enabled if one of its input places (connected by an inhibitor arc) has a token. In other words, the absence of a token in an input place connected by an inhibitor arc will enable the transition, rather than its presence.

SPN has been widely used in modeling and evaluating system dependability attributes. It shows the system in different states with respect to time. This helps to captures the complex interaction between components such as failure propagation.

SRN extends GSPN by introducing new features, such as reward rate and marking dependency. In SRN tangible marking can be associated with a reward rate. On the other hand, marking dependency is an essential characteristic of SRN, which allows for defining newly introduced parameters (such as arc multiplicity, transition guard and firing rate) as a function of the number of tokens in certain places [18]. This feature allows for modeling complex specification in a more convenient way and simplifies the graphical model.

SPNP [113] is the tool for modeling and solving SRN models. It has a graphical interface, as well as a special specification language called C-based SPN language (CSPL). The graphical interface is useful only for small models. CSPL is an extension of the C programming language for describing SRN models. In our case, we rely on the CSPL that is generated during the transformation process. Moreover, SPNP solves the SRN for transient or steady state. It also allows for defining custom measures in terms of

Marking-dependent reward rates. As mentioned earlier, in this thesis we adopt SRN as state based dependability analysis model that is derived automatically from the annotated software models.

Chapter 3: Overview of Proposed Dependability Analysis Framework

Software architecture provides a set of high-level abstractions for representing structure, behavior, and Non-Functional Properties (NFP) of the software. According to [114], component structure, dynamic behavior and allocation are the main categories of architectural constructs. Component Based Development (CBD) applies the "divide and conquer" principle to manage system complexity. It produces an abstract model that shows the static structure of the system in which each component is assigned a specific functionality. In addition, each component is a unit of composition that interacts with other components through predefined interfaces.

Using the software architecture as a basis for the early reasoning and evaluation of the system's NFP helps to reduce the cost and produce software conforming to specifications [115]. Software dependability is one of the examples of NFP that need to be evaluated during the design phase. This, in turn, encompasses a set of attributes: reliability, availability, maintainability, integrity, and safety [1]. The quantitative results of these analyses will support the developer in making the right decisions for building dependable systems. Although different approaches were proposed in the literature to address the reliability and availability modeling [58], [59], many existing methods do not adequately consider error propagation in predicting system reliability [88].

Model Driven Development (MDD) changes the focus from code to models. This focus on models facilitates the analysis of the different NFPs by automating the derivation of formal analysis models from the annotated software model using model transformations techniques [116] [4]. In fact, combining MDD and CBD is an appealing

approach for software development, as it reduces the complexity, time, and cost, and it also helps to integrate NFP analysis during the design phase.

The main goal of our research was to propose a framework based on standard modeling languages (such as UML and QVT-O) that would help modelers and developers to evaluate dependability attributes during a CBD + MDD process, taking into consideration component erroneous behavior and error propagation. We believe that including component erroneous behavior in dependability analysis and prediction will help developers to take the right design decisions. For instance, selecting proper fault tolerance mechanisms, placing error detection, and using suitable recovery approaches are examples of critical decisions taken in the design phase based on quantitative values. The findings of [9], [10] support this belief, since they show that the error propagation may have a significant impact on reliability prediction. Thus, in our approach the evaluation of dependability attributes is based on both normal and erroneous component behavior. Combining the normal and erroneous behavior of the software components in the same UML model leads to a derived SRN analysis model that captures the aggregated effect of normal and erroneous system behavior. This has the advantage of allowing us to calculate global system measures (such as reliability and availability) that combine the probabilities of the system being in normal or erroneous states.

Figure 3.1 illustrates the overall activities of the proposed framework in order to provide context for the contribution of the proposed approach and to put it into perspective. The framework encompasses a set of approaches and phases: 1) modeling component erroneous behavior and composing it with normal behavior; 2) derivation of an SRN model from component behavioral models; 3) verifying component compatibility

and conformance; 4) implementing a transformation chain containing a set of engines for automating the process (such as composing the normal and erroneous behavior, deriving the SRN model, verifying conformance and compatibility, and generating CSPL code describing the derived SRN model); 5) modeling single version fault tolerance as reusable tactics; and 6) building an approximate SRN model by utilizing the proposed transformation composition rules and component decomposition. To explain each phase of the proposed approach, we built a simple case study used throughout the thesis as an illustrative example, which is introduced in Chapter 4. The following sections present a brief overview of each phase.

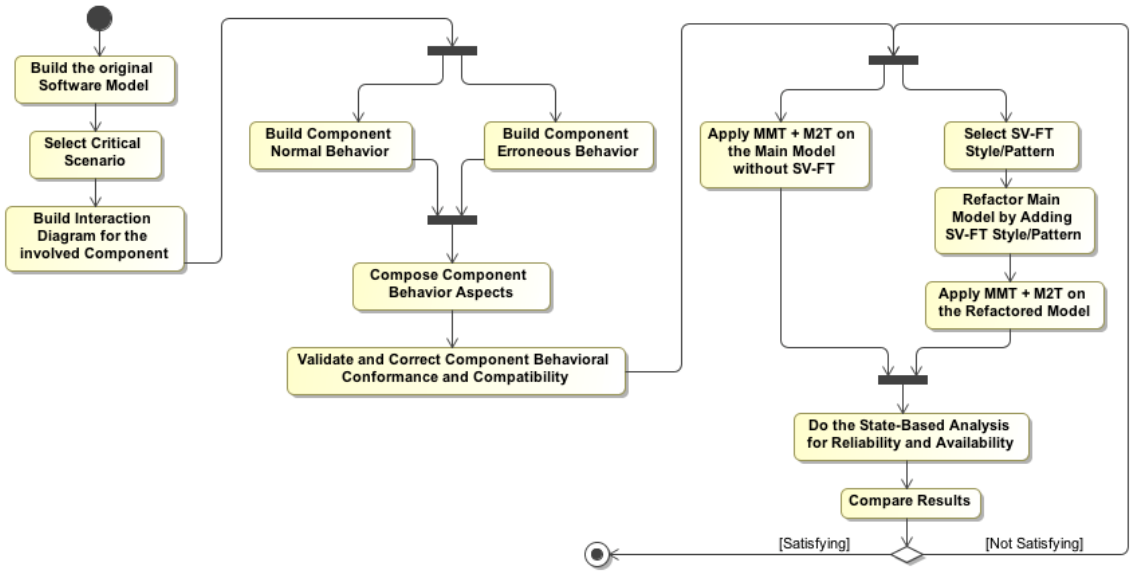


Figure 3.1: Overview of the Proposed Framework

3.1 Modeling Erroneous Behavior

Erroneous behavior in our approach is categorized into three types. First, *hardware node erroneous behavior* models the two states of a hardware node (up, down) as well as the failure propagation to the hosted software components. In the deployment model we use *<<DaComponent>>* stereotype from DAM profile to specify the failure and repair

rates of the hardware node (see Chapter 4 and Chapter 9). An SRN transformation rule presented in Chapter 8 shows how SRN guards are used to model hardware failure propagation to the hosted software components.

Second, *network erroneous behavior* models the failure of the connectors. We assume that a connector failure is recoverable and a failed message will be resent again. The connector links in the deployment model are annotated with <<*DaConnector*>> and <<*GaCommHost*>> stereotypes to specify the link capacity and failure rate (see VT case study in Chapter 4 and FMS case study in Chapter 9). The SRN transformation rule for connectors is presented in Chapter 5.

Third, *software component erroneous behavior* captures the fault activation and propagation to all dependent components. To model software component erroneous behavior, we start with a component-based software architecture model that needs to be evaluated in terms of reliability and availability. For the most important scenarios, we identify the involved components and the interaction between them. Next, we built component behavioral models introducing the Component Erroneous Behavior Aspect Modeling (CeBAM) approach that considers the erroneous behavior of the software components involved. This model is also enriched with dependability annotations using the MARTE and DAM profiles [4], [117]. The Aspect Oriented Modeling (AOM) approach [118] was adopted to allow for more flexibility in modeling erroneous behavior and to provide an automated composition of the erroneous behavior with the normal behavior.

One of the CeBAM advantages is that it provides a practical way to model the erroneous behavior of software components separately from their normal behavior, using

aspect-oriented modeling. This simplifies the state machine models, making them easier to read and maintain. The CeBAM approach is not limited to a certain type of software fault category, but allows the modeler to capture any possible fault activation from UML state activities and to represent the failure propagation to other software components. It is up to the modeler to specify the fault category, how is propagated and how it affects the system behavior (see Chapter 4 for more a detailed description).

The automated composition of erroneous with normal behavior is further used for: a) conformance verification between the internal behavior of each component and its protocol state machines, as well as compatibility verification between components interfaces, and b) derivation of the SRN dependability analysis model. Chapter 4 explains the CeBAM modeling approach using the illustrative case study.

3.2 SRN Model Derivation and Conformance/Compatibility Verification

The next step is the derivation of a Stochastic Reward Net (SRN) model from the CeBAM representations. The derived SRN model is a variant of Petri nets representing the critical scenarios that capture the normal and erroneous behavior. We define a set of transformation rules describing the informal semantics of deriving an SRN subnet from the state machines of individual components. Moreover, another set of mapping rules is introduced to specify the composition patterns of the previously derived SRN subnets.

The transformation is done in four main phases. First, we derive an SRN subnet from each component behavioral state machine. Secondly, we derive an SRN subnet from each port protocol state machine. Thirdly, we apply the composition patterns to compose the SRN subnet derived from the component internal behavior model with the SRN subnets corresponding to the ports. The result is a set of SRN subnets, each representing

one component of the system. The last phase is dedicated to connecting the component SRN subnets according to their provided and required services. The inter-component composition introduces “connector” SRN subnets.

Software architecture validation requires reasoning on the behavioral compliance of a component-based software architecture [119]. The compatibility verification between interacting components and the conformance verification of their internal behavior with the protocol behavior of the corresponding ports are crucial steps for the early identification of unexpected messages between components. A mismatch would negatively impact the component reliability, since the internal behavior would receive unexpected messages that cannot be handled. So any detected mismatch must be corrected before solving the analysis model. In the literature, different approaches are suggested to check for component conformance by finding deadlocks in the formal model obtained from the main software model [120], [121]. Different formalisms may be used, such as various kinds of formal logic or Petri nets. In our dependability prediction framework we follow the state-based analysis methods based on SRN. Therefore, we use the same formal SRN model for conformance and compatibility verification, before using it for dependability analysis. The verification is performed during the composition of the derived SRN subnets and may require the intervention of the developer to fix any detected problems. Chapter 5 presents our conformance and compatibility verification approach, as well as the transformation rules for the derivation and composition of different kinds of SRN subnets.

3.3 Transformation Chain Architecture

Automation is one of the key features in our approach. We utilize Query View Transformation-Operational (QVT-O) language [90] to implement the automation of the proposed approach. We use the Eclipse implementation of QVT-O for model-to-model transformations and the MagicDraw UML editor for constructing software models according to the CeBAM approach. For model-to-text transformation we used Eclipse Acceleo, which is an implementation of the OMG Model to Text Language (MTL). We designed transformation engines by following the best practices of QVT-O, taking advantage of advanced features such as transformation chain, transformation composition, and black-box code. Moreover, we built customized QVT-O libraries for functionalities that are not implemented in Eclipse QVT-O, such as operations that deal with UML stereotypes.

Four transformation engines are developed in the thesis to fully automate the derivation of the analysis model from the annotated software model. Verification mechanism and transformation progress logging are used in order to verify whether the input models conform to the CeBAM modeling approach and to support users with detailed information about the progress of the transformation. Chapters 6 and 7 present the architecture and testing of the following transformation engines:

- *CebamWeaver* automates the CeBAM approach;
- *sm2srn* derives SRN model and validates conformance and compatibility;
- *srn2cspl* is an intermediate transformation that derives CSPL model from the derived SRN and adds SPNP configuration settings;
- *cspl2text* generates CSPL code that is accepted as input by the SPNP tool.

3.4 Modeling Fault Tolerance Tactics as Reusable Tactics

Software fault tolerance techniques can be categorized into three groups: techniques that employ design diversity, techniques that use identical replication with data diversity, and techniques that depend on a single version that has recovery actions [11], [99]. Adding any kind of fault tolerance mechanism is expected to improve the system reliability and availability, but the effects are non-trivial due to the dependency of such mechanisms on the software context [2]. In fact, each fault tolerance technique needs to be customized and tailored to the application using it.

We addressed these issues by introducing the Single Version Fault Tolerance Aspect Modeling (SvFTAM) approach that captures architectural and behavioral models of single fault tolerance tactics as generic reusable aspects annotated with formal dependability attributes. Our work has been inspired by the new vision of software fault tolerance introduced in [12], [114]. Chapter 8 illustrates our proposed approach with three reusable fault tolerance tactics: spare with checkpoint, standby spare, and retry. A generic aspect is instantiated and then its parameters are bound to the application context. The resulting context-specific aspect models are then composed with the original design model. Our proposed approach aims to provide quantitative data for supporting an easy comparison of different fault tolerance tactics, in order to select the best solution for a given system.

3.5 Decomposition and Reduction Technique for SRN model

Solving the derived SRN model using an analytical approach gives accurate results. However, it does not scale-up for large systems due to the state space. The state space explosion is a well-known problem of state-based analysis models such as SRN. In such

cases simulation is used since the reachability graph is not generated, but it only gives approximate results [122].

To address the state space explosion problem, we provide a decomposition and reduction technique for the derived SRN model to compute approximate system reliability and availability measures. The proposed technique takes advantage of the software architecture and composition transformation rules of the SRN model derived from component behavioral models (internal and port). We construct an approximate model that can be solved numerically in four steps. Starting from the SRN model derived for the whole system, we decompose it into subnets following the component-based software architecture. The next step, for each component SRN submodel we identify the candidate SRN subnets that can be reduced without losing the original behavior and preserving subnet properties. This step is performed in a systematic way since we use our SRN composition transformation rules to identify each service execution path and then apply a matching PN reduction rule. In the third step, we build auxiliary models to be solved iteratively for finding the approximate rates of SRN transitions of the reduced subnets. This step is inspired by the work of [14], [15], [123] that introduced an iterative SPN decomposition techniques to compute performance measures. In the last step, we built an approximate model to compute approximate system reliability and other measures. In Chapter 9 we explain the proposed approach, using an illustrative case study, and compare the computed unreliability results between the original SRN model and the approximate model.

Chapter 4: Modeling Component Erroneous Behavior and Error Propagation

A software component has two views: internal and external. The internal view represents the component's private properties realizing the provided services, while the external view shows the public properties of the component in terms of interfaces required and provided. Modeling the erroneous behavior of these views along with the normal behavior in one model tends to be complex and hard to read or modify. Moreover, it is not easy to capture error propagation between components using existing behavior models, such as the UML2 protocol state machine. As a result, developers often focus on the normal behavior of both views and tend to ignore the erroneous behavior.

To overcome these difficulties, we introduce in this chapter the Component Erroneous Behavior Aspect Modeling approach (CeBAM) that captures software component erroneous behavior separately from normal behavior. CeBAM uses aspect-oriented modeling [118] to simplify modeling the erroneous behavior and to automate its composition with the normal behavior. Two kinds of UML state machines are used in CeBAM: a) behavior state machines (BSM) for component internal normal behavior; and b) extended protocol state machine (PSM) for port and interface behavior. Normally, a UML PSM captures only incoming messages, but for completeness we need to model the outgoing messages, as well. Hence, we define a profile to capture both incoming and outgoing messages. Another UML extension developed in this approach is an erroneous behavior profile to capture the chain of dependability threats for the component internal behavior, as well as for its ports and interfaces. In addition, this profile shows the error

propagation from a component internal behavior to its ports and further to other components.

The main contribution of this chapter is the CeBAM modeling approach. This chapter is organized as follows: in Section 4.1 we explain briefly the Vehicle Tracking System (VTS) case study used in this thesis as a running example, the CeBAM approach is introduced in Section 4.2; and the last section explains the CeBAM modeling guidelines.

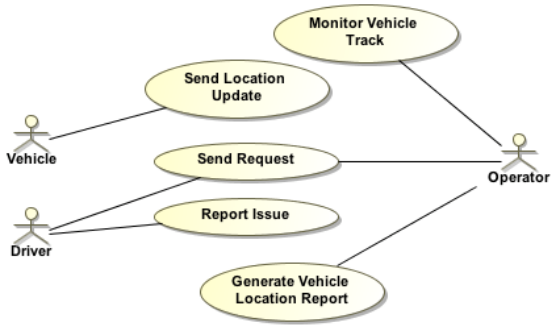
4.1 Illustrative Case Study

The Vehicle Tracking System (VTS) is used as a running example throughout this thesis. In this chapter, VTS illustrates the application of the CeBAM approach, and in Chapter 5 it shows examples of SRN model derivation and of conformance and compatibility verification. This case study was among those used in Chapter 7 to develop test scenarios for the validation of the transformation engines introduced in Chapter 6. Also, it is used in Chapter 8 to show the application of reusable single version fault tolerance tactics to the software architecture and its behavioral models.

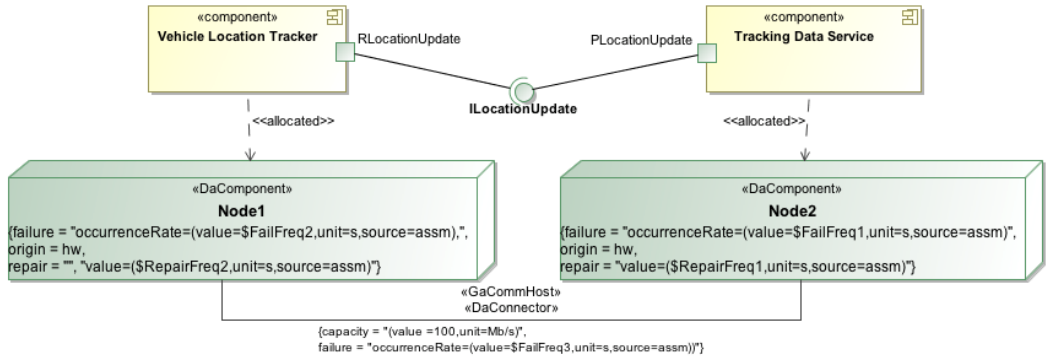
The VTS systems are used in vehicles to send periodic status updates to the central monitoring system, providing the vehicle location and other basic information, such as a send request or a report of issues in the vehicle. Also, the system operator can generate reports or monitor the vehicle track. A system with similar functionality can be installed in taxis, police patrol cars or cargo trucks, in order to provide the central control system with information about the vehicle. We focus on periodically sending a vehicle location update as the most critical scenario. Any failure in this scenario will affect the system availability and reliability. Figure 4.1(a) some of the use cases and Figure 4.1(b) shows

the components involved in the critical scenario (*Send Location Update*), as well as their deployment nodes and connectors. To model erroneous behavior of deployment nodes we use <<*DaComponent*>> stereotype from DAM profile to specify the failure and repair rates. These values will be used in model transformation to derive the SRN subnets of the deployment nodes and to model the hardware failure propagation to the hosted software components using SRN guards (see Chapter 8). Moreover, erroneous behavior of the connector is captured using the <<*DaConnector*>> stereotype that indicates its failure rate. We assume that the network failure is recoverable: if a message is lost, it will be sent again. In Chapter 5 we present the SRN transformation rule for the connectors.

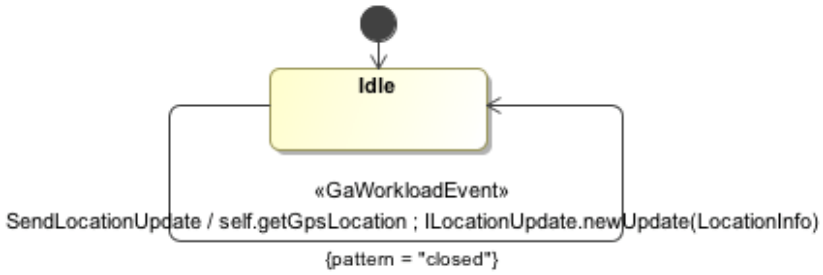
Vehicle Location Tracker is the active component that periodically reports the vehicle location to the *Tracking Data Service* component by calling the *newUpdate* operation. In Figure 4.1(c) the self-transition is stereotyped by <<*GaWorkloadEvent*>> to specify how often the vehicle location is reported to the central system. Once the *Tracking Data Service* component receives the location update request, it will perform the set of actions depicted in Figure 4.1(d). DAM and MARTE profiles are used to annotate state activities. For example, *GaStep* stereotype is applied on state activities to specify their execution duration using *hostDemand* attribute while the *DaStep* stereotype is applied on the same activity to define the fault activation occurrence rate.



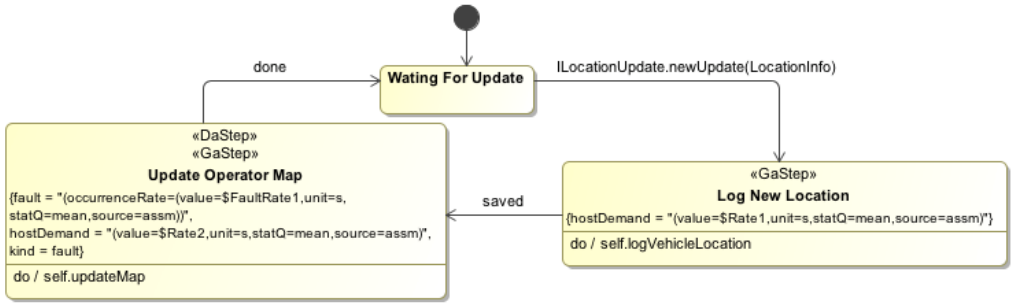
(a) Use cases



(b) Architectural Model of Critical Scenario (Send Location Update)



(c) Internal Behavior of *Vehicle Location Tracker* Component



(d) Internal Behavior of *Tracking Data Service* Component

Figure 4.1: VTS Case Study

According to [1], if an internal fault is activated but not properly handled inside the component, then this fault will end up in a failure, which will propagate to other components that depend on it. For instance, in the selected scenario, if the *Tracking Data Service* component fails due to an internal exception, the manifested failure will be propagated to the *Vehicle Location Tracker* component (see Figure 4.1(b)). As a result, the periodic update of vehicle location cannot be reported to the central system since the core component *Tracking Data Service* component is down.

4.2 Component Erroneous Behavioral Aspect Modeling (CeBAM) Approach

A software component has two views: internal and external. The internal view represents the component's private properties that realize the provided services. The normal behavior of this view can be described using UML behavioral state machine (BSM) [3]. An external view shows the public properties of the component in terms of the interfaces that are required and provided. Interactions between components are either method calls (synchronous) or exchange of notification messages (asynchronous). Protocol state machine (PSM) can be attached to each interface to describe the legal sequence of operations calls [3].

A fault may be activated inside a software component, propagated to the interfaces, and then propagated to all dependent components if it is not handled internally. Moreover, each fault type may have a different propagation path. In fact, modeling both the internal and external view of component erroneous behavior will help to improve the software design. Unfortunately, BSM and PSM do not allow for an easy and practical way of modeling normal and erroneous behavior, due to model complexity and a lack of ability to capture error propagation.

In CeBAM, the normal and erroneous behavior of each software component are described separately, and then are composed together to create a complete behavior model. Each normal and erroneous behavior describes in turn the component interfaces, ports, and internal behavior. In the literature review presented in Chapter 2 we studied many approaches and surveys that present different methods for dependability modeling and analysis, such as [58], [59]. We noticed that most of these methods focus only on the normal behavior, ignoring the erroneous behavior due to its complexity.

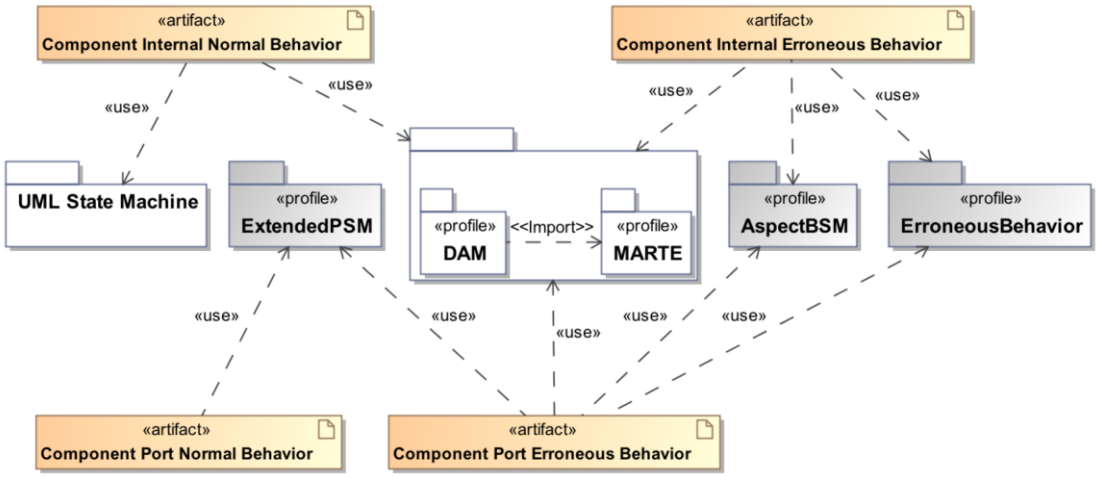


Figure 4.2: Profiles and Artifacts in CeBAM Approach

Our objective in this chapter is to provide a practical solution for modeling the complete software component behavior for both internal and external views, by considering erroneous behavior and error propagation. The CeBAM approach was developed to realize this objective. Figure 4.2 shows the new and existing profiles and artifacts used in CeBAM. For all UML profiles defined in the thesis, we followed the approach from [124], which recommends to define first the domain model as a starting point, and then to map the domain model concepts to the UML2.x metamodel, in order to identify new stereotypes and attributes.

In CeBAM the internal component behavior is modeled using BSM, describing the execution paths for the provided services and the interaction with other components by sending service requests (synchronous or asynchronous). For instance, Figure 4.1(d) shows the internal normal behavior for a single service provided (*newUpdate)* by the *Tracking Data Service* component in the VTS case study. The normal behavior of component ports or interfaces is modeled using extended PSM (as described in Section 4.2.1).

The *ErroneousBehavior* profile and *AspectBSM* are used together to model the erroneous behavior (both internal and external views) separately from the normal behavior as aspect models. Then we use OMG's Query View Transformation Operational (OMG QVT-O) [90] to automate the composition of the erroneous behavior with the normal behavior for both views (Sections 4.2.1, 4.2.2, and 4.2.3). Since we also consider protocol state machines, we validate the conformance between component behavior and their PSMs, as well as between component interfaces, as explained in Chapter 5.

In CeBAM we adopted the aspect oriented modeling approach [118] to model the component erroneous aspects. Actually, we consider erroneous behavior as a crosscutting concern that can be modeled separately and then we automate its composition with the base model (which represents the normal behavior) for both views. Using this approach, a developer will not have to learn a new language in order to model the erroneous behavior with dependability annotations. Additionally, there is full flexibility to update or change any behavior separately as the project evolves, since the composition of the complete behavior is automated, as explained in Chapters 6 and 7.

4.2.1 Extending Protocol State Machine

According to [3], a PSM is a specialized behavioral state machine defined in the context of a classifier that can be used to specify which operations of the classifier can be called in which state and under which conditions. A PSM is used to describe only the legal usage of any classifier, but it does not show any specific behavioral implementation, since actions are not allowed on transitions or in the states. Actually, states in PSM do not have *entry*, *exit*, or *do* activities. On the other hand, composite states and concurrent regions are allowed, but history pseudo-states are not. (We are not using concurrent regions in the proposed approach.)

A protocol transition captures a legal transition of the context classifier, and has a pre-condition, a trigger, and a post-condition. The protocol transition shows that the associated operation can be called under a specific condition (pre-condition) and then after its complete execution, the destination state can be reached if the post condition is satisfied. Moreover, PSM inherits run-to-completion semantics from BSM, i.e., the action on a transition is uninterruptible. This implies that no other event can be accepted during the transition. Additionally, nested calls cannot be captured.

Due to its restrictions, only unidirectional communications can be captured by PSM [3], [119]. For instance, in Figure 4.1(b) we can use PSM to model the communication of the provided interface, which is connected to the *Tracking Data Service* component. In this case, PSM can only capture incoming calls to that interface. Moreover, for each interface we have to create a separate PSM, since the communication can only be captured from the viewpoint of a single classifier.

In UML, a port is a property of a classifier [3]. A port can be associated with a component (i.e., the UML classifier) to specify an interaction point between the component and its environment and between the component and its internal parts. A combination of interfaces that are required and provided can be associated with a port; thus, a port may specify services provided to other components, as well as required services. A component can have any number of ports. Incoming or outgoing request are not cached at the port; the incoming requests are immediately passed from the environment to a component's internal behavioral; and the outgoing requests sent by the component internal behavior are processed in a similar fashion. In the VTS case study, each component has only one port for interaction with the environment; thus, the external view of the component is actually the port behavior.

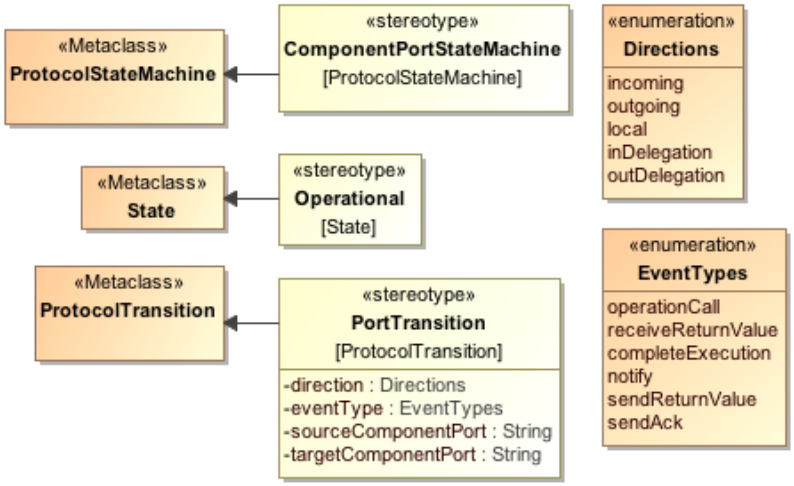


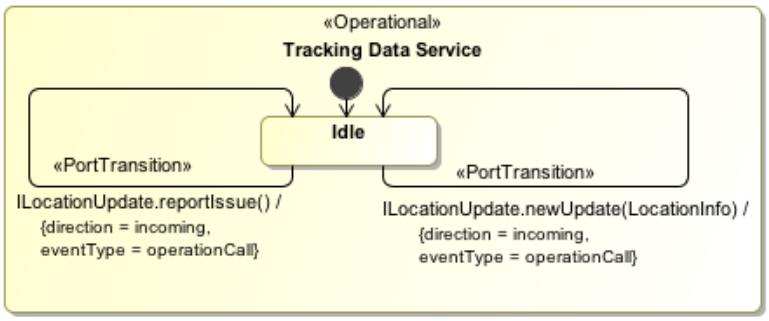
Figure 4.3: ExtendedPSM Profile

The external view of the component behavior is described by its PSM. As mentioned before, PSM can be used to capture that behavior precisely, but it captures only a single direction (incoming) and it does not allow for recursive calls due to the run-to-completion semantic. To overcome these limitations, we extended the PSM by

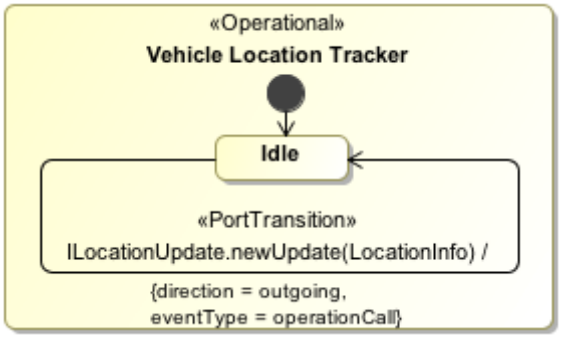
introducing a new profile, as shown in Figure 4.3. This profile will be used to model "extended" protocol state machines.

The external view of a component is represented by the behavior of its ports. Each port behavior has two types of states: operational state and failure mode states. First, an operational state is a composite state stereotyped with <<*Operational*>> stereotype. It owns a set of simple states and port transitions. For each port there is only one operational state that represents the normal protocol behavior of the port. This behavior is represented by component public states and transitions using *PortTransition*. In *PortTransition* we can capture the direction of each passing message, either incoming or outgoing. Sometimes the PSM state may change because of an internal event. The second kind of state, called "failure mode states", capture all failures that are propagated to the component port. A failure mode state is connected to the operational state using a *PortTransition*, which captures the failure propagation.

In this profile we have respected the run-to-completion semantics and we can show atomic events. For each event in *PortTransition*, we specify the direction (incoming, outgoing, or delegation) and the event type (operation call, notify signals, receive return value from the called operation, or complete execution signals). Moreover, we show the source and target component associated with the event. In our approach, the normal port behavior has only one operational state that describes the normal protocol behavior. For erroneous behavior, we have a set of failure mode states that capture all possible failure modes propagated to component ports. Section 4.2.3 presents how to model the erroneous behavior of a component port as an aspect model, to be composed with the port normal behavior.



(a) *PLocationUpdate* Port of *Tracking Data Service* Component



(b) *RLocationUpdate* Port of *Vehicle Location Tracker* Component

Figure 4.4: Ports Normal Behavior Protocol State Machines

The *Tracking Data Service* component of the VTS case study has one provided interface with two services. This interface is associated to *PLocationUpdate* port (see Figure 4.1(a)). To describe the external view of the component, we used UML and the *ExtendedPSM* profile to model the protocol state machine of this port (see Figure 4.4(a)). We created composite state with <<*Operational*>> stereotype, which has states and port transitions that capture the port normal protocol behavior. It receives two incoming messages from other components (*newUpdate* and *reportIssue*). For example, the *newUpdate* method is implemented by the *Tracking Data Service* component and, therefore, different actions (such as logging vehicle location and updating operator map) will be done internally. In some cases, such actions will change the state of the PSM, as precisely captured by the ExtendedPSM profile. In Figure 4.4 (a), each transition of the

PLocationUpdate PSM is atomic and has run-to-completion semantics. In addition, the direction and the event type are also indicated. In a similar approach, we model the *RLocationUpdate* port PSM of *Vehicle Location Tracker* component, as shown in Figure 4.4(b). This port is associated with a required interface called *ILocationUpdate* that has *newUpdate* service and provided by the *Tracking Data Service* component.

4.2.2 Aspect Composition

We adopted the AOM mechanism [118] to model separately a component's erroneous behavior and then to compose it with the normal behavior. Figure 4.5 shows the domain model for the proposed AspectBSM profile and Figure 4.6 shows the actual profile definition.

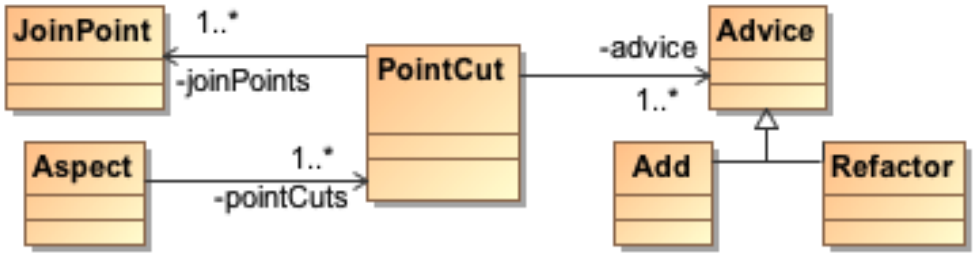


Figure 4.5: Aspect Domain Model in Erroneous Behavior Context

This profile is based on the main concepts of aspect-oriented modeling and is similar to the approach from [125]. *Aspect* describes a crosscutting concern; in our context, the aspect will be the erroneous behavior of both component views. For each crosscutting behavior, we have a *pointCut*, which is a condition expresses by a query that identifies the place(s) where the new behavior should be added in the base model, or which model element needs to be refactored. A candidate element in the base model that corresponds to a *pointCut* is called *joinPoint*. In other word, the *pointCut* query will select one or more *joinPoints* (model elements) where the new behavior can be applied.

In our approach, the *pointCut* is an OCL query that selects states or transitions. Note that we will not add any new stereotype in the base model to identify the *joinPoint*. *Advice* is a new behavior inserted in the base model at the *joinPoint*, and it could be *add* or *refactor advice*.

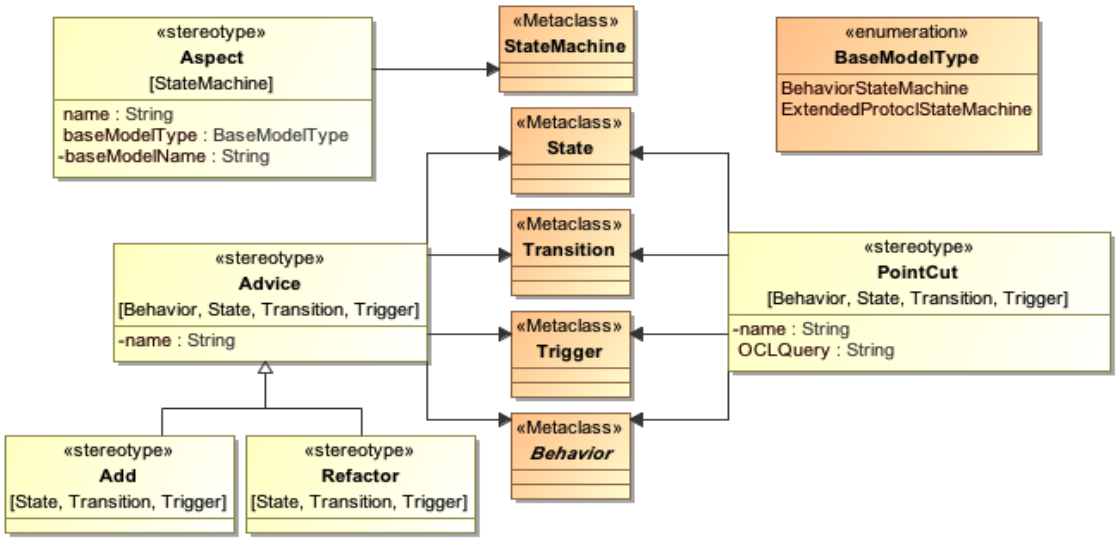


Figure 4.6: AspectBSM Profile

As mentioned before, transitions in behavioral state machines have run-to-completion semantics according to [3]. In some cases, the action associated with a transition is an operation call. The operation must be executed successfully before entering the new state. However, during its execution, faults may be activated that will interrupt the transition. Our objective is to model any fault that may be activated during the transition, but at same time we want to respect the run-to-completion semantics. To achieve that, we introduce a *refactor* advice applied to the respective BSM transition. Before adding the erroneous aspect of that operation we should introduce a new state called *intermediate state* and a new transition called *done*. Figure 4.7 illustrates an

example of *refactor* aspects applied to the internal behavior of the *Vehicle Location Tracker* component.

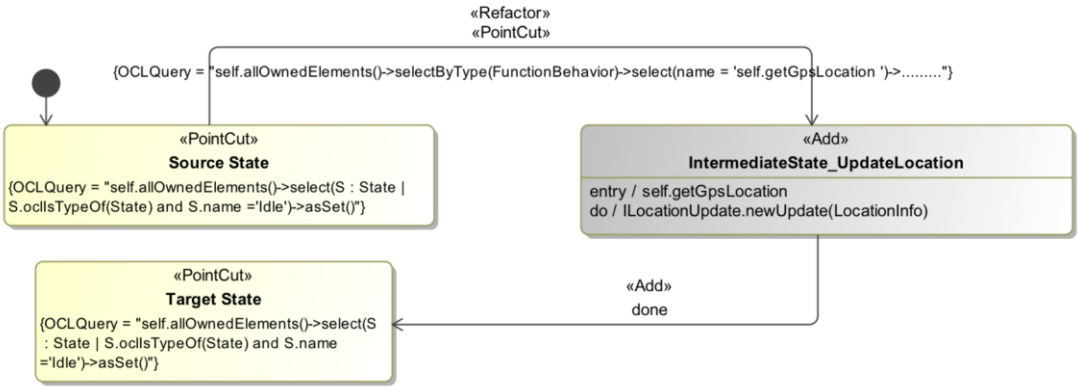


Figure 4.7: Refactor Aspect of *getGpsLocation* and *newUpdate* Activities

For instance, *self.getGpsLocation* and *ILocationUpdate.newUpdate(LocationInfo)* are operations executed as an effect of the self-transition from the state *Idle* (see Figure 4.1(c)). First, we identify the source and target states of that transition and then we add a new state (called *IntermediateState_UpdateLocation*) reached from the source state with the original transition, but without any call to the effect operations. We move the *self.getGpsLocation* and *ILocationUpdate.newUpdate(LocationInfo)* operations to the newly introduced state as a *do* activity. Finally, we add a new transition *done* from the new added state to the target state. This new transition represents the successful execution of the transition action from the base model; it is triggered by an event of type *SignalEvent*. In this way, we preserved the run-to-completion semantics and we can later add an erroneous transition from the *IntermediateState_UpdateLocation* state, as explained in the next subsection.

We use the *refactor* aspect only in BSM in describing the internal behavior of a component, but we do not need refactoring in the protocol state machines developed with

ExtendedPSM because all transitions are already atomic. Moreover, depending on the component BSM, a developer may develop a set of refactor aspect models to be applied to different transitions.

4.2.3 Modeling Component Erroneous Behavior

Different error states and failure modes can be identified for a single component. Each failure may have a different propagation path. Our objective is to model the error propagation between components separately as crosscutting concerns in order to study how this propagation impacts the overall reliability and availability of the system. We developed a new profile to model the component internal and external erroneous behavior, as well as the error propagation between components. Figure 4.8 shows the profile stereotype and attributes. This profile captures the two kinds of states (erroneous and failure modes) and transitions (erroneous and recovery). For each transition type it depicts the direction, event, source operation, and target operation.

Using ErroneousBehavior profile we can represent different kinds of failure caused by software exceptions or other failures types. The modeling techniques proposed here is not limited to a specific software fault category. It is the modeler's responsibility to specify the possible faults from the state activities and to represent the failure path and propagation. In CeBAM we use this profile together with the AspectBSM profile to model erroneous internal component behavior and PSM erroneous behavior as aspects, separately from the normal behavior.

The erroneous aspect model captures the erroneous and recovery behavior of a component's port and internal behavior in a set of erroneous paths. The first state in this model is the submachine state that points to the normal behavior of PSM or BSM. The

outgoing transition must be stereotyped by <<*localFaultActivation*>> for a local activated fault or <<*ExternalFailurePropagation*>> for a failure propagated from another dependent component. This transition represents the start of an erroneous path. The target states of these transitions are *pointCut* states that identify the *joinPoint* the base model using the *OCLQuery*. Therefore, an erroneous path will be added to the identified *joinPoint*. The erroneous path can have one or more erroneous states and a single failure mode state connected to each other using propagation transitions.

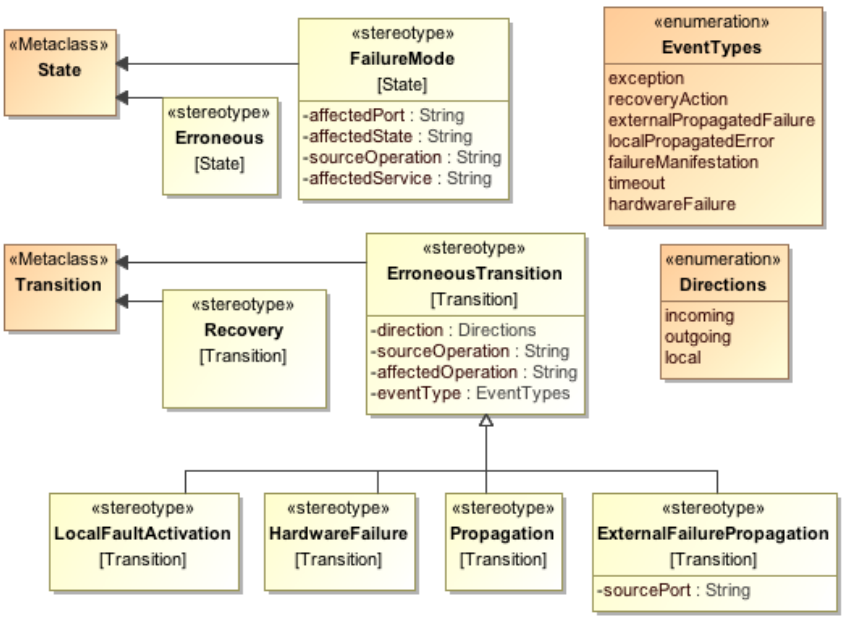


Figure 4.8: Erroneous Behavior Profile.

We used a *DaStep* stereotype of the DAM profile on propagation transition to specify the propagation rate from erroneous state to another erroneous state or to the failure mode state. A recovery transition can be used to model a recovery action. It starts from the erroneous state and is linked with the *pointCut* state that specifies the target state in the base model. An example is presented in [116]. In recovery transition we use also

DaStep stereotype to specify the recovery rate. In Chapter 8 and [126] we show examples of recovery action as fault tolerance style.

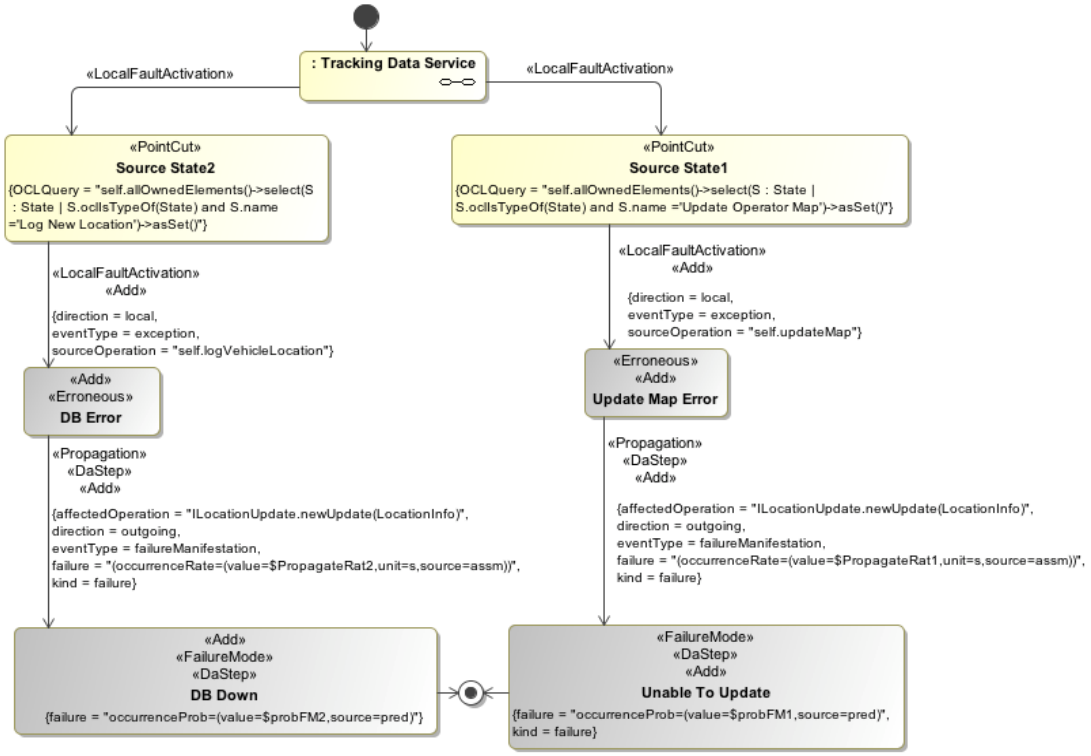
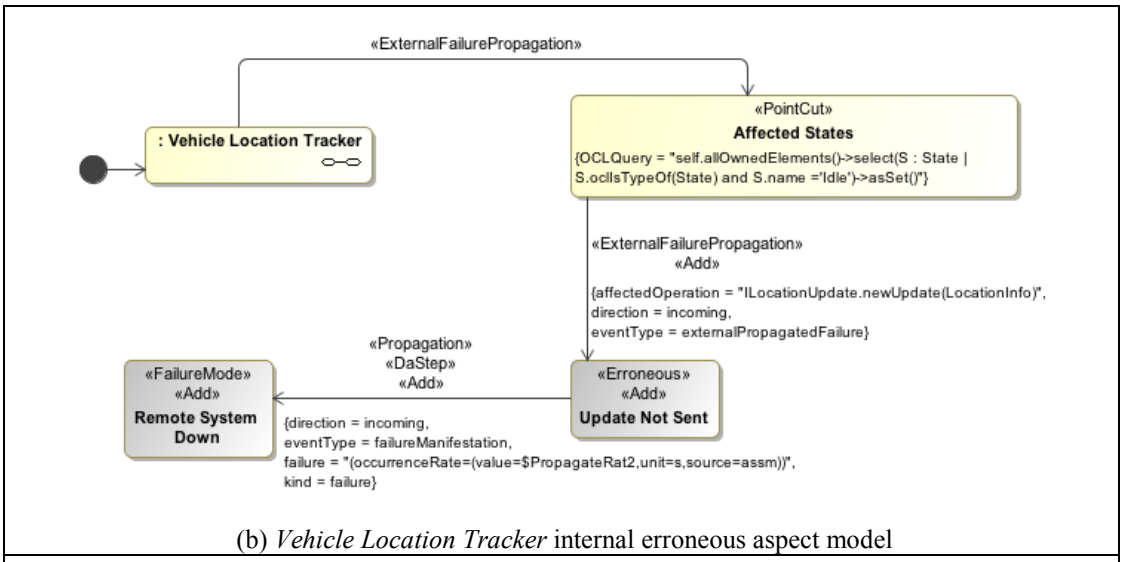


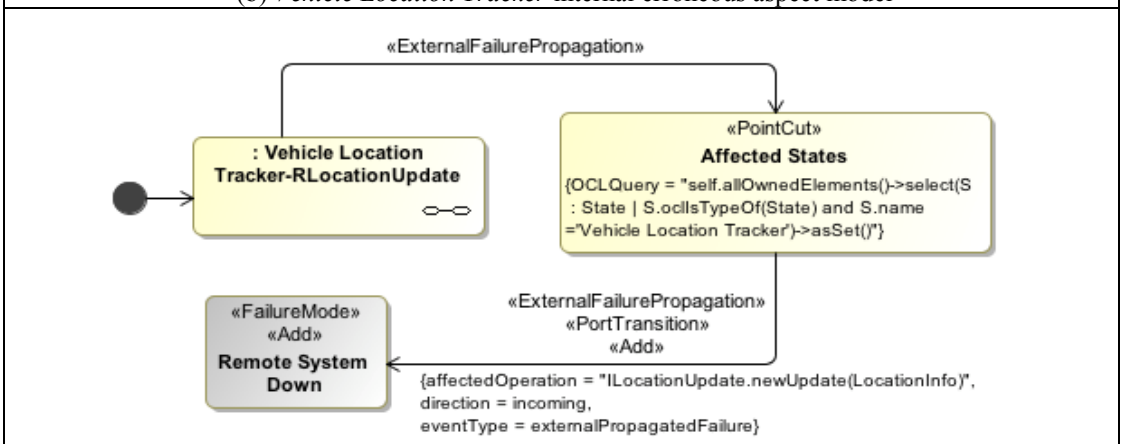
Figure 4.9: *Tracking Data Service* Internal Erroneous Aspect Model

Figure 4.9 shows different errors and failure modes activated internally in the *Tracking Data Service* component. It has two local failures belonging to the provided service execution path (*ILocationUpdate.newUpdate(LocationInfo)*). Initially, faults are activated because of an internal exception occurring in the *UpdateMap* and *logVehicleLocation* operations that are part of the realization of the provided service. The activation of any fault will lead to erroneous states and then a failure mode manifestation. The internal failure will be propagated to the component port and then to the connected component causing another error and failure type according to [1]. In addition, the profiles in CeBAM were designed to capture all required details described in [1], [72],

[88] to model component internal and external erroneous behavior. The failure manifested in *Tracking Data Service* component propagated to the port and internal behavior of the *Vehicle Location Tracker* component. Figure 4.10 shows how this propagation is modeled as an aspect model.



(b) *Vehicle Location Tracker* internal erroneous aspect model



(c) *RLocationUpdate* port erroneous aspect model

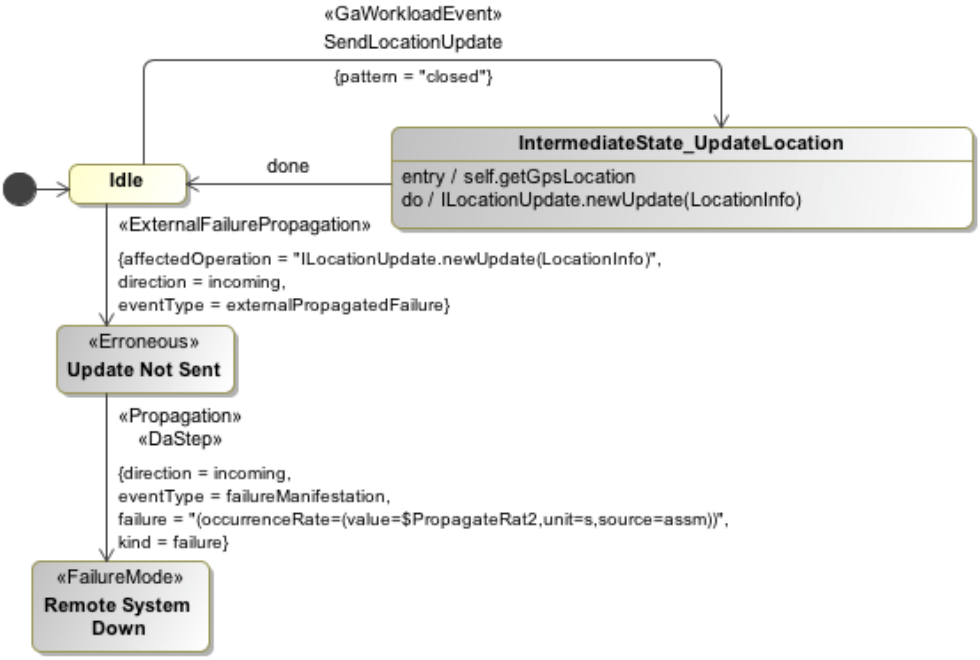
Figure 4.10: Erroneous Behavior Aspect Models (internal and port) of *Vehicle Location Tracker* Component

The internal behavior of a component describes the execution path of each provided service. For each service, a set of faults may be activated, causing different failure modes.

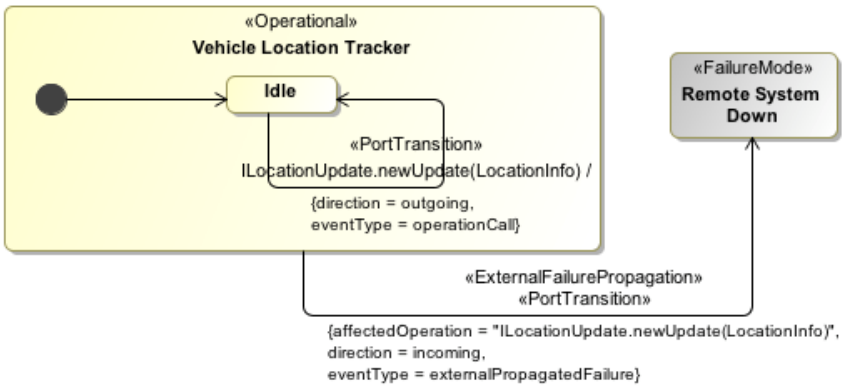
The failure modes of each service path are mutually exclusive, therefore, if a given failure mode is manifested in a service path, then it propagates to a component port; no other failure mode is activated. Since the port shows the public states of the component, each failure mode is captured in the component port and represents one or more failure modes of the service execution path. We call this behavior a failure mode encapsulation and it should be considered by the modeler when developing component erroneous behavior. For instance, the failure modes of the *Tracking Data Service* component (Figure 4.9) will be propagated to *PLocationUpdate* port as a single failure mode since both of them belong to the same provided service and they are on the same execution path of the provided service (*newUpdate*).

4.2.4 Behavior Composition

We follow the AOM [118] approach to compose both behaviors. Composition directives describe the sequence and the order in which aspects need to be composed with the base model. For instance, in the behavioral state machine of the component internal behavior the *refactor* aspects will be processed and applied first, before any *add* aspects. Moreover, the applied stereotypes of AspectBSM will be used as a composition directive to apply the refactor aspect and to weave erroneous behavior. Figure 4.11 shows the BSM and PSM for the *Vehicle Location Tracker* component after composition. The composition process is implemented using QVT-O, as explained in Chapters 6 and 7. A user will develop component behavioral models (PSMs and BSM) and a dependability expert will build component erroneous behavior using the CeBAM approach.



(a) *Vehicle Location Tracker* complete internal component behavior



(b) *RLocationUpdate* port complete protocol behavior

Figure 4.11: *Vehicle Location Tracker* Complete Component Behavior

The *CebamComposition* transformation engine will compose each component behavior with the aspect models. It starts by applying refactor aspect models on a component's original BSM. The next step is weaving the erroneous aspect model to the

refactored model. To fully automate the process of aspect model composition, the transformation engine uses the QVT-O Black-Box extension mechanism to parse a *pointCut* query string and execute it on the base models (BSM and PSM). Moreover, it verifies the correctness of the aspect models that must conform to CeBAM modeling guidelines, before weaving them on the base model.

4.3 Guidelines for using CeBAM modeling approach

Modeling component behavior using CeBAM can be done in two phases. In the first phase we just model the normal behavior of both component views (internal and external). BSM will be used for the component internal normal behavior (see Figure 4.1(c,d)) and extended PSM for the external view (see Figure 4.4). The second phase focuses on modeling component erroneous behavior separately using two profiles: AspectBSM and ErroneousBehavior. The outcome of this phase is represented by two aspect models: one for the erroneous behavior of the internal view and another for the external view. We may need a few iterations to build these two models. First, we capture the local failures and then in the next iteration(s) we may have to add propagated failures that affect other components. The iterations will end when all errors/failures have been "propagated." In some cases, we may need to create refactor aspects to preserve the run-to-completion semantics of BSM transitions. However, implementation of the *CebamWeaver* and *sm2srn* transformation engines helps in detecting modeling errors, such as failure mode defined in BSM, as explained in Chapters 6 and 7, but it is not propagated to component PSM and other dependent components.

The ultimate goal is to derive and compose an SRN model of the critical scenario under analysis. As explained earlier, the aspect models applied to compose complete

component internal and ports behavioral models that have normal and erroneous behavior. A complete component behavior model is used to derive the SRN model by applying a set of transformation rules, as illustrated in Chapter 5. However, to derive the correct SRN model, the user must follow modeling guidelines while developing component behavioral models and aspect models. Models conforming to the CeBAM modeling approach and guidelines will guarantee deriving a correct SRN model using the transformation engines presented in Chapters 6 and 7. In fact, the transformation engines (*CebamWeaver* and *sm2srn*) verify the correctness of the input models before weaving the aspect models and deriving the SRN model. If any violation of the CeBAM modeling approach and guidelines is found, descriptive error messages are provided to help the user fix the modeling error.

The following list provides modeling guidelines that must be considered while modeling component behavior according to the CeBAM approach. They help the users to build a component behavioral model (internal and ports) that translates to a correct SRN model using the developed transformation engines:

- The component internal behavioral model represents the service execution path of provided services. Each service path starts from any state (usually the initial state, i.e., *idle*); the trigger event of the outgoing transition represents a service call.
- The service execution path may end in a failure mode or in getting back to the initial state or to another normal state. This rule will lead to the derivation of a SRN subnet that is bounded and safe by construction for each service execution path. This subnet has only one token representing a service request that ends either in normal state or in a failure mode (see Chapter 5 and Chapter 9).

- A provided service of other components is called only from the *do* activity of any state in the service execution path.
- A state can have any combination of *entry, exit,* and *do* activities.
- The *do* activity represents a local method execution or a provided service call.
- The *entry* and *exit* actions are used to represent only local method executions.
- The *GaStep* stereotype is applied to state *entry, exit,* and *do* activities as follows:
    - *HostDemand=(value=$Rate,unit=s,statQ=mean,source=assm)*
- The *DaStep* stereotype is applied to state *entry, exit,* and *do* activities as follows:
    - *failure=:(occurrenceRate=(value=$Rat1,unit=s,source=assm))*
- The *DaStep* stereotype is applied to a propagation transition as follows:
    - *failure=:(occurrenceRate=(value=$PropagateRat1,unit=s,source=assm))*
- The interface name must be included in the provided service call i.e., <*InterfaceName*>**.**<*operationName*>.
- For local method execution, the *self* keyword must be included, i.e., <*self*>**.**<*operationName*>.
- All transitions must have triggers associated with the event types *CallEvent*, *Signal,* and *SignalEvent*.
- The names of state activities and transition events must be unique.
- A state with no actions or activity should have only an outgoing transition with a trigger event type, such as *CallEvent*.
- For a synchronous service call, the return parameter must be specified, otherwise it will be considered as an asynchronous call.

- The behavior type of *entry, exit,* and *do* activities must be *FunctionBehavior* or *OpaqueBehavior*.
- The outgoing transition of states that has an *exit* action must only trigger with an event type, such as *CallEvent*.
- All erroneous states and transitions must be stereotyped according to the CeBAM modeling approach.
- Failure mode encapsulation must be considered while developing component internal erroneous behavior and its port behavior. The *affectedOperation* attribute of ErroneousBehavior profile applied on propagation UML transition is used to specify the affected provided service from the manifested failure mode. All failure modes with the same value of *affectedOperation* attribute will be considered as an encapsulated failure mode since it belongs to the same service execution path.
- The OCL query used to specify the *pointCut* must be correct and return a value.

  The following are examples of template OCL queries:

```
self.allOwnedElements()->selectByType(FunctionBehavior)->
      select(name = '<FunctionBehaviorName>')->
            collect(owner)->selectByType(Transition)->asSet()
```

```
self.allOwnedElements()->selectByType(State)->
      select(name ='<stateName>')->asSet()
```

```
self.allOwnedElements()->selectByType(FunctionBehavior)->
      select(name ='<FunctionBehaviorName>')->
            collect(owner)->selectByType(Transition)->
                  select(target.name='<stateName>')->asSet()
```

- All states in an erroneous aspect are simple without any activities.
- An erroneous path could be shared between multiple local behaviors.

- The source operation attribute value of *LocalFaultActivation* stereotype is the name of *entry*, *exit*, or *do* activities.
- One fault activation transition is allowed for each state activity (*entry*, *exit*, or *do* activities).
- An external failure propagation transition can be added to any state in the service exaction path.

Chapter 5: SRN Model Derivation from Component Erroneous Behavior Model and Compatibility/Conformance Verification

The contributions of this chapter are twofold. First, it defines a set of transformation rules to derive SRN models from a component's internal and ports behavioral models. These rules are categorized into mapping rules and compositional rules. The mapping rules show the derivation of the SRN subnet for each model element in the behavioral models. Compositional rules are responsible for composing the derived SRN subnet, (i.e., composing the internal and port behavior of a component).

Second, the chapter provides an approach for verifying component behavior conformance and compatibility. The behavior of a components port, described by an extended protocol state machine (PSM), must conform to its internal behavior modeled by a behavioral state machine (BSM). Thus, the goal of conformance verification is to avoid any unexpected messages between the internal behavior and the ports of a component. Component compatibility verification aims to avoid a mismatch between connected components in terms of provided services or failure propagation. For instance, a mismatch is created when an internal component failure is raised but not captured in the corresponding component port, or when a component failure cannot be propagated and handled by communicating components.

This chapter is organized as follows. Sections 5.1 and 5.2 illustrate the derivation of SRN models from BSM and PSM, as well as the semantics behind each transformation rule. Section 5.3 shows component behavioral composition patterns. The last section

describes the proposed approach to component conformance and compatibility verification and applies it to the running case study.

5.1 Transformation Rules for Behavioral State Machine

A state machine is a behavioral diagram that may be used to specify the behavior of a part of a designed system, and it comprises states and transitions. A state represents a situation of the component or object when some invariant condition holds. For instance, a specific state may represent a static situation when the component is waiting for another event to occur [3], or a dynamic situation when the component is performing some internal computational activities resulting from an external call event.

UML2 defines three kinds of states: simple state, composite state, and submachine state. The simple state machine does not have any sub-states or regions, while a composite state has sub-regions and sub-states. A submachine state models the reuse mechanism by referring to another state machine. As explained in Chapter 4, along with simple states we use composite states for modeling port protocol behavior and submachine states for the component erroneous behavior aspect model. In addition, UML2 defines ten different kinds of pseudostates. In our case, we focus on initial and final pseudostates. States can optionally have *entry/exit* actions and *do* activities. An *entry* action is executed when entering the state, while the *exit* action is executed when leaving the state. A *do* activity is executed after the entry actions (if any) and continues as long as the state is active. Both actions (*entry/exit*) have "run-to-completion" semantics, which means that those actions are uninterruptable and once they have started no other new events may be executed. On the other hand, a *do* activity does not have "run-to-completion" semantics. In our approach, we respect and preserve these semantics. For

instance, an entry activity that contains a method execution will not be interrupted by another call until it has finished its execution. However, during the execution, a fault may be activated to change the component's state from a normal to an erroneous state. Fault activation in this case is not a new incoming event dispatched from an event pool of that state machine, but it is changing the execution path of the activity that we precisely model in CeBAM.

According to [3] state machines have three kinds of transitions: local transitions, internal transitions, and external transitions. Local transitions are used in a composite state to leave any state and/or to enter a new state (i.e., by entering a new sub-state). Internal transitions do not cause a state change, since the source and target states are the same. External transitions do cause a state change; they have an *event[condition]/action* label indicating a dispatched event and a guard evaluated when the event is received by the component: if the guard condition is true, the transition is taken and condition is right and the action is executed. A transition action has "run-to-completion" semantic, which means that the software component will enter the target state only when the action is successfully completed, and no other event may be accepted during the action execution. A special kind of external transition is a completion transition (*done*), which is triggered by an event of type *Signal* or *SignalEvent* (marking the end of the execution of a *do* activity in the state) and has no actions.

As mentioned before, we use a behavioral state machine to describe the internal behavior of each component in the system. In order to capture the erroneous behavior of any action on state transition without violating the "run-to-completion" semantics, we introduced the refactoring aspects in Chapter 4 (see also [116]). This aspect adds a new

intermediate state and transforms the transition action into a *do* activity inside of the new state. As a result, the *do* activity is interruptible and allows us to model the fault activation. Moreover, after applying the refactoring aspect on the original behavioral state machine, the refactored model will only have transitions with events or completion (*done*) transitions.

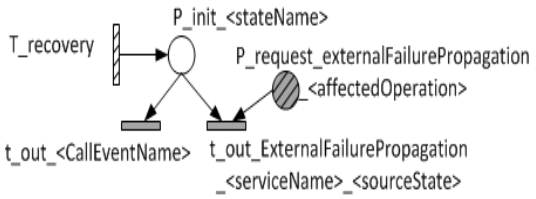
In the following section we will describe the informal semantics of how we derived the SRN model from the refactored behavioral states machine that captures the internal software component behavior (normal and erroneous). In the literature, different approaches are suggested to translate state machines to Petri Net for analysis, such as [50], [121], [127]. Our approach of deriving SRN from BSM is inspired by [50]. Initially, it concentrates on how to iteratively build the SRN model from BSM and PSM, and then it shows the mapping of MARTE and DAM profile attributes to the derived SRN.

5.1.1 Transforming a Simple State

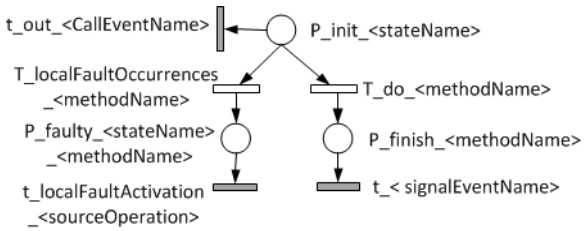
A simple state in our approach may represent a normal or erroneous behavior of the component. A normal state may have optional *entry/exit* actions and/or *do* activity that model method executions or signals. As explained in Chapter 4, the only state activity that sends service requests (operation calls) to another component is the *do* activity, while *entry/exit* actions are always executed locally. Conversely, erroneous states (error and failure modes) do not have any actions or activity, as they capture component dependability threats and error propagation.

For normal behavior, we define different generic transformation rules shown in Figure 5.1, covering possible cases of a simple state. In order to compose the translated model elements we adopt the idea of interface transitions and places as in [50]. We use

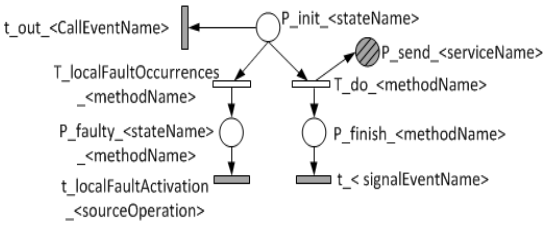
interface transitions and places that connect a simple state with its transitions based on matched names. We have three types of SRN interface transitions: a) a *done* or completion execution interface transition (*t_<signalEventName>*) triggered once the corresponding state activity is completed; b) an *out* interface transition (*t_out_<CallEventName>*) representing the transition between two states with a triggering event; and c) a *recovery* interface transition associated with the initial place of the target state to represent recovery from the erroneous state. The legend in Figure 5.1 describes the SRN model elements and shows how to graphically differentiate between different transitions and place types. To facilitate things, we use a naming convention for each derived element during the composition of SRN subnets. It helps to identify the errors in composition and to trace them back to the conformance issues in the main software model. For instance, for an initial place we concatenate the label *"P_init"* with the state name. The *"t_localFaultActivation_<sourceOperation>"* is another example of naming convention for SRN transitions derived from UML erroneous transitions (fault activation). The value of *<sourceOperation>* is derived from the applied ErroneousBehavior profile attribute, and it is used to compose an SRN subnet derived from a simple state with an SRN transition derived from an outgoing fault activation transition. In the implementation of these rules (see Chapters 6 and 7) we associate each SRN model element with a number to avoid naming conflicts in the SPNP tool.



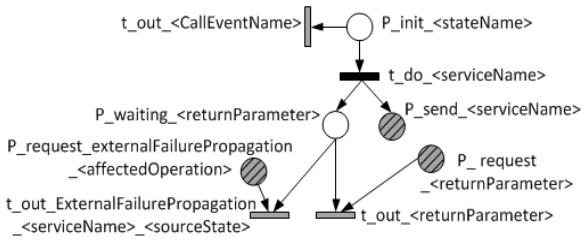
(a) Simple state with no activities and no actions



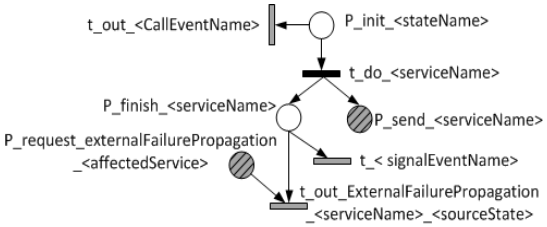
(b) Simple state with *do* activity executed locally with no return



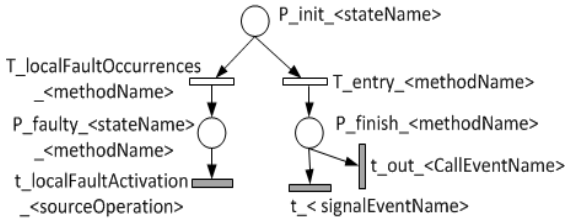
(c) Simple state with *do* activity executed locally with return value



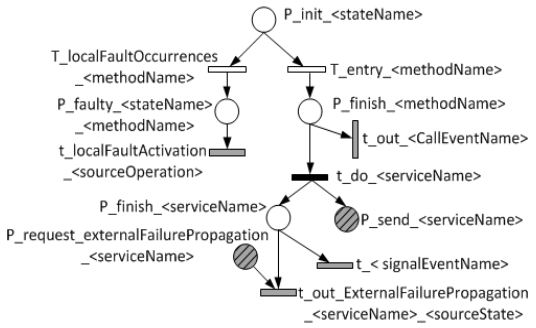
(d) Simple state with *do* activity calling provided service (synchronous call style)



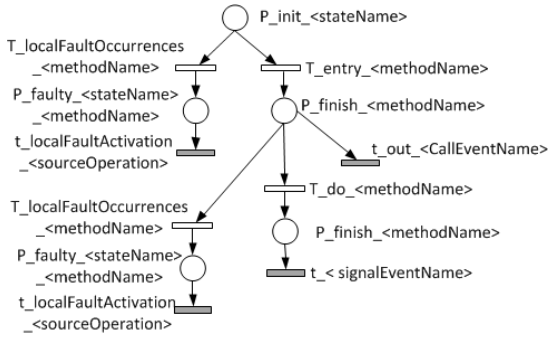
(e) Simple state with *do* calling provided service (Asynchronous call style)



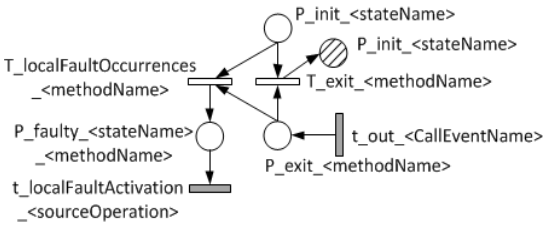
(f) Simple state with *entry* action executed locally



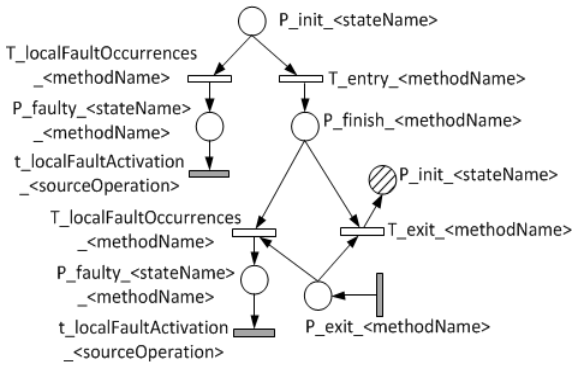
(g) Simple state with *entry* action executed locally and *do* activity call remote service



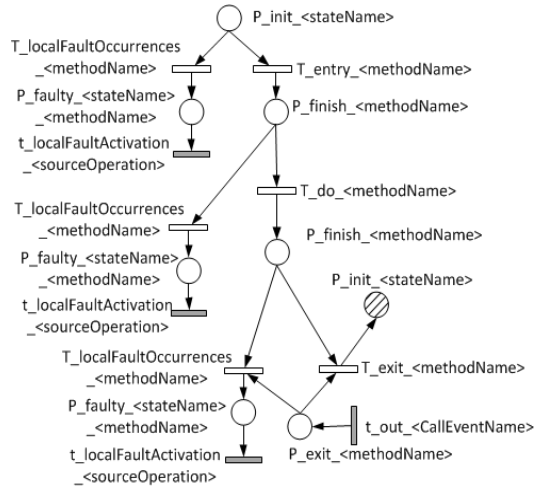
(h) Simple state with *entry* action and *do* activity both executed locally



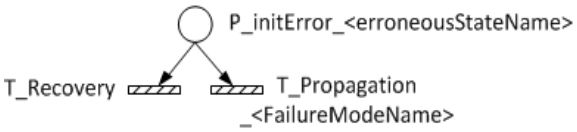
(i) Simple state with *exit* action executed locally after triggering state transition



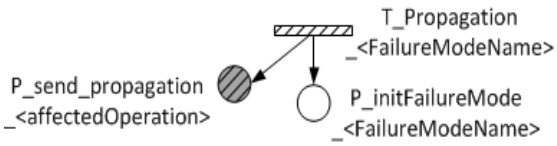
(j) Simple state with *entry* and *exit* executed locally



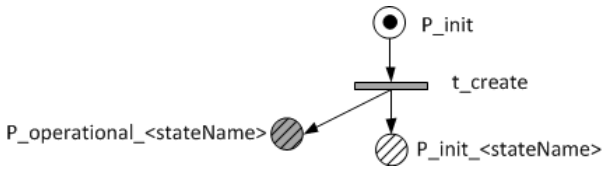
(k) Simple state with all local activities



(l) Error state with recovery action



(m) Failure mode state with propagation to component port



(n) Initial Pseudostate

Legends

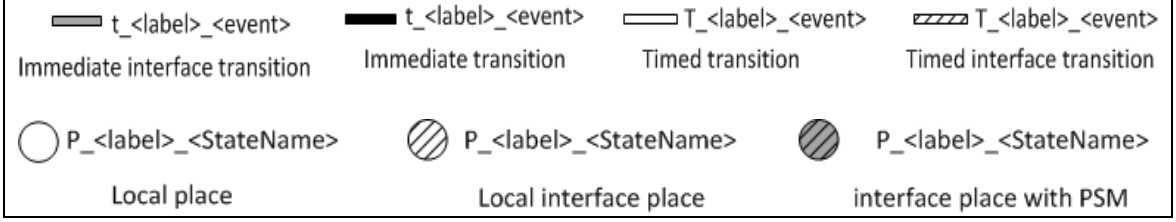


Figure 5.1 Graphical Transformation Rules for Behavioral State Machine

Figure 5.1(a) shows the transformation of a simple state without any activity or actions. We transforme it into one place connected to the interface transitions. The first interface is the *out* SRN transition with the naming convention *t_out_<CallEventName>,* which represents the SRN transition derived from the outgoing UML transition triggered by an event of type *CallEvent*. The number of *out* SRN transitions depends on the number of outgoing UML transitions from the simple state. The second interface captures the failure propagated from a dependent component. The last interface is a timed transition that represents the recovery action. In some cases, the error may be recovered and the system can be restored to a specific state. To model recovery, we added a timed interface transition connected to the initial healthy state. The transition is derived from an incoming UML transition stereotyped with <*Recovery*>, and it is always linked with the derived initial place of any simple state *"P_init_<stateName>"*.

In all transformation rules we considered the local fault occurrences and the fault activation for every state with an activity. For instance, Figure 5.1(b) shows the translation of a simple state with a *do* activity that is executed locally and it does not return any value. This activity may be executed successfully or exceptions may be raised during the execution that will change the component's behavior from normal to erroneous, as it propagates to the caller and then to the connected components. To capture this semantic, we add two timed transitions connected to the initial place (*P_init_<stateName>*), which represents the entrance to the simple state. The first timed transition (*T_do_<methodName>*) captures the correct execution of the activity, while the other timed transition (*T_localFaultOccurrences_<methodName>*) captures the fault occurrences of the *do* activity. When the first one is enabled, it will gain a token from the

initial place. For instance, if the fault occurrence is enabled first, then the token will be moved to the faulty place, which means that the request of that activity will fail.

According to CeBAM, *GaSetp* and *DaStep* stereotypes are applied to *do* activities to annotate them with the processing demand and fault activation occurrences rate. We map the *hostDemand* attribute of *GaSetp* to specify the processing time of the *do* activity. In other words, the rate of the *T_do_<methodName>* is mapped to the value of the *hostDemand* attribute. The transition *T_localFaultOccurrences_<methodName>* representes the fault activation occurrence rate of the *do* activity and its rates are mapped to the *occurrenceRate* attribute of the *DaStep* stereotypes. For *entry* and *exit* activities, we follow the same rule in mapping the attributes of the MARTE and DAM profiles.

Figure 5.1(c) is similar to the previous case, but it has a new interface place for sending the return value or completion execution signal to the component port and then to the caller component. Moreover, in both cases we have three SRN interface transitions that help to connect the SRN subnet derived from the simple state to other model elements. According to the software model, we compose the derived SRN subnets of the simple state and its outgoing transitions by matching the SRN interface transitions (t_*out*_<*CallEventName*> or *t*_<*SignalEventName*>). Similarly, the local fault activation interface transition is composed with a matched erroneous interface transition according to the value of <*sourceOperation*> and <*methodName*>.

Figure 5.1(b) to (e) and Figure 5.1(g, h, k) present all possible transformation rules of a simple state with *do* activity, depending on whether the *do* activity is executed locally or it calls for a remote method (e.g., other component's provided service). Moreover, we considered the call type when a return value is required, treating

differently synchronous and asynchronous requests. In all situations, we capture the fault occurrences, fault activation, and fault propagation transitions via the interface transitions. As mentioned earlier, the *do* activity does not have a run-to-completion semantic, therefore, all interface transitions (*t_out_<CallEventName>)* will be connected to the input place of the *do* activity. This SRN interface transition represents an UML outgoing transition with a trigger event of *CallEvent* type. Note that all failure prorogation transitions have a higher SRN priority than other transitions.

Software components interact with each other in two styles: synchronous and asynchronous. Figure 5.1(d, e) show these two styles, respectively. The *do* activity in these cases is modeled as an immediate transition, which represents the calling of a remote method (provided service) implemented by another component. As mentioned earlier, an internal behavior state machine will send/receive messages to/from other connected components only through its port. For instance, in Figure 5.1(d) we have different interface places (shaded and striped places): 1) *P_send_<serviceName>* for sending the request; 2) *P_request_<returnParameter>* for receiving the replay from the called method; and 3) *P_request_externalFailurePropagation_<affectedOperation>* for cases where the called method fails during its execution and the failure is propagated to the caller. All such interface places are used to connect the component port behavior to the internal behavior during the composition of the SRN subnets derived from BSM and PSM. In fact, each place named *P_request* represents the state machine event pool for each component provided service or failure propagation.

Figure 5.1(f) shows a simple state with *entry* action. The *entry* activity, according to CeBAM modeling guidelines, is always a local method that may raise an exception

during the execution. In order to model the normal execution and local fault activation semantics, we added two timed transitions connected to the initial place, which represent the entrance to the state. One represents the execution time for the method, and the other the fault occurrence. This transformation rule of the entry action does not violate the "run-to-completion semantic", since the transition that represents the method execution is not interruptible and we do not model the acceptance of a new event dispatched from the event pool. Indeed, the fault activation during the action execution is not a new event, but belongs to the execution path of that method. As a result, the *out* transition (t_*out*_<*CallEventName*>) is connected to the place that represents the end of entry behavior execution (*P_finish*_<*methodName*>). On the other hand, for the *do* activity in Figure 5.1(b) to (e) we have a *t_out*_<*CallEventName*> transition connected to the initial place to represent the fact that the *do* activity may be interrupted during the execution by other events.

Figure 5.1(g) shows the transformation rule for a state that has an *entry* activity executed locally and followed by a *do* activity that sends service request calls to other component-provided services. The transformation of a simple state with *entry* action and *do* activities both executed locally is captured in Figure 5.1(h). In fact, these transformation rules are a combination of the rules in Figure 5.1(f and e) and Figure 5.1(f and b), respectively.

Figure 5.1(i) shows the transformation rule of a simple state with only an *exit* action. According to the UML standard, the *exit* action will be executed before exiting the current state and after receiving the external transition event. To implement this semantic in the SRN derivation we add a new place called *P_exit*_<*methodName*>, which is the

output place of the *out* transition (*t_out_*<*CallEvent*>) obtained from the UML outgoing transition. Since the *exit* action, according to CeBAM modeling guidelines, is a local method execution, we represent the normal execution and fault activation in the same manner as the *do* activity and *entry* actions. However, both timed transitions that represent normal execution and fault activation are linked to the *P_exit*<*methodName*> and *P_init_*<*stateName*>. The *T_exit_*<*methodName*> will always be linked to the initial place of the SRN subnet derived from the target state that is shown as a striped place. Similarly, Figure 5.1(j) shows the transformation of a simple state with *entry* and *exit* activity while Figure 5.1(k) shows the transformation of a simple state that has all activities (*entry*, *do*, and *exit*). Note that we do not show all the possible combinations of a simple state with its own activities and actions. However, for these cases we follow the same mapping semantics as in the actual transformation engine implementation presented in Chapter 6 that cover all possible cases.

Error state and failure mode state represent the dependability threats of each operation in the component behavior and they show the fault activated and propagated as well as how it may be recovered. According to CeBAM, these states do not have an *entry/exit* action or *do* activity. Figure 5.1(l) and Figure 5.1(m) show the transformation of these states. An Error state is connected with two interface transitions. One transition represents the propagation to other error or failure mode manifestation, while the second one models the recovery transition. These transitions are timed to represent the delay for error propagation and recovery action. The input interface transition of a failure mode in Figure 5.1(m) is connected with the interface place named

(*P_send_propagation_<affectedOperation>*) that represented the failure propagation from a component internal behavior to its port.

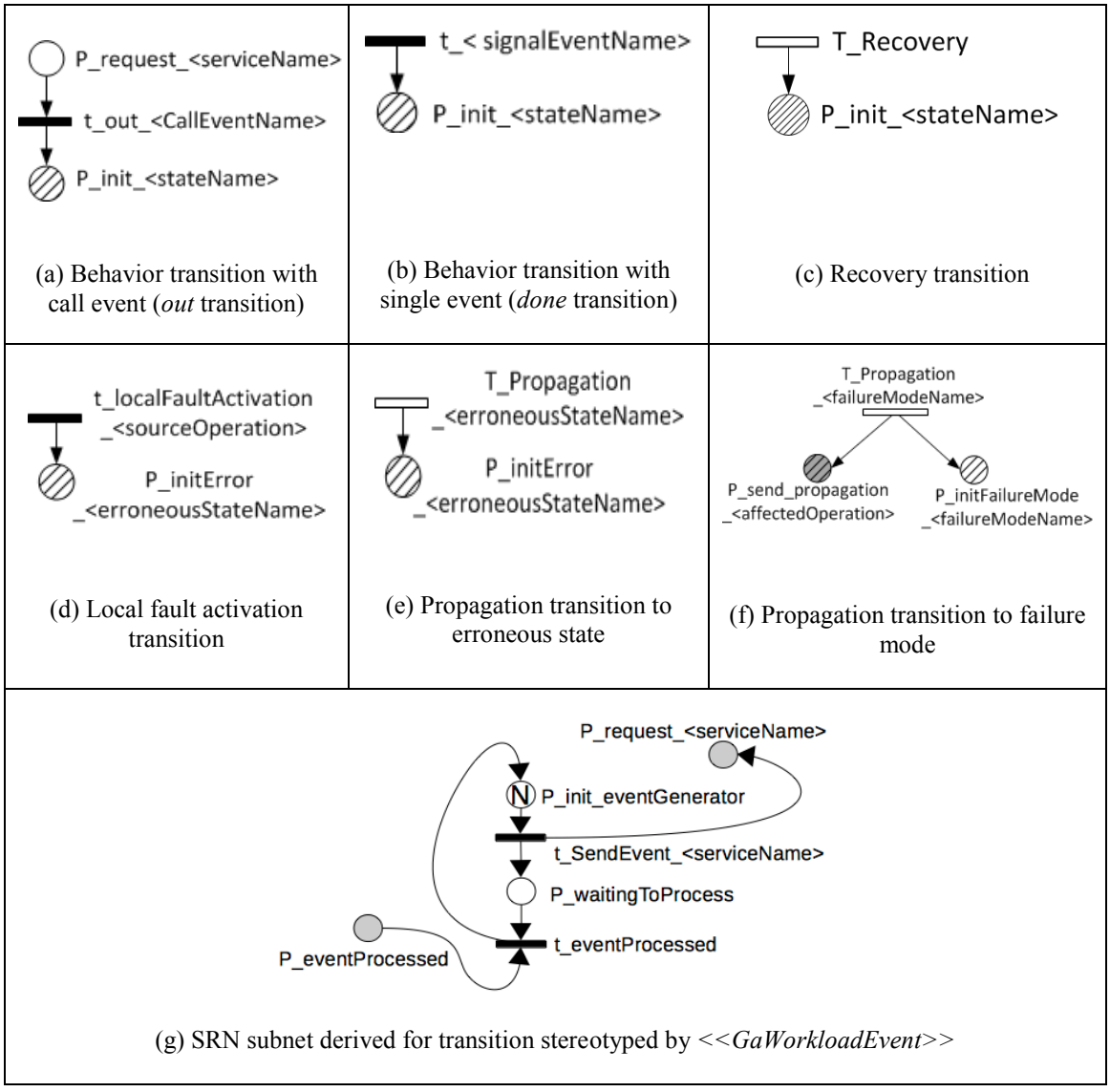
Finally, Figure 5.1(n) presents the transformation rules for initial *Pseudostates* states into a place with a token, attached to the interface SRN transition (*t_create*) that represents the creation of a new instance of the corresponding behavior. This SRN transition is transformed from the outgoing UML transition of the initial state, and it has two outgoing places. The first place represents the derived initial place of the target state while the other place represents the derived operational place of a component port. This composition captures the semantic of starting-up internal component behavior and its port at the same time.

5.1.2 Transforming Behavioral State Machine Transitions

In the CeBAM approach, the BSM transitions of do not have any actions due to the refactoring aspect that moves all transition actions to a new intermediate state. Figure 5.2 shows all possible transformation cases of BSM transitions. Such a transition is composed with the source state by an interface transition that matches its name and label. The initial places of the target state are shown as interface places in the transformation rules. Figure 5.2 illustrates that, in all cases, a state machine transition should only be connected to the initial place of the target state (the striped place in the figure). The *DaStep* stereotype is applied to the UML propagation transition that is transformed into a SRN timed transition, as shown in Figure 5.2(e and f). The rate of the derived transition is mapped to the occurrence rate value of the *failure* attribute.

Figure 5.2(g) shows a special transformation rule for the UML transition stereotyped by <<*GaWorkloadEvent*>>. This subnet captured the closed pattern of the

workload generator. Place *P_init_eventGenerator* may contain any number of requests that initiate the first request of a critical scenario. The output transition of this place was connected to the *P_request_<serviceName>* place in the internal component behavior that represented the event pool of the first request in the scenario. The request will be processed sequentially, therefore no new request will be sent until the *P_eventProcessed* place receives a token from the last activity in the scenario.



(a) Behavior transition with call event (*out* transition)

(b) Behavior transition with single event (*done* transition)

(c) Recovery transition

(d) Local fault activation transition

(e) Propagation transition to erroneous state

(f) Propagation transition to failure mode

(g) SRN subnet derived for transition stereotyped by <<*GaWorkloadEvent*>>

Figure 5.2 Transformation Rules for Behavioral State Machine Transitions

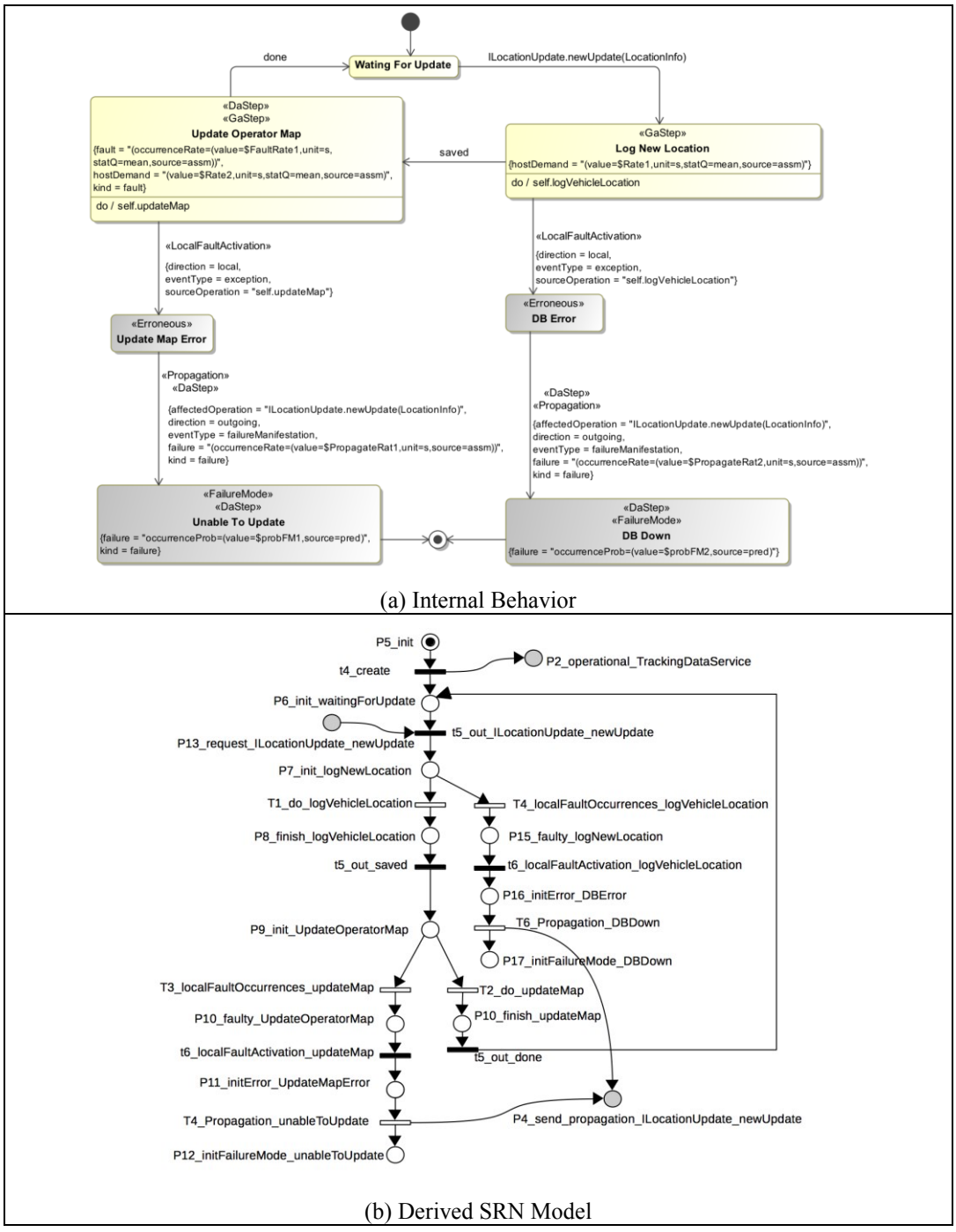
5.1.3 Composing Derived SRN subnets of Simple State and Transitions

For the internal behavior of the *Tracking Data Service* component from the VTS case study (see Figure 5.3(a)), we applied the transformation rules explained earlier in order to get the corresponding SRN model, as shown in Figure 5.3(b). The translation to SRN was performed in an iterative fashion. We begin by transforming each state and transition separately, then use matching interface places and transitions to connect the states and transition according to the main software model. In Chapter 6 we explain the implementation of the transformation rules presented in this chapter.

A software component will be in one state at any point in time, as it processes the requests sequentially. In other words, the UML State Machine property to be in one state $s \in S$ at a time leads by construction to the corresponding SRN property that a request token will be in one place $p \in P$ at a time, where P is the set of places derived from S. Therefore, the generated SRN subnet of a service execution path is 1-bounded. The place *P13_request_ILocationUpdate_newUpdate* is representing the event pool of the provided service and it is K-bounded. Each token represents one service request and these requests will be processed sequentially.

According to the internal component behavior, we have only one service execution path and it has two possible failures modes (*Unable to Update* and *DB Down*). For each service request in the event pool place, one of these failures may manifest or it will be processed correctly and get back to the initial place to process the next request. Since both failure modes belong to the same service that was provided, they will be shown as a single failure mode in the component port. We call this semantics failure mode encapsulation. In other words, all internal failure modes that belong to the same service

are propagated to just one failure mode in the component port, which is visible to the other components and propagated to a dependent component. Such semantics follow the CeBAM modeling guidelines presented in Chapter 4.



(a) Internal Behavior

(b) Derived SRN Model

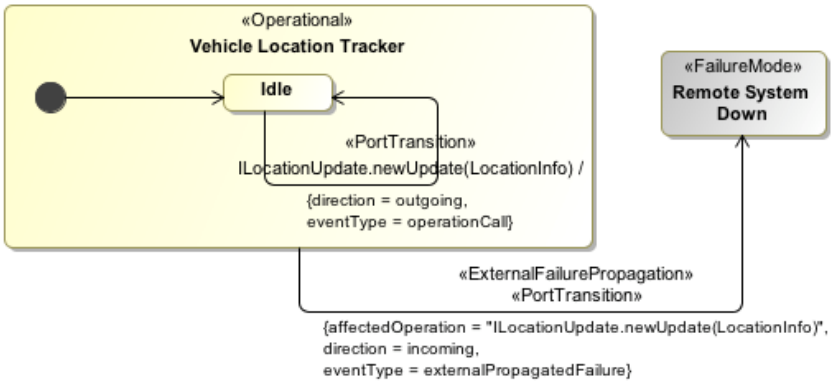
Figure 5.3: Derivation of SRN Model from *Tracking Data Service* Component

5.2 Transformation Rules for Extended Protocol State Machine

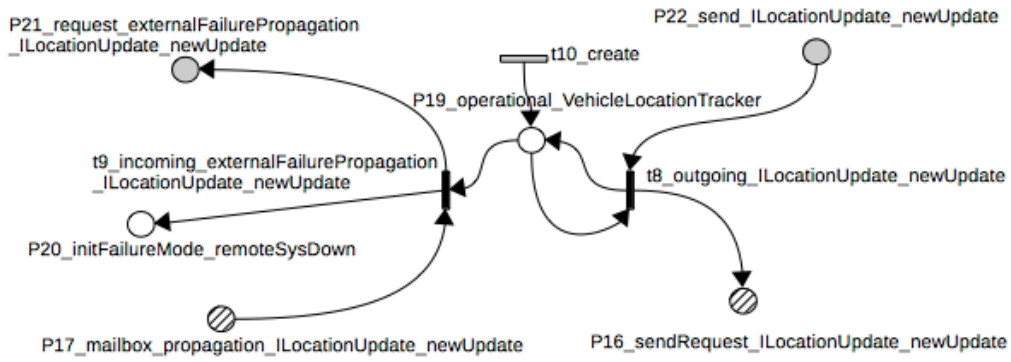
A UML protocol state machine (PSM) is a specialized behavioral state machine [3]. What makes a PSM different is that its states do not have any *entry/exit* action or a *do* activity. In addition, the transitions do not have any action; instead each has a *pre* and a *post* condition. On the other hand, composite states and concurrent regions are permitted, but history pseudo-sates are not. Usually, a protocol state machine describes the legal usage of the corresponding classifier, showing which operations may be called in which states and under which conditions.

In our previous work [116] and in Chapter 4, we identified the limitations of the PSM and extended it to model the component's port behavior. Thus, in the CeBAM approach, PSM describes the external view of the component, by specifying incoming and outgoing messages through each port. This extension helped us to model the failure propagation between the component interfaces along with their normal behavior. A composite state stereotyped by <<*Operational*>> is used to model the port normal protocol behavior. It has all the component public states as well as incoming and outgoing service requests. The failure of incoming or outgoing service requests are modeled as outgoing protocol transitions from the composite state, as shown in Figure 5.4(a). Note that PSM in our approach does not show any behavioral implementation.

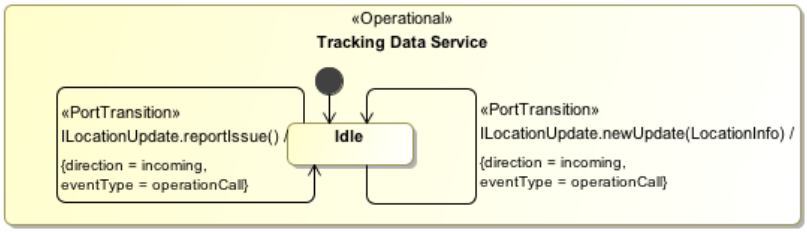
All transitions of the extended PSM are atomic transitions and they possess "run-to-completion-semantics." Moreover, any message (service request or failure propagation) reaching the component port will be passed immediately to the internal behavior for incoming messages or to other component ports for outgoing messages. In other words, these messages are not cached in the component port.



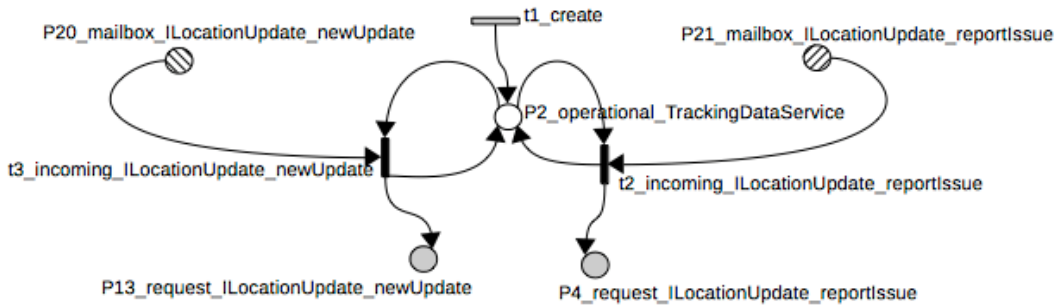
(a) *RLocationUpdate* port behavior of *Vehicle Location Tracker* component



(b) Transformed SRN with interface places of *RLocationUpdate* port behavior of *Vehicle Location Tracker* component



(c) *PLocationUpdate* port behavior of *Tracking Data Service* component



(d) Transformed SRN with interface places of *PLocationUpdate* port behavior of *Tracking Data Service* component

Figure 5.4: Applying PSM Transformation Rules on *RLocationUpdate* and *PLocationUpdate* Ports

Three steps must be followed to derive an SRN subnet from a port protocol behavior modeled according to CeBAM to preserve its semantics. First, the composite state is transformed into a single SRN place named *P_operational_<stateName>* and each failure mode protocol state is transformed to an SRN place named *P_initFailureMode_<failureModeName>*.

Second, each protocol transition included in a composite state stereotyped with *<<Operational>>* is transformed to an SRN immediate transition named *t_outgoing_<ServiceName>* or *t_incoming_<serviceName>* according to the protocol transition direction. These transitions represent the normal service request (incoming or outgoing). Each of the derived transitions is linked to the derived operational place (*P_operational_<stateName>*) as an input and output place. Moreover, these transitions are linked to the interface places to connect the port SRN subnet with the internal behavior SRN subnet and with the other port SRN subnets.

The last step is transforming protocol transitions representing failure propagation into SRN immediate transitions linked to derived operational place as input place and to derived failure mode place as output place. These transitions have a higher SRN priority and are connected to interface places similar to the derived transitions representing normal service requests. The operational place has no token in the initial marking. It is connected to the component internal behavior via a *t_create* transition. As explained in the previous sections, this SRN transition is derived from the outgoing UML transition of the UML initial pseudostate and is linked to the initial SRN place of the target state, as well as to *P_operational_<stateName>* to model the synchronized start-up semantic of the component internal behavior and its ports. In case of any failure propagated to the

port, the token in the operational place will be moved to the failure mode place, and no more requests can be passed through the component port.

Figure 5.4 shows the application of these rules to the *RLocationUpdate* and *PLocationUpdate* ports of two components in the VTS case study. Note that we show the interface places that will be used to compose the port's behavior model with its internal component behavior (shaded places) and this port with other component ports (stripped places).

5.3 Component Behavior Composition Patterns

5.3.1 Composition of Component Internal and its Ports Behaviors

A port is the property of a classifier (i.e., a component) that represents a distinct interaction point between the classifier's internal behavior and its environment [3]. Two types of ports are defined in UML: service ports and behavioral ports. A behavioral port possesses the implementation of its classifier, which is not externally visible. In our approach, we limited ourselves to service ports, which lack implementation. We use extended PSM to capture the component's external visible behavior through its stateful ports. In fact, the port PSM model precisely captures both incoming and outgoing messages for normal and erroneous messages. These messages represent how the components interact and communicate with each other, as well as the legal usage of the interfaces attached to the port.

According to [3], once the instance of a classifier is created, a new instance of its ports will also be created along with its specific interaction points. A link from the port instance to the owning classifier instance will be created in order to forward any incoming requests from the environment to the owning classifier instance, or to send the

outgoing requests from the classifier's internal instance to the environment (to other connected components).

In composing the derived SRN subnets of the PSM with the BSM for normal behavior, we apply this semantic by adding an interface place between the two subnets. In fact, each provided service modeled in the port's PSM must have a corresponding transition in the internal component's behavior (BSM) with the same event name; otherwise the incoming request will not be handled properly. For a required service to become implemented in the component providing it, each activity in the BSM that calls for another's component service must have a corresponding transition in the PSM with the same event name; otherwise this request is not passed to the connected component. These conditions must hold for all communication types (i.e., service calls, signals, and failure propagation).

Figure 5.5 shows how we compose the SRN subnet derived from the component BSM with the SRN subnet derived from its port PSM for the normal behavior by using only interface places (gray places). As explained earlier, the PSM captures the legal usage of the interfaces (provide/required) attached to the component's ports and it does not contain any implementation. A provided service is implemented as part of the internal behavior of the component providing it. According to CeBAM, if a fault is activated in the internal component behavior, it will be propagated to the interface's implementation causing a new error type and failure mode. Accordingly, this propagation will be shown in the port's PSM as a failure mode. In other words, any internal failure that does not get recovered will be propagated to the component's ports and then to the connected components. In Chapter 4, the VTS case study shows how to model a failure manifested

in the internal behavior as it propagates to the port and to other dependent components, causing a new error and failure mode.

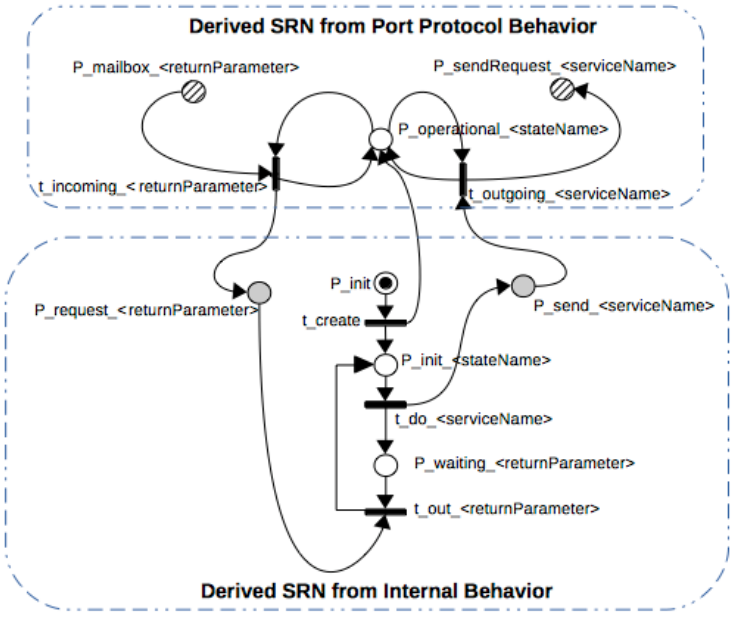


Figure 5.5: Compositing Component Derived SRN Subnets from BSM and PSM for Normal Behavior

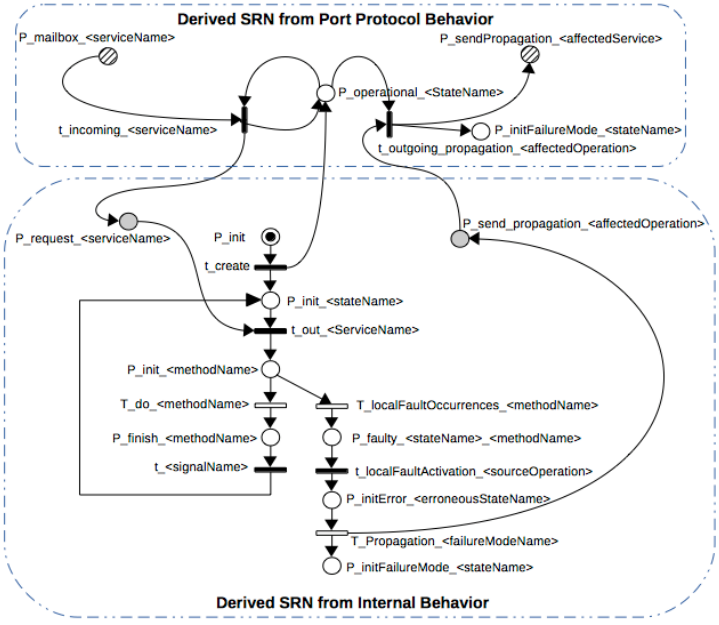


Figure 5.6: Composition Pattern for Outgoing Failure Propagation from BSM to PSM

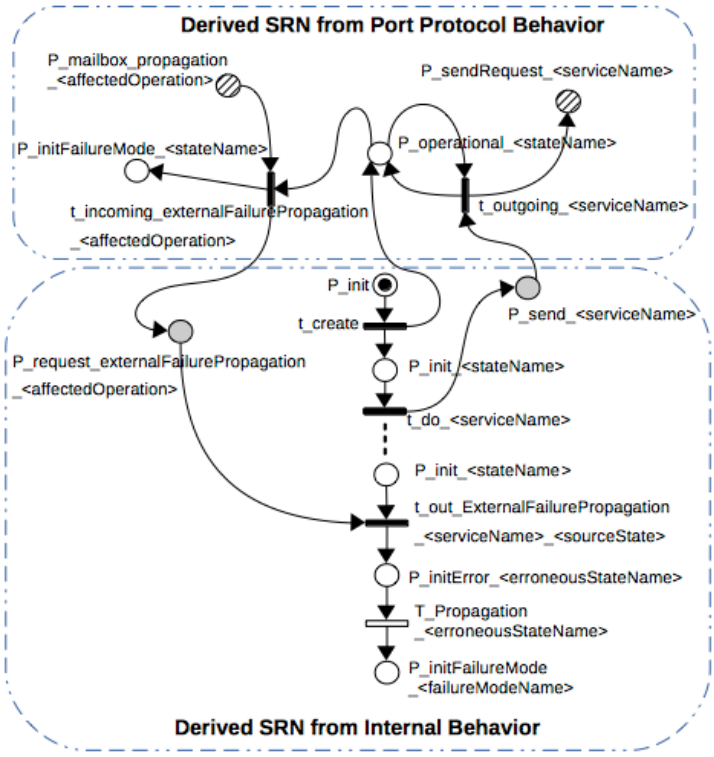


Figure 5.7: Composition Pattern for Incoming Failure Propagation from PSM to BSM

In order to show the error propagation from BSM to PSM in the transformation rules, we use an SRN interface place that connects the failure propagation transition in the BSM-derived SRN subnet to the outgoing failure propagation transition in the PSM-derived SRN subnet (see Figure 5.6). According to CeBAM, any internal propagation transition whose direction attribute value is *outgoing* is linked to the interface place (*P_send_propagation_<affectedOperation>*) as an output place. We use the affected operation attribute value to match and connect the interface place with the outgoing port transition. For an external propagated failure from PSM to BSM, we use a similar composition pattern, as shown in Figure 5.7. An incoming propagation failure will be passed to the component internal behavior via an interface place named *P_request_externalFailurePropagation_<affectedOperation>*. This place is connected to

the matching external failure propagation transitions in the internal behavior SRN subnet which starts the internal erroneous behavior.

5.3.2 Inter-component Composition

The communication between different UML state machine instances is handled by an underlying event pool, and it is usually not modeled [3]. A caller object (i.e., component instance) can call an operation of another object by sending an event (i.e., a service request). An implicit event pool of the called object receives this event. At a later point in time, the event is dispatched from the event pool to the state machine where it either triggers a transition or it may be discarded. Actually, UML 2 does not specify the semantics of the selection criteria or the priority of the event from the event pool.

In our transformation rules, an event pool is considered for each provided service or incoming messages. We create a place *P_mailBox_<serviceName>* attached to each transition representing an incoming operation call. A *P_sendRequest_<serviceName>* place is attached to each of the outgoing transitions that represent the buffering of the outgoing requests, before passing them on to the connected port. As shown in Figure 5.8, in order to connect provided and required service or failure propagation between two components deployed on different nods, we needed to construct an intermediate SRN subnet consisting of a send request place for the outgoing requests and a mailbox place for the incoming requests. These two places are then connected by a *T_sendRequest_<serviceName>* timed transition, which represents the network connection delay. Moreover, we add a timed transition (*T_lost*) from these places to represent the loss of the request due to the communication channel. The rates of these timed transitions are derived from *<<GaCommHost>>* and *<<DaConnector>>*

stereotypes respectively as shown in Figure 4.1(a). However, we assume that the connection channel is recoverable; therefore, if the event was lost it will be sent again. In fact, this transition may be replaced with a subnet that models the availability of the network connection. However, if both components are deployed on the same node then this transition is replaced with an SRN immediate transition and *T_lost* is removed, as we assume that the event will never be lost and is delivered immediately. This composition pattern is shown in Figure 5.9.

According to [1], if an internal fault is activated but not properly handled inside the component, then this fault will end up in a failure mode, which will be propagated to its corresponding port and to the other components that depend on it. To model the failure propagation between components, we needed to add an SRN intermediate subnet that links the failure mode of the provided interface to the connected required interface in a similar way as for the service request. As a result, for every failure mode of the provided service, we must have an incoming transition on the required interface that models the external failure propagation. This transition causes a new error state and failure mode for the caller component. Figure 5.8 shows the composition of the provided service failure mode with the external propagated failure transition of the required interface, by adding an intermediate SRN subnet consisting of a timed transition (*T_sendPropagation_<affectedOperation>*) to model the propagation delay between the components and the failure place (*P_mailBox_propagation_<affectedOperation>*). Failure propagation events could be lost due to the communication channel. To model this, we have added a timed transition connected to the mailbox place. Figure 5.10 shows the composition pattern for failure propagation to multiple components.

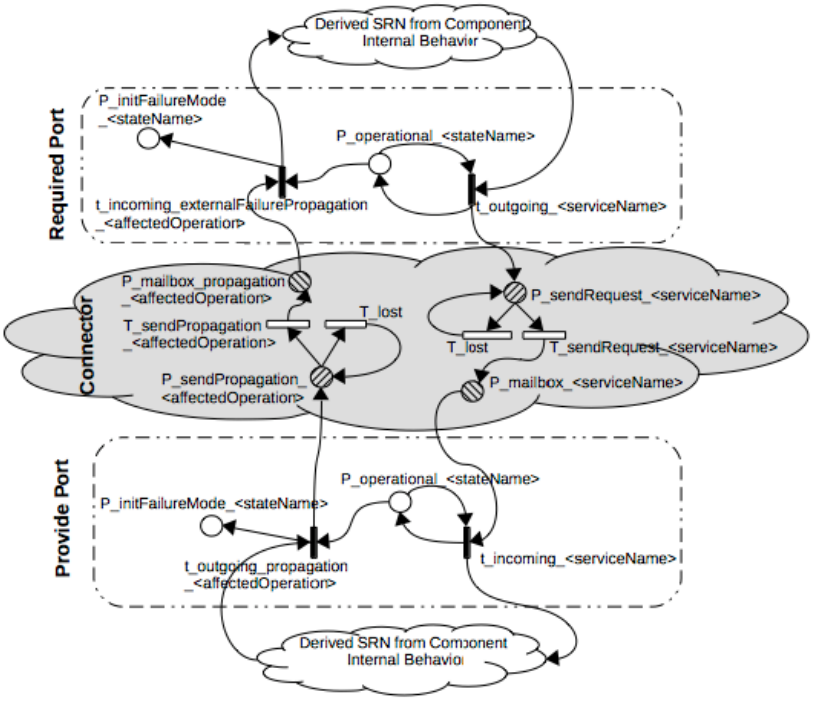


Figure 5.8: Service Request Composition Between two Components Deployed on Different Nodes

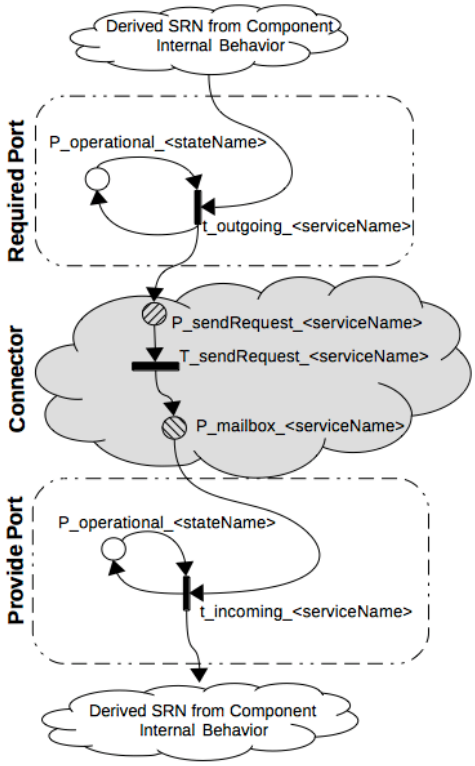


Figure 5.9: Service Request Composition Between two Components Deployed on the Same Node

Figure 5.11 shows that the provided service by one component is connected to more than one required interface of different components. For each incoming PSM transition there is only one *P_mailBox_<serviceName>* interface place for either the signals or for the provided service. On the other hand, for the connected components, each outgoing transition for the signals or the operation call must have a corresponding *T_sendRequest_< serviceName>* transition and *P_sendRequest_<serviceName>* place.

Another pattern for the synchronous communication style that waits for the return value is modeled in Figure 5.12. In this composition style, a two-connector SRN subnet is linked to the *P_sendRequest_<returnParameter>* place of the provider port. The *T_sendRequest_<serviceName>* transition has the SRN guard used to identify the waiting component by checking the marking of *P_waiting_<returnParameter>* place in the SRN subnet derived from the component's BSM.

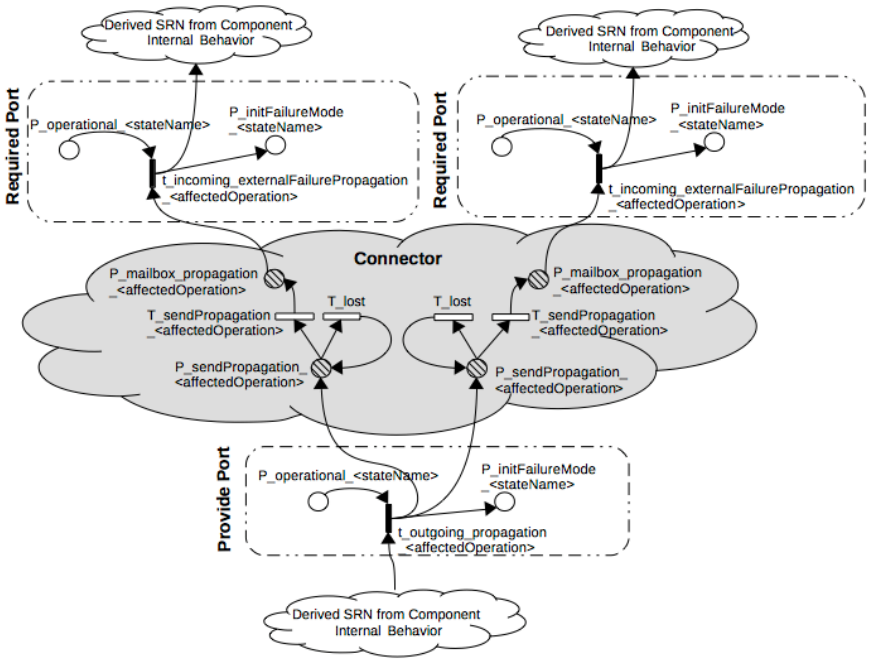


Figure 5.10: Failure Propagation between One Provider Component Interface and Multi Connected Required Interfaces

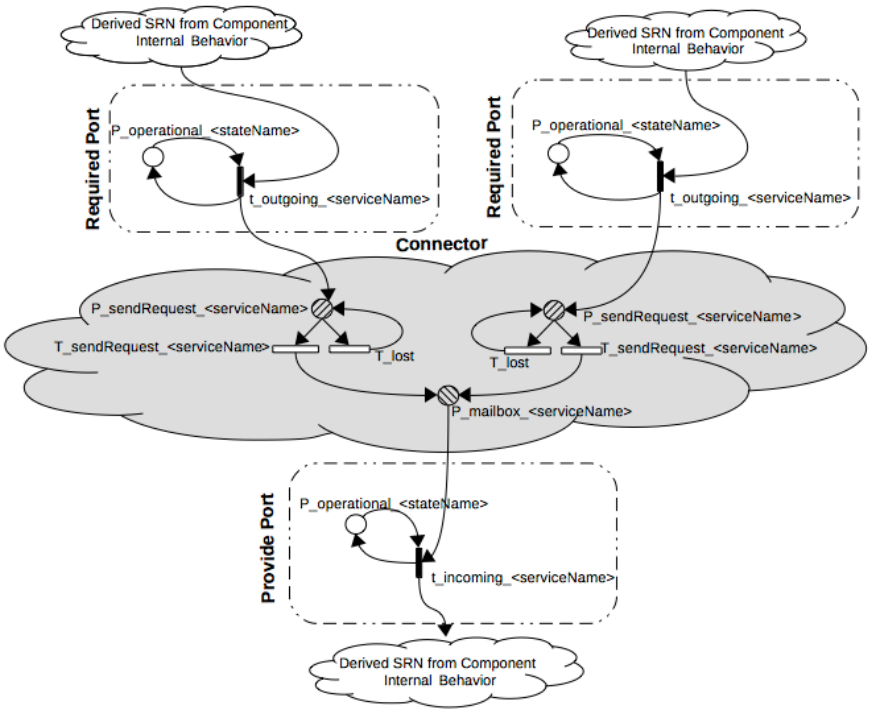


Figure 5.11: Service Request Composition between One Provider's Component and Multi-Required Required Interfaces

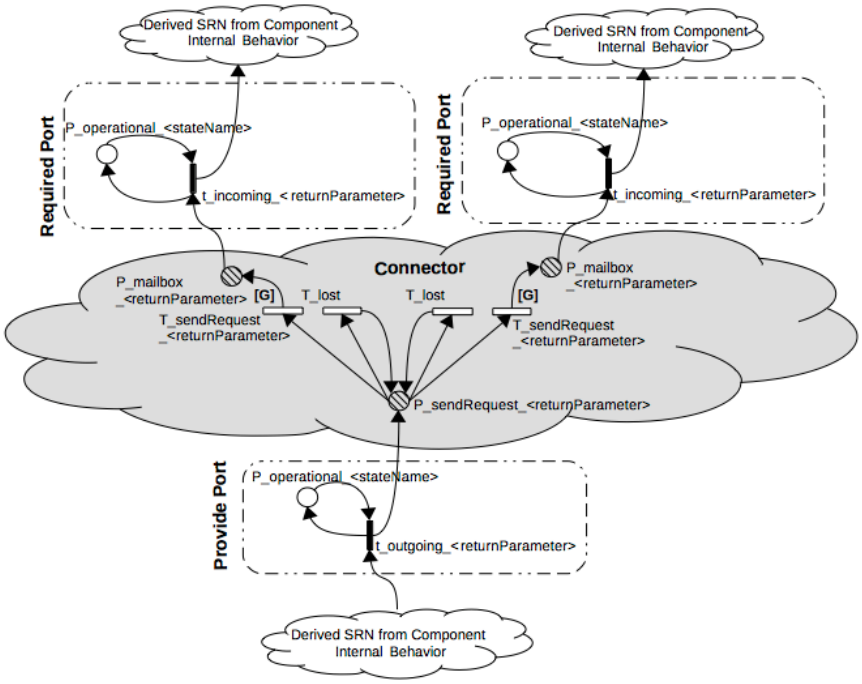


Figure 5.12: Single Provider and Many Required Interfaces Communicating Synchronously

5.4 Conformance and Compatibility Verification

In the CBD paradigm, a component is a logical unit of abstraction with well-defined interfaces that provide or require services from its environment [114]. In our approach, we assume that the port is the interaction point and that it has the needed interfaces to be integrated with other system components. Interfaces consist of a set of public services that may be called by other components in the system through their required interfaces. A call may be synchronous, blocking the caller until it receives a reply, or asynchronous, starting a new thread. In all connection styles a received call is processed by the internal component behavior since the interface has no implementation.

In order to analyze the reliability or availability, we need to verify the conformance and compatibility of the involved components. In conformance verification we check if a component's internal behavior state machine conforms to its ports' protocol state machines. In other words, we check whether or not each incoming message to a component port has a corresponding transition in the internal component behavior that can handle the message. We also verify whether or not an outgoing event from the internal component behavior is captured by the port behavior. Such events may describe normal, erroneous or propagation behavior.

Compatibility deals with the communication between components in the system. For two connected components, the provided services must be compatible with the requested services. In fact, using extended PSM as proposed in CeBAM helps us to model how the system components interact with one another, by capturing the passing of messages through their ports in a specific order.

UML [3] considers the conformance of the protocol state machine through the *ProtocolConformance* model element. It explicitly assures that a specific state machine conforms to a generic one. Thus, component realization must conform to the component interfaces. Unfortunately, the conformance definition in the UML standard is limited and there is no clear framework for reasoning about it and for verifying it. As a result, there are many approaches in the literature [121], [127]-[129] that attempt to address the problem of how to verify component conformance and compatibility.

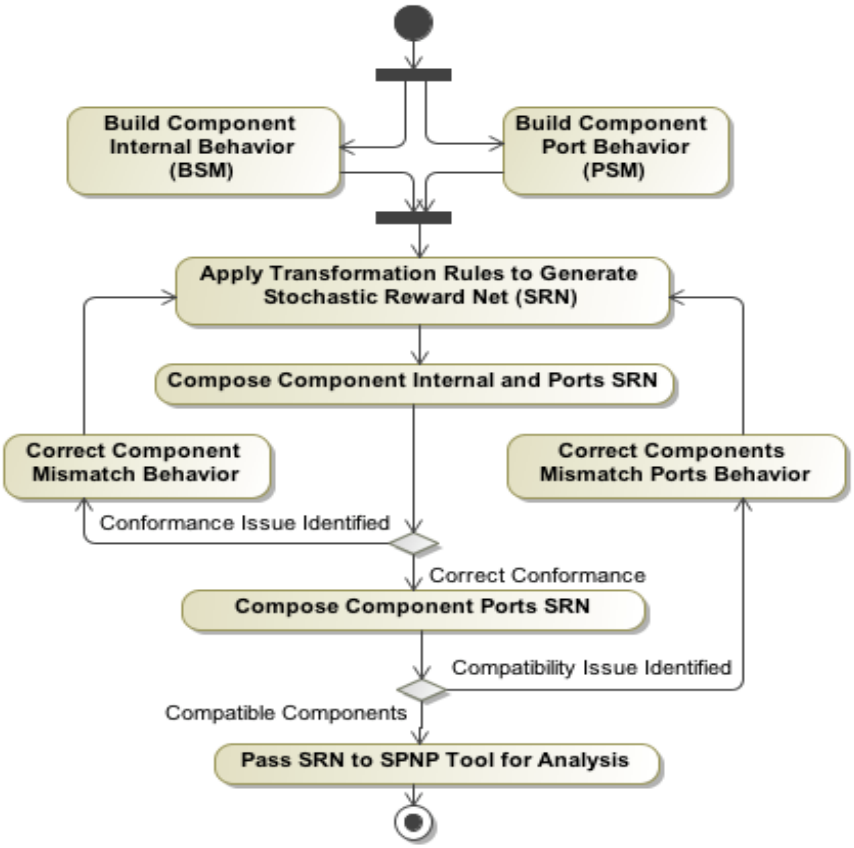


Figure 5.13: Conformance and Compatibility Verification Activities

In our dependability prediction framework we followed state-based analysis methods based on SRN, so we use the same formal model for conformance and compatibility verification. The verification phase should resolve a number of issues in the

software model before starting the reliability/availability analysis phase. For instance, we need to be certain that any manifested internal failure inside a component will be propagated to all connected components and that the model is free of deadlocks. We identify conformance or compatibility issues during the construction and composition of the SRN subnets obtained by transformation from the software model. The activity diagram of Figure 5.13 describes the verification process.

For each component in the model, we apply the transformation rules to its BSM and PSM models, which describe its internal and ports behaviors, respectively. The next step is to compose them. This composition includes normal as well as erroneous behavior, which connects incoming and outgoing messages from the ports to its internal behavioral model. During this step, we may identify some conformance issues that must be fixed in the main software model before continuing to the next step. For example, in the PSM model we may have incoming messages representing external failures propagated from another connected component, even though the effects of this message are not modeled in the component internal behavior model. Consequently, the internal behavior does not conform to its port, and this mismatch must be repaired.

Once each component internal behavior conforms to its ports behaviors, we may start the next step, which deals with the inter-component compatibility. For the connected components, we compose their SRN port behavior models to identify incompatibilities between components for the provided and required services and for failure propagation. For instance, if there is a service failure modeled in one component's port, it may not necessarily be modeled in a connected component. This type of incompatibility must be resolved in the main software model by verifying the conformance again in that

component before completing the compatibility verification, as is shown in Figure 5.13. This verification approach is implemented in the *sm2srn* transformation engine. Users are provided with descriptive messages that help in fixing all identified conformance and compatibility modeling errors. Chapters 6 and 7 have more details on the transformation implementation and testing of SRN subnets derivation and composition.

5.4.1 Conformance and Compatibility Verification Examples

As already explained, the purpose of conformance and compatibility verification is to identify the mismatch between the connected components, their internal behavior, and their ports. This is a mandatory step in our dependability analysis approach that helps to fix all components mismatch that impact the reliability and availability of the system. The verification process is performed in two phases. First, during the construction of the analysis model (SRN) we identify dangling model elements, which are not connected to other model elements. These non-connected elements may represent the failure propagation not modeled in the connected component, or a service incorrectly used. Second, once the SRN analysis model was constructed, we study its properties, such as deadlocks and dead transitions. The SPNP solver is used in this phase. By interpreting the results from these two phases, we trace back the location of the mismatch and fix it in the software model, and then run the verification process again until we get a correct model that may be used for the reliability analysis.

Figure 5.14 shows two examples of conformance modeling errors. The first is a conformance issue when an incoming message is modeled in the component port (PSM), but there is no corresponding event in the component's internal BSM. Then the message will not be handled (possibly lost) and will keep the requester waiting forever if the call

style is synchronous. The *T_incoming_ILocationUpdate_reportIssue* is modeled in the *PLocationUpdate* port of *Tracking Data Service* component, and it is connected to the event pool place (*P_request_ILocationUpdate_reportIssue*). However, this place is dangling, since no corresponding SRN transition exist in the BSM-derived SRN subnet. Such a modeling error will prevent the processing of all incoming requests for this service. The dangling place may involve absorbing markings in the derived SRN model.

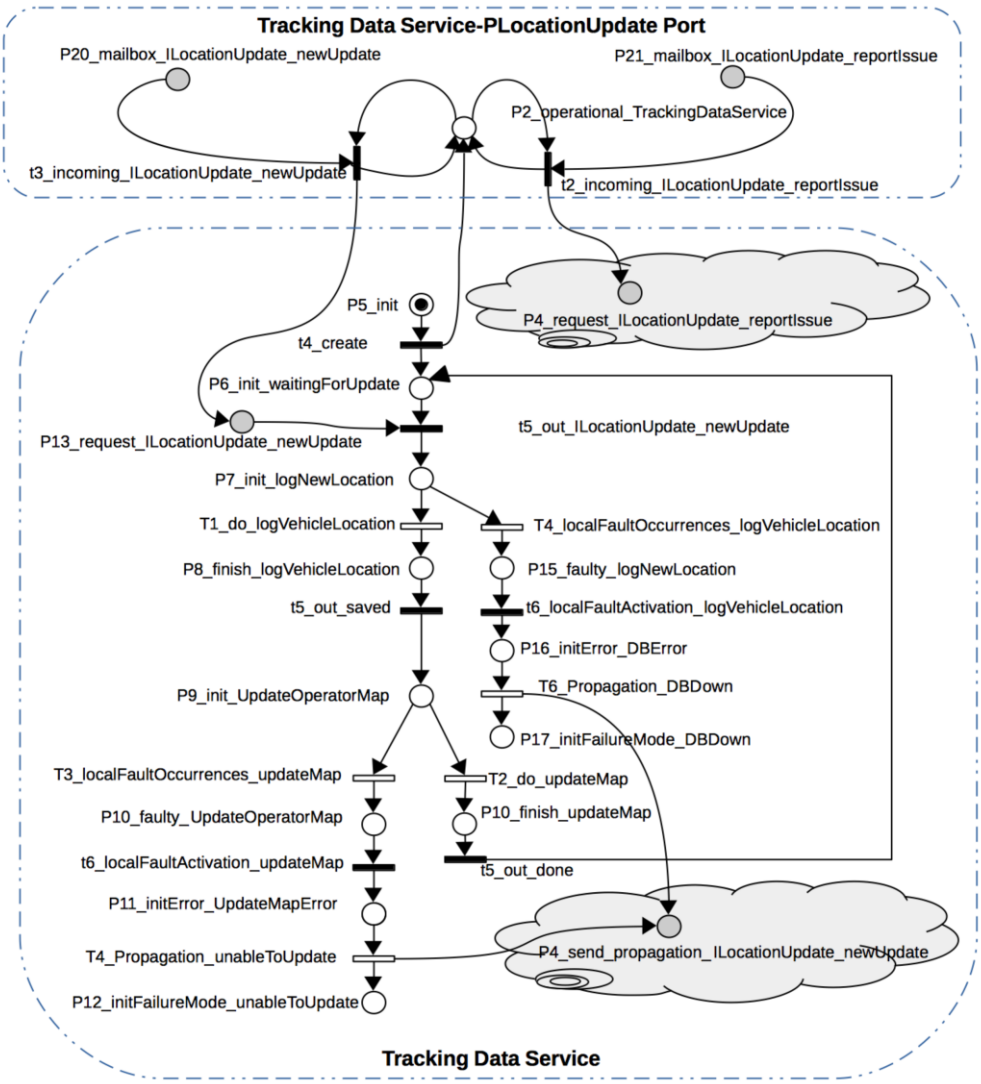


Figure 5.14: Example of Conformances Issues: dangling provided service and failure propagation

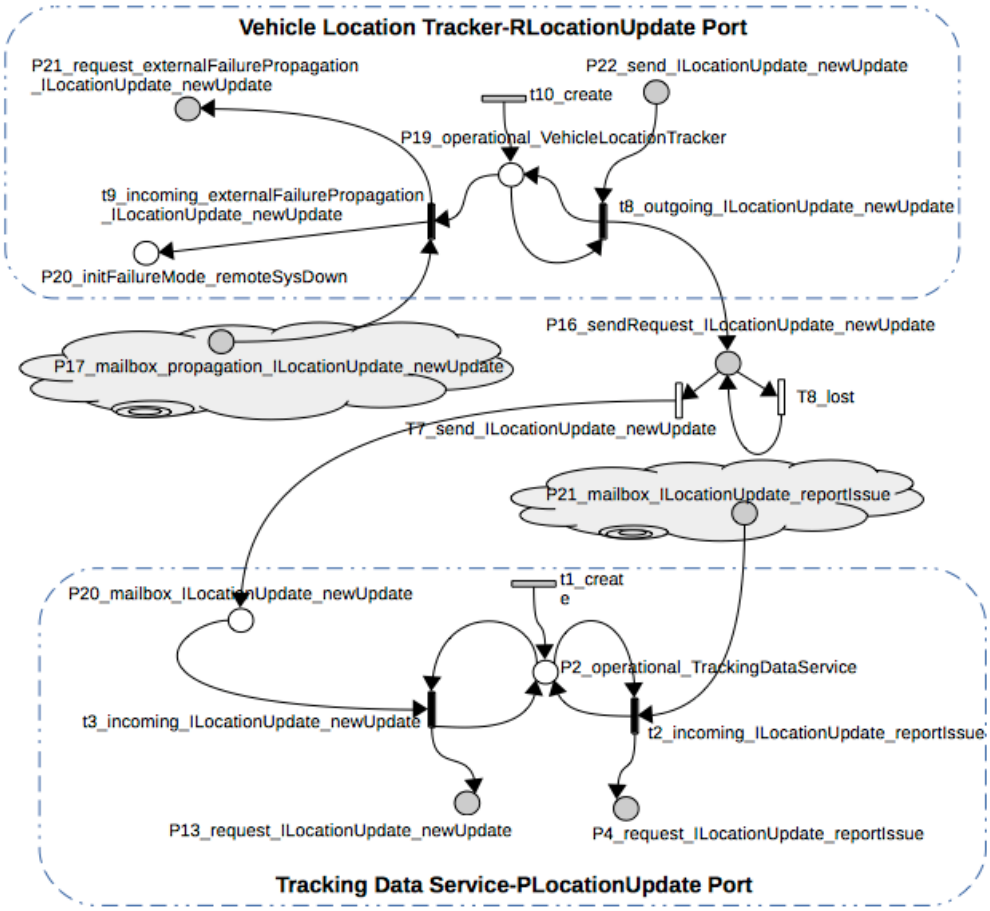


Figure 5.15: Example of Components Compatibility Issues: dangling provided service and failure propagation

The second conformance example is related to the propagation of a manifested internal failure mode to the exit port and then to all other dependent components. In Figure 5.14, none of the failure modes for the provided service (*ILocaiotnUpdate_newUpdate*) are propagated to the port. In fact, the outgoing port transition that is supposed to receive the internal failure propagation is not represented in the port PSM (*PLocationUpdate*). This conformance modeling error is identified by the dangling interface place (*P_send_propagation_ILocationUpdate_newUpdate*).

For compatibility verification, Figure 5.15 shows the generated SRN model from the *PLocationUpdate* port of a *Tracking Data Service* component wired with the *RLocationUpdate* port of the *Vehicle Location Tracker* component. During construction, we identify two incompatibility issues.

First, a dangling place (*P17_mailbox_propagation_ILocationUpdate_newUpdate*) indicates that the external failure propagation modeled in the *RLocationUpdate* port is not connected to the *PLocationUpdate* port. As shown, this port does not capture the failure model manifestation. Such a model is incorrect, having an incompatibility between these ports. The user must fix this modeling error in the original UML model.

Second, the *ILocationUpdate_reportIssue* provided service in the *PLocationUpdate* port of the *Tracking Data service* component is not linked to the connected component. This non-connected provided service might not be needed within this context, or the required interface does not model correctly its use. The mismatch must be fixed in the software model before starting reliability analysis.

Chapter 6: Architecture and Process of Model Transformation Chain

The MDA approach contributes to the reduction of the gap between software models and analysis models. Model transformations techniques, in particular, are employed for the derivation of the analysis models from annotated software models [49], [96]. The work in this chapter is an example of how to use standard model transformation techniques for automating the generation of SRN analysis models from annotated UML software models. In Chapter 5 we presented the design of the transformation rules from UML state machines to SRN and the SRN subnets composition mechanisms. In this chapter, we present the actual implementation in QVT-O of the transformation rules and composition mechanisms proposed in Chapter 5.

Section 6.1 provides an overview of the transformation architecture and implementation. The next section shows the developed transformation chain and how transformation engines work together in a sequential process. Sections 6.3, 6.4, 6.5, and 6.6 include a detailed description of all the transformation engines developed to generate CSPL code from the annotated UML software model. Instead of presenting all detailed transformation algorithms, we use graphical representations to show a high-level overview of each transformation class. This graphical representation includes main information about each transformation class such as input/output models and key transformation rules. For each transformation engine, we show the architecture and the actual QVT-O code of the main transformation class that invokes other classes in the transformation chain.

6.1 Transformation Architecture and Implementation Overview

This chapter provides a detailed description of our design and implementation of the model transformation chain. The first transformation engine, *CebamWeaver*, is the implementation of the CeBAM approach. This engine is weaving context specific aspect models to internal component behavior and its port protocol behavior. Using composed QVT-O transformation, it starts by applying refactoring aspects models to the internal component behavior and then weaves erroneous aspect model with the refactored model. Also, erroneous aspect models are applied to each component port protocol behavior. This transformation engine verifies the context aspect models to identify any violation of the CeBAM modeling guidelines.

The next transformation engine, *sm2srn,* derives the SRN model in a series of transformation tasks. It starts by transforming the internal component behavior and its ports protocol behavior into SRN subnets according to the transformation rules presented in Chapter 5. The next task is to compose the SRN subnets for each component from its internal behavior and port protocol behaviors. During this task, conformance is verified to identify any incoming or outgoing messages (normal or erroneous) that are not handled properly. The last task is to compose a single SRN model that represents a critical scenario subnet from the derived SRN subnets for all the collaborating components. Compatibility is verified during the composition process, to identify any modeling error such non propagated failures or requested but non provided services. In addition, the implementation has a verification mechanism that checks the correctness of the input model to identify any violation of the CeBAM modeling guidelines.

The derived SRN model describes the structure of the subnet along with the marking dependent expressions such as reward rate and transition firing rates. However, the final specification of the SRN model accepted by the existing SPNP solver is textually defined by the CSPL language [113]. Thus, to fully automate the process of generating a solvable analysis models, a model-to-text transformation is required to generate CSPL code from the derived SRN model. However, due to the rather large gap between the graphical SRN model and the textual CSPL code, we create first an intermediate model-to-model transformation that generates a graphical CSPL model from the SRN model, which in turn is transformed into CSPL text. This approach reduces the complexity of generating CSPL code from the SRN model. The final generated code is executed by the SPNP solver to compute reliability measures.

This transformation chain has three model-to-model transformations implemented in QVT-O and one model-to-text transformation implemented in MTL. Both transformation languages are OMG standards [90], [130]. For QVT-O, we used a series of open-source Eclipse implementations, which evolved during the thesis research. We started with the Eclipse Indigo version and kept migrating our transformation code to a new platform every time a new QVT-O version was released. The current version of our transformation chain implementation is working under QVT-O version 3.4, which is part of Eclipse Luna. For model-to-text, we use Acceleo which implements OMG MTL [130] and became part of the Eclipse modeling project. For UML models, we use MagicDraw professional edition [131] for developing all the required UML models and profiles such as CeBAM, DAM and MARTE. One of the advantages of this tool is a clean and readable representation of applied stereotypes. To execute the model transformation

chain, we export all the models developed according to CeBAM guidelines from the MagicDraw tool to the Eclipse environment.

Although the Eclipse implementation of QVT-O was partial and some functionality presented in the OMG standard was missing, we managed to implement all transformation rules presented in Chapter 5. Our implementation evolved and was enhanced with every Eclipse release, by including recent implemented QVT-O features, such as transformation composition and extension. In fact, during the development of this thesis, we discovered some bugs and communicated them to the developers, such as the application of UML stereotypes and the setting their values [132]-[134]. Despite all tools bugs, we managed through different iterations after each release to construct transformation engines that follow the best practices and realize our objectives in this thesis.

6.2 Transformation Chain Process

The transformation chain process is carried out by four transformation engines, as shown in Figure 6.1. The first transformation engine is the *CebamWeaver* that implements the CeBAM approach presented in Chapter 4. For each component in a critical scenario, the *CebamWeaver* transformation engine starts by applying the context specific refactor aspect to the internal component behavior and then weaves the context-specific erroneous aspect with the refactored model. It also weaves context specific erroneous aspect to component's port protocol behavior. The output of this transformation is a complete component internal behavior and complete port protocol behaviors for all components involved in the critical scenario.

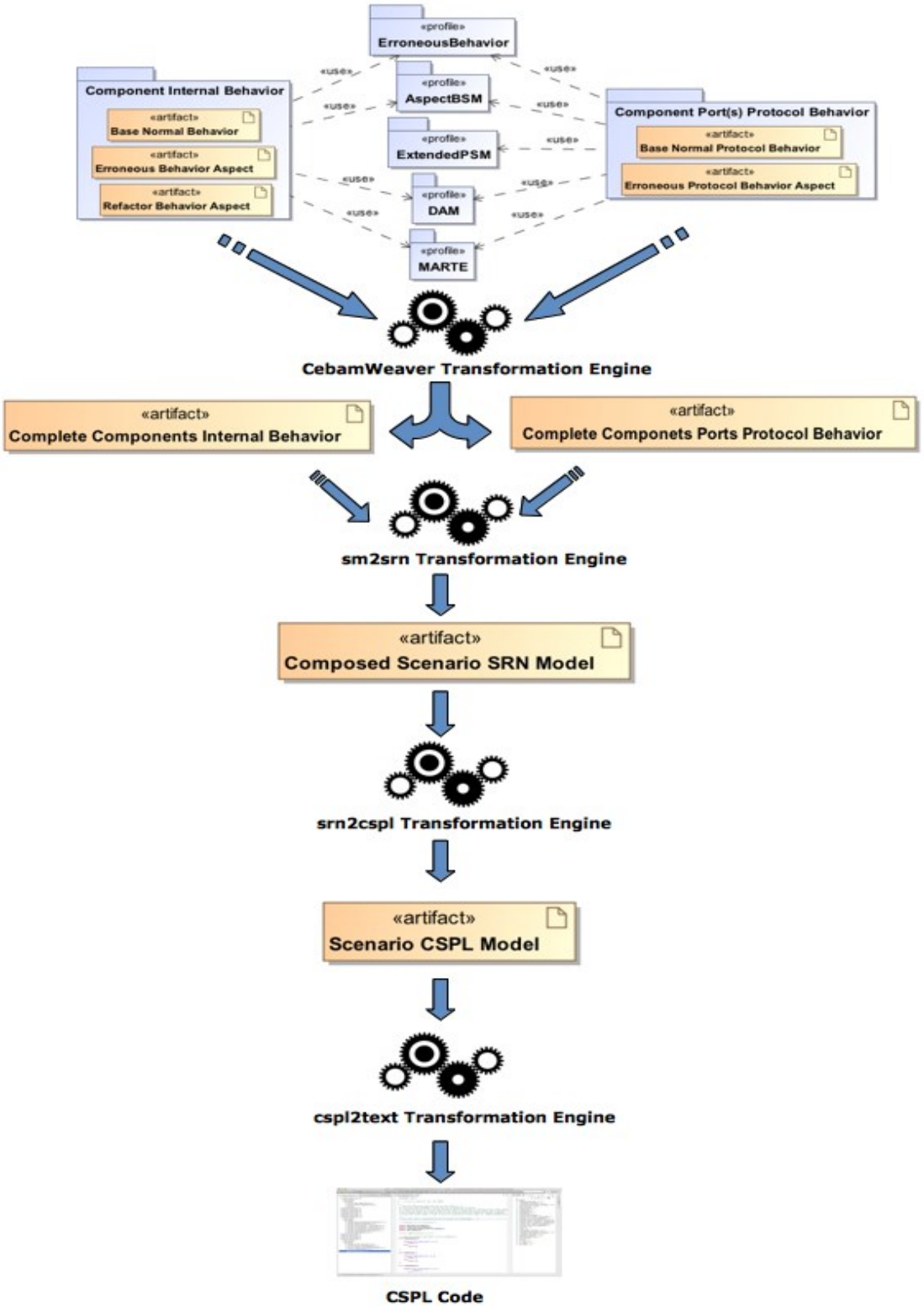


Figure 6.1: Process of Model Transformation Chain

A verification mechanism that uses QVT-O uses assertions (expressed as constraints) to verify the correctness of context specific aspect models that must conform to CeBAM modeling guidelines.

The models obtained from the *CebamWeaver* transformation engine will be used as input models to the *sm2srn* transformation engine, the second in the chain that derives the SRN model according to the transformation rules defined in Chapter 5. This transformation engine takes all components internal and port behavioral models to derive SRN model for each component and then composes from them a critical scenario SRN model. During composition, the components' conformance and compatibility will be verified. Moreover, the *sm2srn* transformation engine will check the input model to identify any violation of the CeBAM modeling guidelines. Section 6.4 presents the architecture and transformation details.

As mentioned in Chapter 2, CSPL is the textual language used to completely describe the SRN model, and to modify the configuration settings and analysis parameters of the SPNP solver. CSPL has many features that are not captured in the SRN metamodel (see Appendix A). To reduce the complexity of generating CSPL code from the derived SRN model (that represents only the structure of the SRN model), we defined another intermediate metamodel that captures the CSPL language features as shown in Appendix A. The main advantage of developing this metamodel is to have an intermediate level transformation between two different abstractions. This helps to reduce the complexity of generating the textual code from the derived SRN model. As a result, we defined a third model-to-model transformation called *srn2cspl* that derives the CSPL

model and creates all other mandatory functions. It also creates all configuration settings and analysis parameters required by SPNP solver.

The last transformation engine is *cspl2text*. This model-to-text transformation will generate textual file conforming to CSPL language. This transformation is generic, and does not have any complex transformation rules, since the CSPL metamodel is describing rather closely the textual language. However, this transformation could be complex and hard if we just directly generate the CSPL code from the derived SRN model. We followed the best practices by generating the textual code from a metamodel that closely describes the textual language.

The following sections describe the architecture of each transformation engine as well as the transformation algorithms. Due to a large number of QVT-O transformation rules, helpers and queries, we will show graphically the main transformation rules and how they relate to each other. This graphical representation has complete information about each transformation class, such as the involved models and extended transformation class, as well as and QVT-O black-box plugins.

6.3 Phase One: CeBAM Approach Implementation Architecture

CebamWeaver is an In-Place transformation which updates the input UML behavioral models instead of creating new target models. According to the CeBAM approach described in Chapter 4, it updates the internal component behavior and port protocol behavior models to derive a complete model that has both normal and erroneous behavior. For each component, this transformation will start refactoring its internal behavior model by weaving context specific refactor aspect models and then applying context specific erroneous aspect model to the refactored model. Also, this

transformation will weave context specific erroneous aspect models into the port protocol behavior.

Figure 6.2 presents the *CebamWeaver* transformation engine architecture. It is a composed transformation that consists of two core transformation classes: *CebamWeaver.qvto* and *PsmErroneousComposition.qvto*. Both transformations import *CebamLibrary.qvto,* which includes common helpers and queries. Moreover, this library class contains also customized implementations of some operations that are not implemented in the Eclipse QVT-O implementation, such as operations to deal with stereotyped UML model elements.

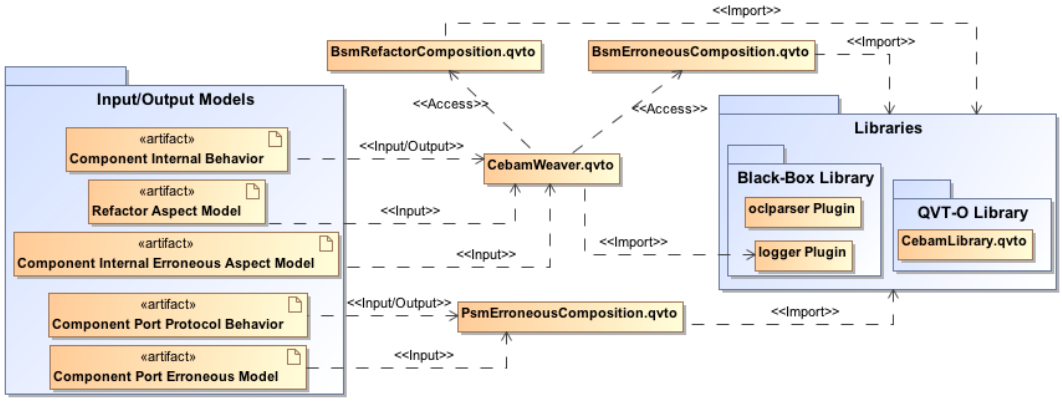


Figure 6.2: *CebamWeaver* Transformation Engine Architecture

Modifying component behavior (internal and port protocol) is performed based on the result of the *pointCut* OCL query that identifies the join point(s) in the base model. As explained in Chapter 4, this query is entered by the software modeler as a string attribute of the aspect model. During the In-Place transformation execution, the *pointCut* query string is parsed and executed on the base model to weave automatically the aspect models with the selected model element (join point). To implement this feature, we use the Eclipse implementation of OCL and QVT-O black-box mechanism. Eclipse modeling

platform provides OCL standalone configuration that allows invoking their OCL implementation from Java code or JUnit to parse an OCL query string and execute it on the input model [135]. The black-box mechanism in QVT-O allows a transformation to call arbitrary external code, i.e., a Java operation. We developed a Java-based plugin called *edu.cu.naif.thesis.oclparser* that has all the required configurations of Eclipse OCL pivot standalone. The extension of this plugin is configured according to the QVT-O black-box mechanism. This plugin has several methods that can be invoked from the QVT-O transformation code. An example is a method accepting two input parameters: a UML state machine model and an OCL query string. It starts by parsing the query string and then executes it on the passed UML State Machine model and returns the result as a set of model elements.

In order to produce an external text log file that logs the transformation progress and lists all identified modeling errors during the transformation, we developed another plugin called *edu.cu.naif.thesis.logger*. Again the QVT-O black-box mechanism is used here to call a Java operation that accepts three string parameters: log file path, log file name and logging message. This feature helps the user to review all the transformation progress and read all the possible modeling errors from an external text log file. Moreover, this logging mechanism will be useful if the transformation engine is called from another application. We use this logger plugin in all the implemented transformation engines.

As shown in Figure 6.2, both In-Place core transformations *CebamWeaver.qvto* and *PsmErroneousComposition.qvto* use a black-box library. The first usage is for identifying join points in the base model. A *PointCut* query string is passed along with a base model

to *oclparser* plugin to parse the query and execute it on the base model. The results are returned as a set of join point elements in the base model. This result is then used by the respective QVT-O transformation to apply the aspect model to each selected join point element. The second usage is for logging the transformation progress as well as other error messages in an external text file, as defined in the transformation code. During the development, we defined many logging messages to support the user in getting enough information about the transformation progress. Moreover, error messages help the user to find and fix modeling errors of the input models. These messages can be further enhanced according to the needs.

A composed transformation allows for invoking and reusing other transformations, so it is a crucial feature for large and complex transformations [90]. The reuse mechanisms in QVT-O are *access* and *extension*. The former is similar to an import package in an ordinary programming language, while the latter is a combination of import and inheritance. Using the *extension* mechanism we can call specific mapping rules from the imported transformation, but in *access* we can call only the main transformation. This feature is applied to the design of all transformation engines developed in this thesis. In the first version of our CeBAM implementation, we did not use composed transformation and reuse, because these features were not supported in the early versions of QVT-O. However, after each release of QVT-O implementation by the Eclipse modeling group, we kept updating and enhancing our design by utilizing newly implemented features.

According to CeBAM, for each UML State Machine that captures internal component behavior we may have more than one context specific refactor aspect and

only one erroneous aspect. The Refactor aspects must be applied before the erroneous aspect. Our objective is to have a single transformation that takes a set of refactor aspects models and erroneous aspect model along with the component internal behavior model as the base model and then applies these aspects; the returned result is a complete component internal behavioral model. In this way, the user will run just one transformation instead of manually invoking different transformations. To realize this objective, we have to use the composed transformation feature of QVT-O that was mentioned earlier.

```
import edu.cu.naif.thesis.logger;
import CebamLibrary;
import BsmRefactorComposition;
import BsmErroneousComposition;

modeltype UML uses 'http://www.eclipse.org/uml2/2.1.0/UML';
modeltype ecore uses 'http://www.eclipse.org/emf/2002/Ecore';

/*CebamWeaver is In-Place transformation so the input model (bsm) will be same output model after
applying refactors and erroneous aspects since it belongs to the same metamodel.*/
transformation CebamWeaver(inout BsmBaseModel : UML,
                           in bsmRefactorAspectModel : UML,
                           in bsmErroneousAspectModel : UML)
        access transformation BsmRefactorComposition
        access transformation BsmErroneousComposition;

// external log file path and name
property filePath : String = "/Volumes/MyData/PhD_Thesis/Implementation/TransformationsLogs/";
property fileName : String = "CebamWeaver.txt";
property logMsg : String;

main() {
        logMsg := "CebamWeaver started";
        writeLog(filePath , fileName, logMsg);

        //Apply refactor and erroneuons aspect on component BSM
        new BsmRefactorComposition(BsmBaseModel, bsmRefactorAspectModel).transform();

        new BsmErroneousComposition(BsmBaseModel, bsmErroneousAspectModel).transform();

        logMsg := "CebamWeaver finished";
        writeLog(filePath , fileName, logMsg);
}
```

Figure 6.3: Main Mapping Operation of *CebamWeaver* Transformation Engine

Weaving refactor and erroneous aspects with the internal component behavior is performed by using *CebamWeaver.qvto* transformation. This composed transformation is accessing *BsmRefactorComposition.qvto* and *BsmErroneousComposition.qvto*

transformations, as shown in Figure 6.2. The user executes only the *CebamWeaver.qvto* transformation to weave multiple refactor and erroneous aspects models. The code captured in Figure 6.3 shows the actual implementation of this composed transformation. First step is to apply refactor aspect to the base model (*BsmBaseModel.uml*) by making a new instance from *BsmRefactorComposition.qvto* transformation and then executing it with the *transform()* operation. This transformation will apply one or multiple refactor aspects to the base model.

The second step is to weave erroneous aspect models to the refactored model resulted from the *BsmRefactorComposition.qvto* transformation. It is done by instantiating and executing the *BsmErroneousComposition.qvto* transformation. In both transformation classes, we use QVT-O assert expressions to identify any violation of CeBAM modeling guidelines. The transformation will be terminated if any breach is detected. In Chapter 7 we show examples of detecting such modeling errors and snapshots of the descriptive error message that help the user to fix detected modeling errors. The starting and ending of transformation composition and other messages are logged in an external text file using *writeLog()* operation, which invokes the *logger* block-box plugin.

The following subsections show the main mapping rules and architecture of *BsmRefactorComposition.qvto* and *BsmErroneousComposition.qvto* transformations.

6.3.1 Refactoring Component Internal Behavior

The *BsmRefactorComposition* is an In-Place transformation. As shown in Figure 6.4, the component internal behavior is passed as *inout* model parameter of type UML model to be refactored according to the *refactorAspectModel:UML* passed as input

parameter. This transformation is importing *oclparser* black-box plugin used for identifying join points elements in the base model. The transformation progress and identified modeling errors are logged in an external text file using the imported *logger* black-box plugin. Additionally, we import the *CebamLibrary* that has a set of common queries and customized implementations of QVT-O standard operations. For example, the *isStereotyped* operation checks all applied stereotypes to a selected model element returning true if a given stereotype passed as a parameter is applied, and false otherwise.

In the CeBAM approach, the modeler can define and build multiple refactor aspects models to be applied to a single component internal behavior. To apply all these refactor aspects in an automated way, the user is asked to pass only one UML model that has nested submodels for each refactor aspect model. The first step in the *BsmRefactorComposition* transformation is to parse the input from the UML model (*refactorAspectModel)* to select each aspect sub-model and copy it to an intermediate QVT-O dictionary collection. This parsing process is performed by using two helpers: *getAllRefactorAspectModels()* and *getSingleRefactorAspectModel()*. The second step is to apply all refactor aspect models stored in the intermediate dictionary collection sequentially in one invocation of the transformation.

A refactor aspect is applied to transitions with attached activities, as explained in Chapter 4, in order to preserve the run-to-completion semantic by adding a new intermediate state to execute the former transition activity as a state *do* activity. As a result, the erroneous behavior of such an activity can be weaved with the newly added intermediate state. Such a refactoring can be applied to a single or to multiple transitions, as specified in the *PointCut* query.

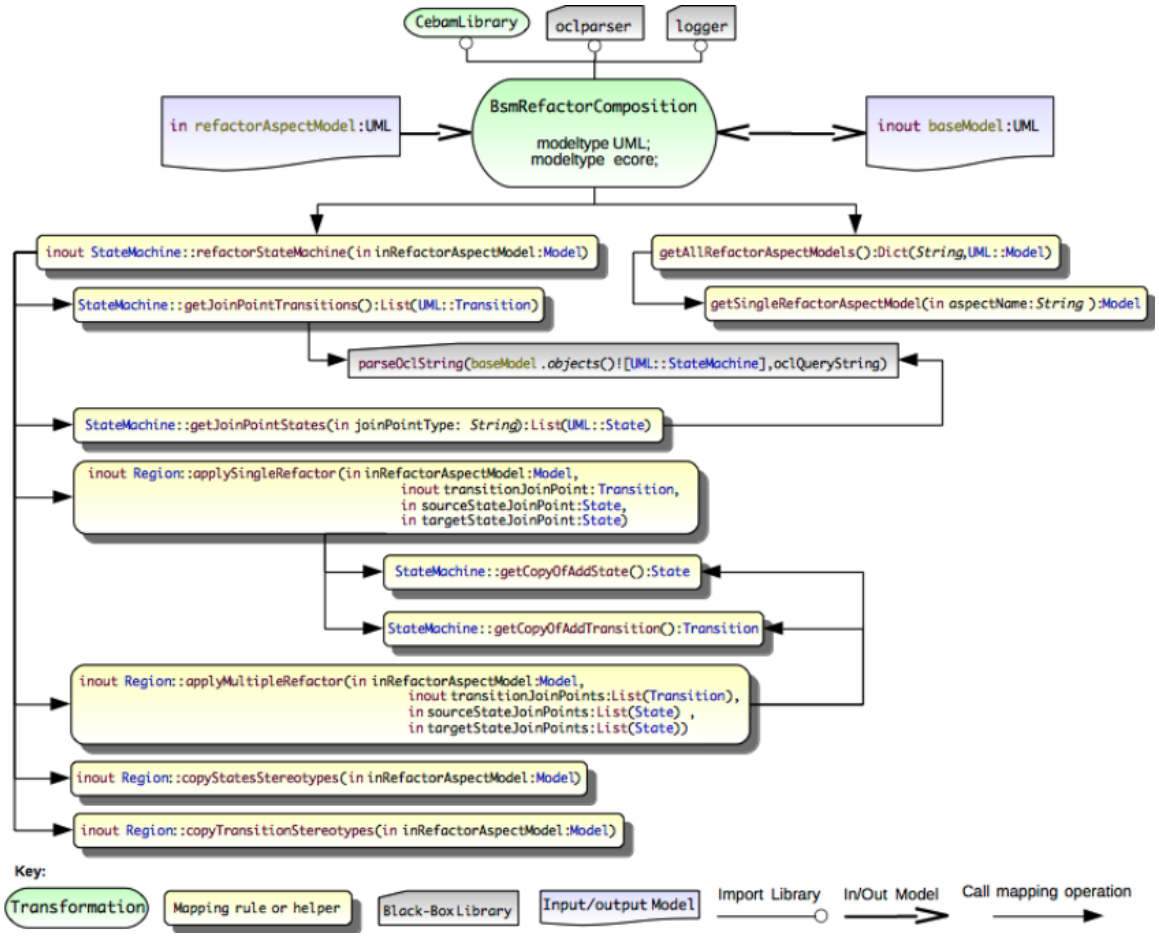


Figure 6.4: Graphical Representation of *BsmRefactorComposition* Transformation

Accordingly, the *BsmRefactorComposition* transformation will start by identifying the join points in the internal component behavior by calling the black-box operation *parseOclString*, as shown in Figure 6.4. The next step after identifying the join point is to invoke either *applySingleRefactorAspect* or *applyMultipleRefactorAspect* mapping based on the number of selected join point(s). While applying the aspect model, the transformation will use AspectBSM stereotypes as directives that guide the transformation. For instance the <<*Refactor*>> stereotype is to modify a property of the selected model element while the <<*Add*>> stereotype is to add a new model element.

QVT-O provides two standard operations called *clone()* and *deepClone()* that create a new instance copy of a model element. These two operations seem useful for copying the model element stereotyped by <<*Add*>> in the aspect model added to the base model. However, the actual implementation of these operations is not helpful in our case, since it copies not only the desired model element, but also other model elements referred by it (such as the incoming and outgoing transition property of the intermediate state stereotyped by <<*Add*>>). Another issue in using the standard copying operations is that the applied stereotypes are not included in the newly created instance of the cloned model elements. In fact, this behavior is expected because applying a profile on the model is done before applying a stereotype to the model element. For these reasons, we use explicit object creation and inline mapping operation mechanisms for weaving aspects models to the base model.

The last step in this transformation is to apply stereotypes to the newly added model elements to the base model as defined in the aspect model. As mentioned earlier, we use the inline object creation mechanism when applying a refactor aspect, therefore copying and applying stereotypes is done explicitly. It was performed by two mapping *copyStatesStereotypes* and *copyTransitionsStereotypes* that find a matching model element in the base model and explicitly apply stereotype and associated attributes by copying them from the aspect model. These two mappings select stereotypes of CeBAM, DAM, and MARTE profiles only and ignore stereotypes of the AspectBSM profile, since they are not required any more after refactoring the base model.

6.3.2 Weaving Component Erroneous Aspect

According to CeBAM, each internal component behavior has one erroneous aspect model, which may contain one or more erroneous paths. Each path is starting with UML state that is stereotyped by <<*PointCut*>> that specifies the UML state join point in the component internal behavioral model. In Chapter 4 and 7, we present examples of different erroneous models. The *BsmErroneousComposition* transformation class navigates through an erroneous aspect model passed as input parameter and applies each erroneous path sequentially to the *inout refactoredModel:UML,* as shown in Figure 6.5. It starts by identifying the join point, which is a UML state in the component behavioral state machine, by calling the *parseOclString* black-box operation. The next step is to verify the existence of an erroneous path in the base model. If it exists, then it weaves only the fault activation transition from the joint point state, otherwise it weaves the whole erroneous path.

The first step in the *BsmErroneousComposition* transformation is dealing with weaving the erroneous path without applied stereotypes. This step is performed using explicit object creation and inline mapping operation mechanisms. AspectBSM stereotypes are used as transformation directives to guide the weaving process. As explained in the previous section, we do not use QVT-O standard operations called *clone()* and *deepClone()* since it does not clone applied stereotypes and it copies referenced model elements along with the cloned element. However, if the transformation detects any violation of CeBAM modeling guidelines, the transformation is terminated and a descriptive error message is logged in the external log file. The next step is to apply stereotypes to weaved erroneous paths, as specified in the aspect model. The two

responsible mapping rules for applying stereotypes and their attributes to UML state and Transition are *copyStatesStereotypes* and *copyTransitionsStereotypes*.

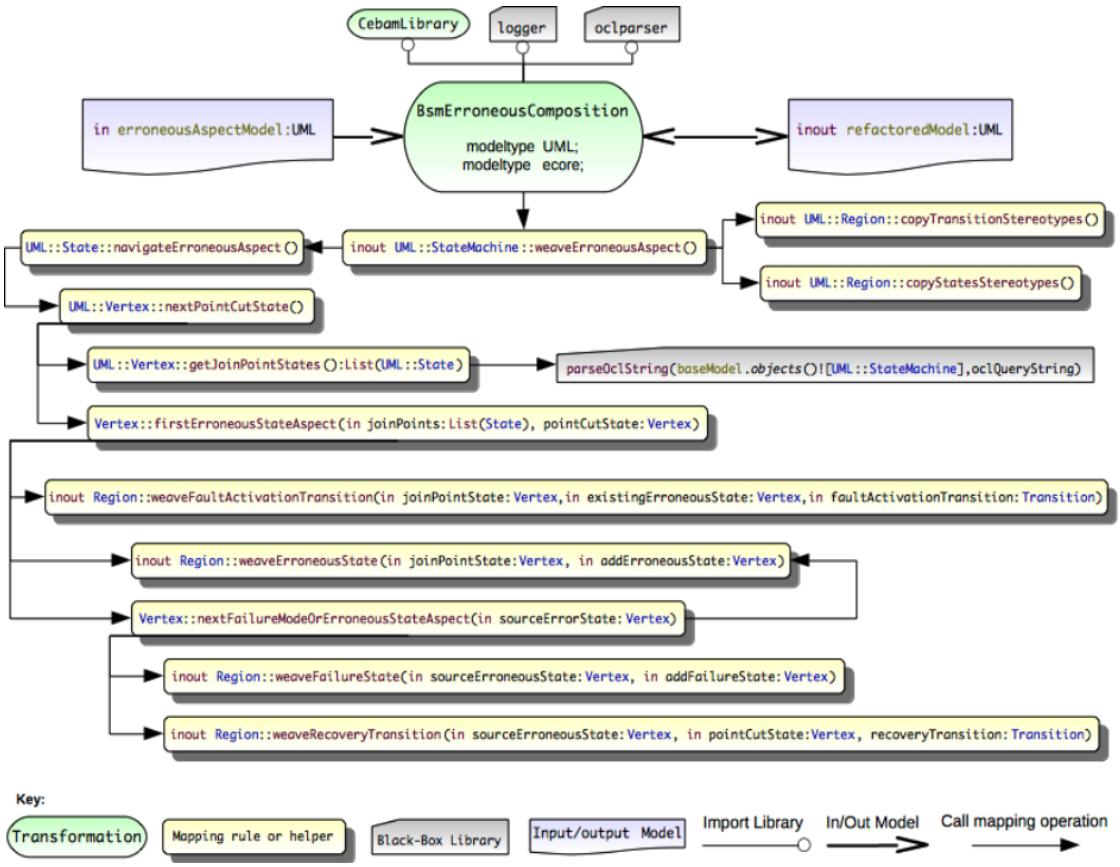


Figure 6.5: Graphical Representation for *BsmErroneousComposition* Transformation

The transformation *PsmErroneousComposition* shown in Figure 6.6 is dedicated to applying component erroneous aspect model to the port protocol behavior. According to CeBAM, this aspect model has only failure mode states to be weaved to ports protocol behavior. The join point in this case is the composite state stereotyped by <<*Operational*>>. However, it must be specified by the user in the *oclQuerey* of *PointCut* state to be then parsed and executed using *parseOclString* operation in *oclparser* black-box. The process of weaving the port erroneous aspect model and

applying stereotypes is similar to the process of *BsmErroneousComposition* transformation.

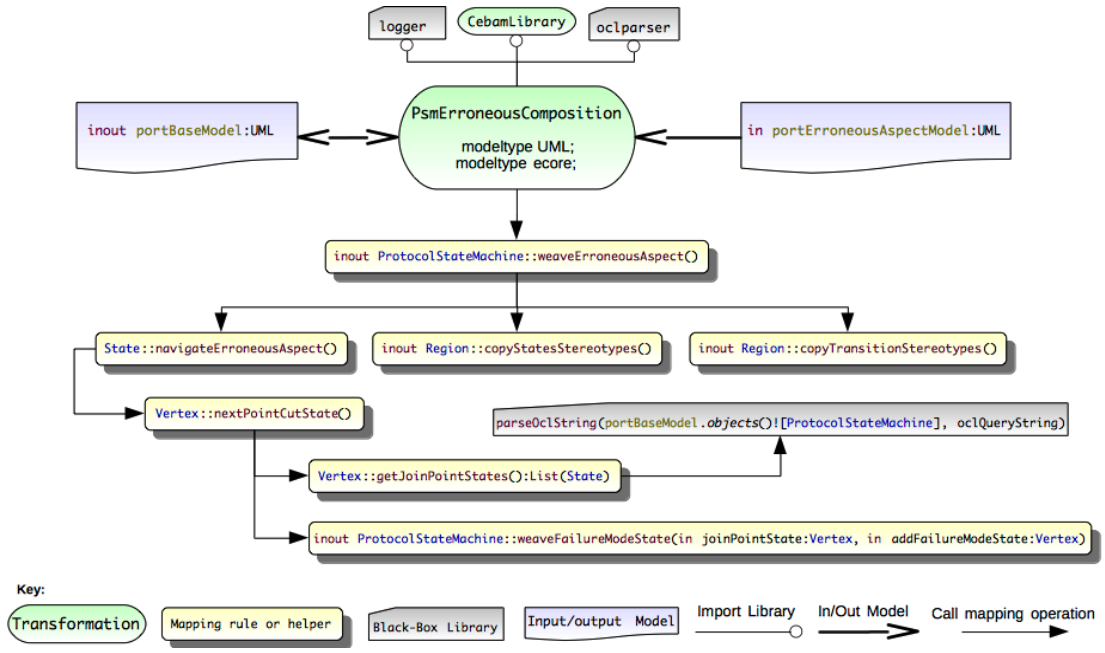


Figure 6.6: Graphical Representation for *PsmErroneousComposition* Transformation

6.4 Phase Two: Derivation of SRN model Implementation

The critical scenario under analysis consists of multiple components interacting through its ports that have provided and required services. The complete component internal behavior is describing the component normal and erroneous behavior. Whereas, the complete component port protocol behavior is capturing incoming and outgoing messages, either normal service call or failure propagation. The objective of this transformation phase is to design a generic QVT-O model transformation that takes three input models: critical scenario architectural model, all components complete internal behavior, and all components ports complete behavior. This generic transformation should first transform all behavioral models to SRN subnets. The next task is composing derived SRN subnets to single SRN model according to the architectural model as

explained in Chapter 5. Moreover, this transformation must verify the conformance and compatibility and log all modeling errors and transformation progress into an external text file.

6.4.1 Generic Composed Transformation Architecture

The architecture displayed in Figure 6.7 presents the *sm2srn* transformation engine architecture. It has three Out-Place transformation classes, *srnComposition.qvto*, *bsm2srn.qvto*, and *psm2srn.qvto*. (An out-place transformation has a read-only source model and creates a target model according to the mapping rules).

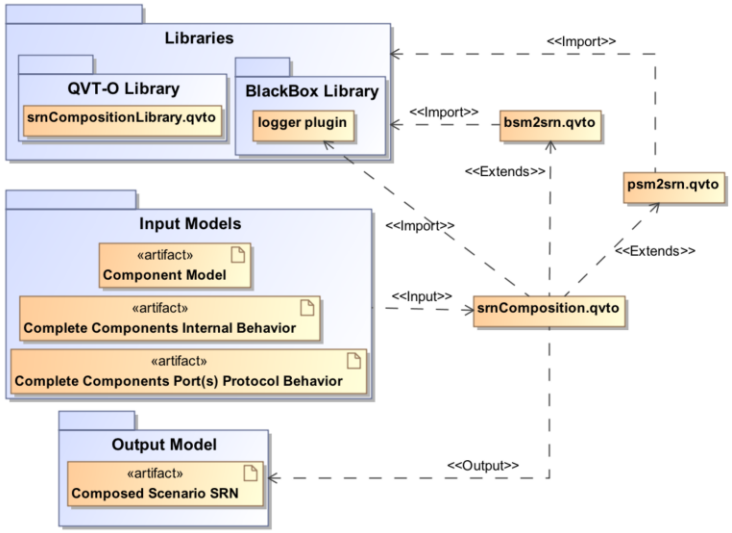


Figure 6.7: *sm2srn* Transformation Engines Architecture

The QVT-O *extends* mechanisms is employed for composed transformation. The input models are the critical scenario component model and all components' complete internal and ports behavior models. To transform and compose all components complete behavior (internal and ports), the user is asked to pass one UML file with nested submodels. As shown in Figure 6.8 each submodel holds one component internal behavior and another UML file for component ports protocol behaviors.

The *srnComposition* transformation is the main transformation class executed by the user. It performs several steps before calling other extended transformation classes as shown in the code in Figure 6.9 and the graphical representation in Figure 6.12. The first step of the *srnComposition* transformation is verifying the input models by calling *verifyInputModels()* helper. It checks whether the nested input models have the proper behavioral model, and it follows the CeBAM modeling guidelines in creating input models. This verification helper logs all violation of the CeBAM modeling guidelines to an external log file, using descriptive error message that help the user.

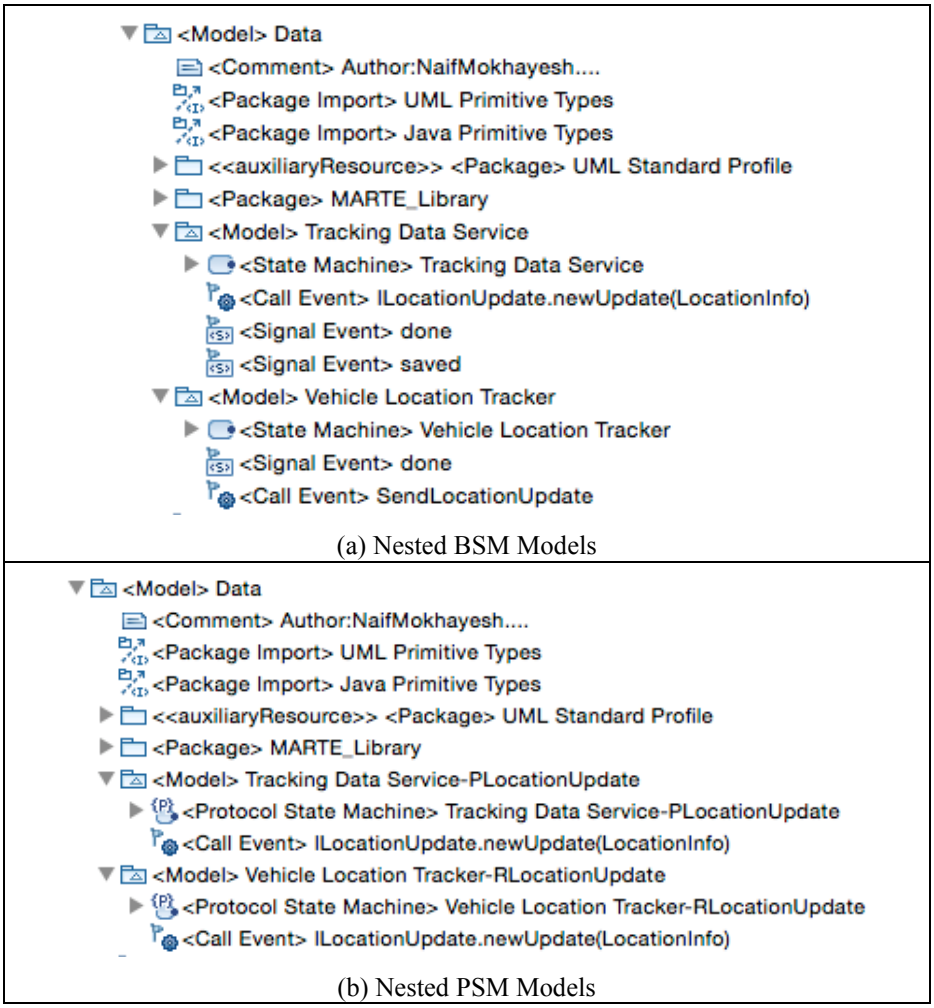


Figure 6.8: Nested Input Models of Components Behavior

```
transformation srnComposition(in inCOMPONENT : UML, in inBSM : UML,
                 in inPSM : UML, out outFinalScenarioSrnWithoutDeployment : SRN)
        extends transformation bsm2srn
        extends transformation psm2srn;

// counters and index for places and transitions id
property placeIndex : String;
property immediateTransitionIndex : String;
property timedTransitionIndex : String;
property placeToTransitionIarcIndex : String;
property transitionToPlaceOarcIndex : String;

property filePath : String =
"/Volumes/MyData/PhD_Thesis/Implementation/TransformationsLogs/";
property fileName : String = "srnComposition.txt";

main() {
        writeLog(filePath , fileName, "srnComposition started...\n" );
        // start counters for SRN model elements
        String.startStrCounter(placeIndex);
        String.startStrCounter(immediateTransitionIndex);
        String.startStrCounter(placeToTransitionIarcIndex);
        String.startStrCounter(transitionToPlaceOarcIndex);

        -- Step 0 : verify the input models
        if( verifyInputModels() ) then{
                --Step 1:get component BSMs in dictionary collection
                var bsmModelsDict : Dict(String,UML::Model);
                bsmModelsDict := inCOMPONENT.getAllComponentsBSMs();

                --Step 2:get component PSMs in dictionary collection
                var psmModelsDict : Dict(String,UML::Model);
                psmModelsDict := inCOMPONENT.getAllComponentsPSMs();

                -- Step 3 : Drive SRN for each BSM in step 1
                var derivedBsmSrnDict : Dict(String,SRN::StochasticRewardNet);
                inCOMPONENT.map bsm2Srn(bsmModelsDict , derivedBsmSrnDict);

                --Step 4:Drive SRN for each PSM in step 2
                var derivedPsmSrnDict : Dict(String,SRN::StochasticRewardNet);
                inCOMPONENT.map psm2Srn(psmModelsDict, derivedPsmSrnDict);

                --Step 5:Compose components internal derived SRN with its port(s)
                var completeNestedComponentsSrnDict :
                        Dict(String,SRN::StochasticRewardNet);

                --Step 5-1:build nested SRN models for each component
                completeNestedComponentsSrnDict :=
                        inCOMPONENT.buildNestedComponentSrns(derivedBsmSrnDict,
                                                             derivedPsmSrnDict);

                --Step 5-2:link (glue) each component internal SRN with its port
                inCOMPONENT.map
                   linkNestedComponentsSRNs(completeNestedComponentsSrnDict);

                --Step 6:link (glue) scenario SRN by adding connectors srnSubnets
                inCOMPONENT.map
                    linkFinalNestedScenarioSRNs(completeNestedComponentsSrnDict);
                writeLog(filePath , fileName, "srnComposition End..." );

        }else {
                writeLog(filePath,fileName, "verify all input models and retry");

                assert fatal(false);
        }endif;
}
```

Figure 6.9: Main Mapping Operation of *srnComposition* Transformation Engine

The next step is to build an intermediate dictionary collection to hold the components' complete internal behavior as well as the ports' complete protocol behavior. This step is performed by *getAllComponentsBSMs* and *getAllComponentssPSMs*. These two helpers traverse the input models and select each behavior model in nested submodel based on component's name and then copy it to the intermediate dictionary collection. Once this collection contains complete behavioral models (internal and ports), the *srnComposition* transformation calls *bsm2Srn* and *psm2Srn* mappings, according to the behavioral model type. These mappings delegate the request to the *bsm2srn* and *psm2srn* QVT-O classes, accordingly, for each behavioral model as shown in Figure 6.9 and Figure 6.12. The derived SRN subnets of complete component behavior (internal and port) will be stored in another dictionary collection to be used while composing derived SRN subnets, as explained in Section 6.4.4.

6.4.2 Deriving of Internal Component SRN Model

The *bsm2srn* transformation is the realization of all derivation rules defined in Chapter 5. Figure 6.10 shows a graphical representation of its design, as well as the first three levels of the hierarchical QVT-O mapping rules and helpers. This transformation accepts one behavioral state machine as input model and produces one SRN subnet. It imports *logger* black-box library to log the transformation progress and identified modeling errors in an external text file. In addition, it imports the *srnCompositionLibrary* a QVT-O library containing common queries and helpers. The *bsm2srn* transformation is generic and is called by *srnComposition* since it is part of the composed transformation. However, it can be used as a standalone transformation to derive a SRN model for any state machine conforming to the CeBAM approach.

The input model is the state machine of component's complete internal behavior, the result of the *CebamWeaver* transformation. First it is verified against CeBAM modeling guidelines defined in Chapter 4. This verification is performed by *VerifyStateMachineModel()* helper before deriving a SRN model to identify any modeling error. It traverses the model and identifies all modeling errors, logging them in the external text file with a descriptive error message to help the user locate the error and fix it. For example, it iterates over all vertexes of type state and checks if the entry or exit activity is local or calling other component's provided service. If it is not local, it is considered as modeling error. Another example is to check if any erroneous state has an activity or any erroneous transition has a trigger; both are considered as modeling errors. If any modeling error is detected, then the transformation is interrupted and the user is asked to verify the model and fix all the errors listed in the log file.

The transformation of the UML state machine to SRN model is performed in three steps. The first step is deriving SRN subnet for each state. In fact, we have many possible cases that must be handled by this transformation. For instance, a simple UML state with no activity, a simple UML state with *do* activity that calls remote service provided by other component, and simple UML state with *entry* and *exit* activities. For all possible cases defined in Chapter 5, there is a separate mapping rule called by *vertex2SrnSubnet* mapping that identifies the UML state case and calls the proper mapping rule, as shown in Figure 6.10. Interface places *P_send*, *P_send_Propagate* and *P_request* will be created as it is required in each UML simple state mapping rule. It is then linked (glued) to the proper component port SRN transition *t_incoming* and *t_outgoing* during the composition of the internal behaviour SRN subnet with its ports subnets. During this composition, the

compatibility verification is performed by the *srnComposition* transformation, as explained in Section 6.4.4.

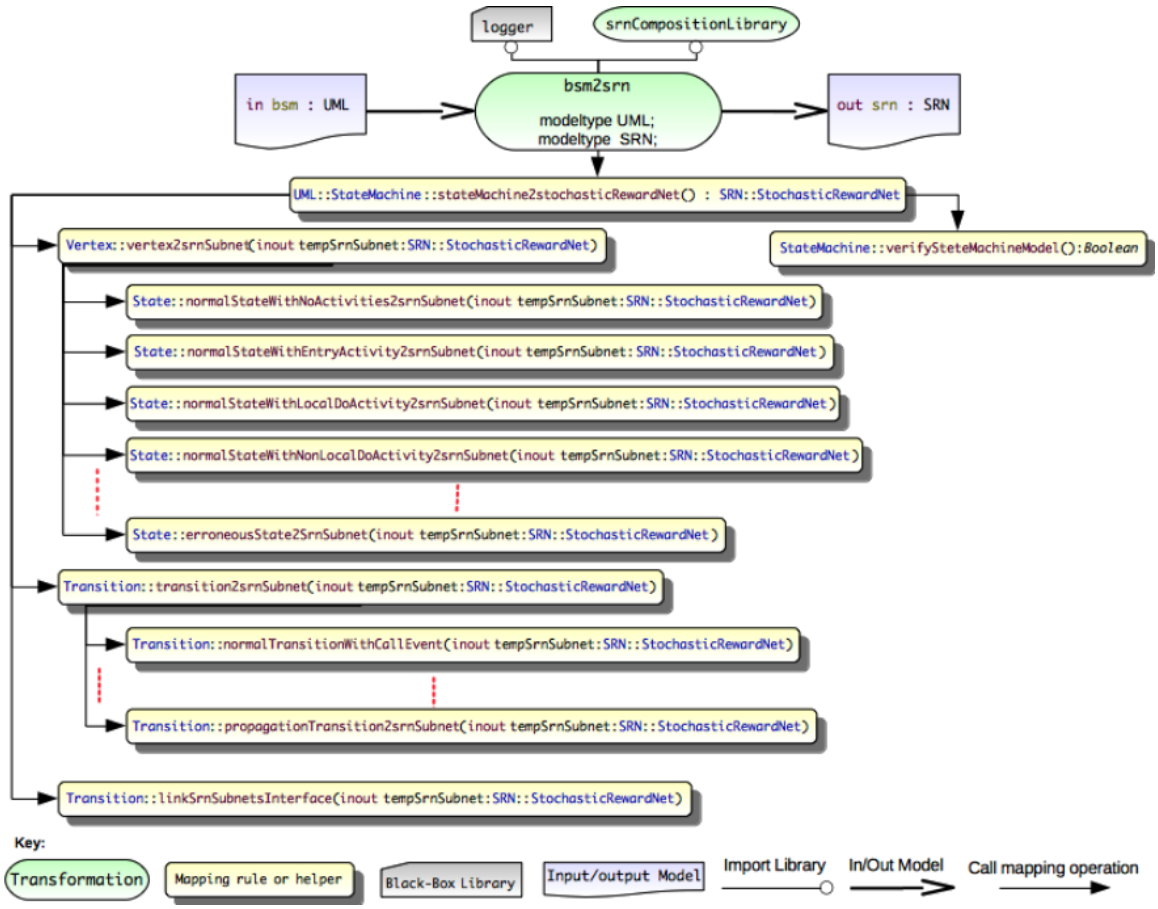


Figure 6.10: Graphical Representation of *bsm2srn* Transformation

The next step is to map each state machine transition to a SRN subnet model. There are different types of UML transitions, such as normal transitions with trigger event of type *CallEvent*, normal transitions with trigger event of type *SingleEvent*, propagation transitions, or fault activation transition. All these cases are identified by *transition2srnSubnet* mapping rule, and call corresponding mapping operations, as shown in Figure 6.10. In this step, we link the derived SRN subnet of a UML transition with the SRN subnet of its target UML state. This gluing will add a new arc from the derived SRN

interface transition to the *P_init_<stateName>* place of the target state. Due to limited space, we only list a small sample of these mapping rules (see Figure 6.10). The last step is to link the derived SRN subnet of UML simple states with the derived SRN subnets of its outgoing UML transitions, as illustrated in Chapter 5. This task is performed by *linkSrnSubnetInterface*. This mapping will iterate over all derived SRN subnets and link interface places and SRN transitions by adding new arcs.

6.4.3 Deriving Component Port SRN Model

The *psm2srn* transformation is the QVT-O implementation of the transformation rules defined in Chapter 5. In Figure 6.11 we present a simplified graphical representation of the first three levels of the hierarchical transformation rules. It imports *logger* black-box library and the QVT-O library *srnCompositionLibrary*. This transformation is designed to be generic and can be used as a standalone transformation to derive the SRN model for any port protocol state machine that conforms to the CeBAM approach. However, we use this transformation as a part of a composed transformation. The *srnComposition* transformation calls the mapping rule *protocolStateMachine2StochasticRewardNet* to derive a SRN subnet model for the port protocol state machine.

Similar to *bsm2srn* transformation, the first step is to verify the conformance of the input model to the CeBAM modeling guidelines. The transformation will be aborted if any modeling error is found. We customized the code to list all the modeling errors and log them in the external log file, to be fixed by the user.

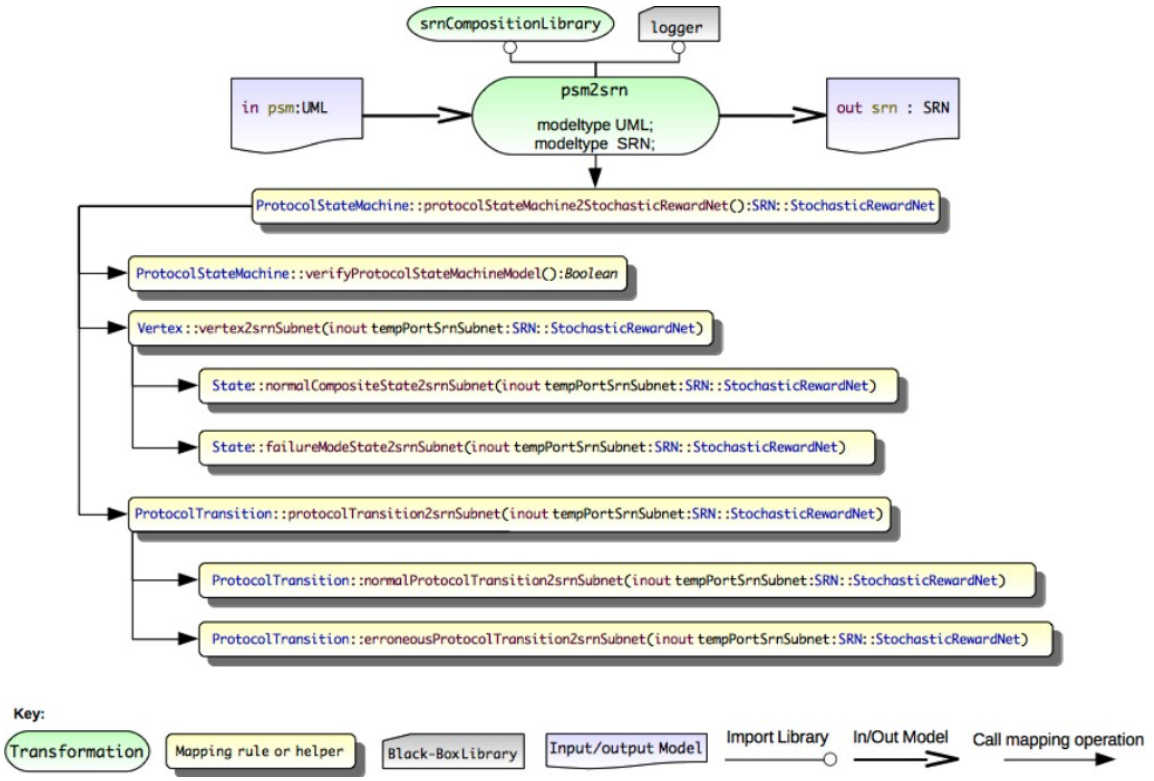


Figure 6.11: Graphical Representation of *psm2srn* Transformation

The next step is to derive the SRN subnet of each state, either composite normal state or failure mode state. The last step is to derive the SRN subnet of each protocol transition and link it accordingly with the derived SRN subnets of protocol states. The interface places *P_mailbox* and *P_mailBoxPropagation* will be created for each incoming protocol transition either normal or erroneous. Also *P_sendRequest* and *P_sendPropagation* places will be created for each normal or erroneous protocol transitions stereotyped as outgoing protocol transition. These places will be used to compose the provided and required services by *srnComposition* transformation. Moreover, they will be used to check the component compatibility as explained in the next section.

6.4.4 Conformance and Compatibility Verification for Composed Derived SRNs

The final output model of *srnComposition* transformation is a single SRN model that represents a critical scenario behavior as shown in Figure 6.12. Therefore, the derived SRN models from internal behavior and port protocol behavior must be under one root *StochasticRewardNet* model elements. The *buildNestedComponentSrns* helper will iterate over the generated SRN subnets models stored in the dictionary collections and build one SRN model with nested SRN subnets models for each component. A sample of the final output model of this transformation is shown in Figure 6.13, which shows the final SRN model for the VTS case study.

Conformance verification is performed at the time of gluing SRN subnets of each component internal behavior with its port(s) SRN(s) subnets as explained in Chapter 5. The *linkNestedComponentSRNs* mapping glues the component internal behavior with its ports by adding new arcs. For an incoming request, it adds an arc from *t_incoming* port transition to *P_request* place in the internal SRN model. Furthermore, for an outgoing request, it adds arcs from *P_send* places in the internal SRN to *t_outgoing* port transitions. The code of this mapping is customized to identify all compatibility issues in the scenario and save it in external text log file. This helps the user to fix all modeling errors at once and re-execute the transformation.

Compatibility verification is the next task performed by *srnComposition*. This verification will check the compatibility between provided and required services and also the failure propagation between dependent components, as illustrated in Chapter 5. The *LinkFinalNestedScenario* helper performs this task. It starts by creating a new nested SRN model that holds all *T_send* and *T_lost* timed transitions of connectors SRN subnet.

According to the provided service name, it will link *P_mailbox* places with matched *P_sendRequest* places by adding *T_send* and *T_lost* timed transitions and required arcs. If no matching is found, it will be considered a compatibility error. This process continues until all other components' ports are connected, regardless of the identified compatibility issues. In fact, this allows for logging all compatibility issues in the external log file as errors, to be fixed by the user at once, before re-executing the transformation.

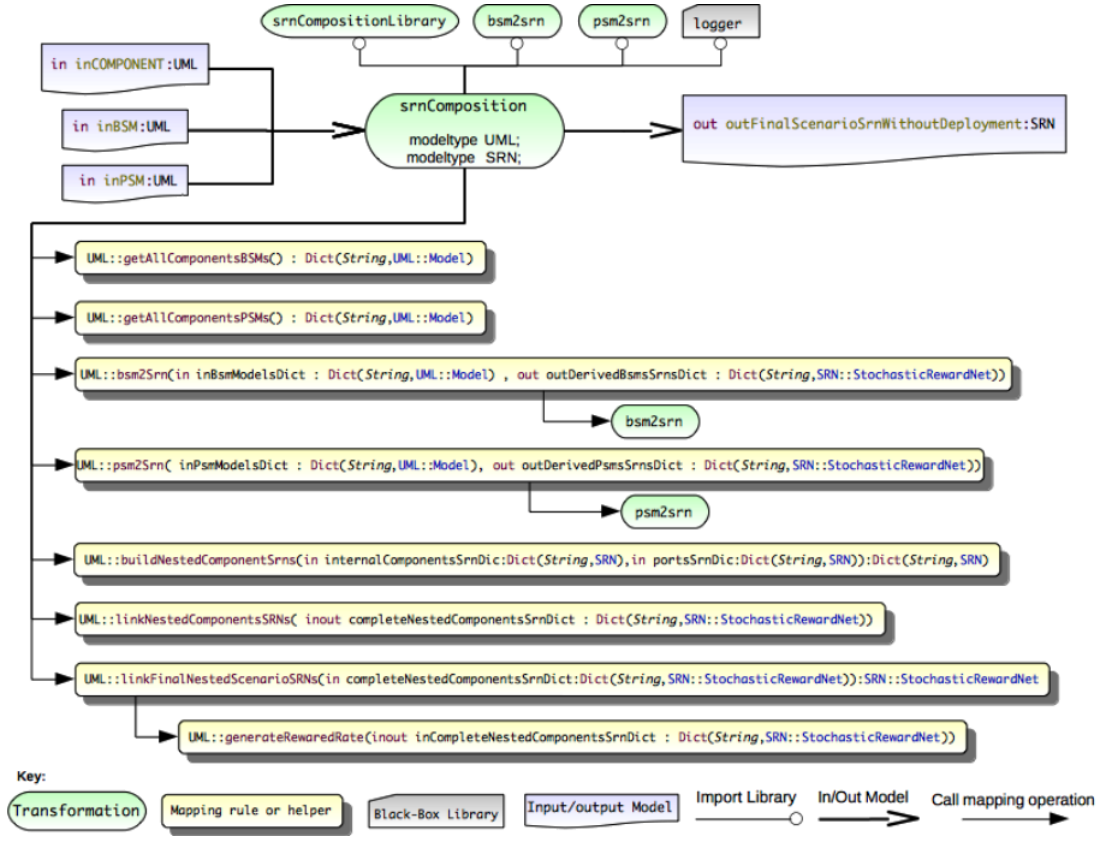


Figure 6.12: Graphical Representation of *srnComposition* Transformation

The last task performed by *srnComposition* transformation engine is generating reward rate expressions. Here we are interested in SRN reward rate that will be used to compute the unreliability of the critical scenario. It is a conditional expression that checks the number of tokens in the SRN places that represent failure modes of the critical

scenario. The SRN *mark()* operation will be called to return the number of tokens in the selected places. According to our transformation rules and naming conventions presented in Chapter 5, we can easily identify the failure mode places by selecting any place whose name starts with *"P_failureMode"*. For example, in Figure 6.13 the generated unreliability reward rate of VTS case study has two failure modes places that are identified according to the naming convention. For each of these failure modes, we call the *mark()* operation to consider the current marking for the unreliability computation if any of these places has one or more tokens. This SRN expression will be transformed to a reward rate function by the *srn2cspl* transformation engine as explained in next section. In Chapter 8 we present a complete example of using reward rates for computing critical scenario unreliability.

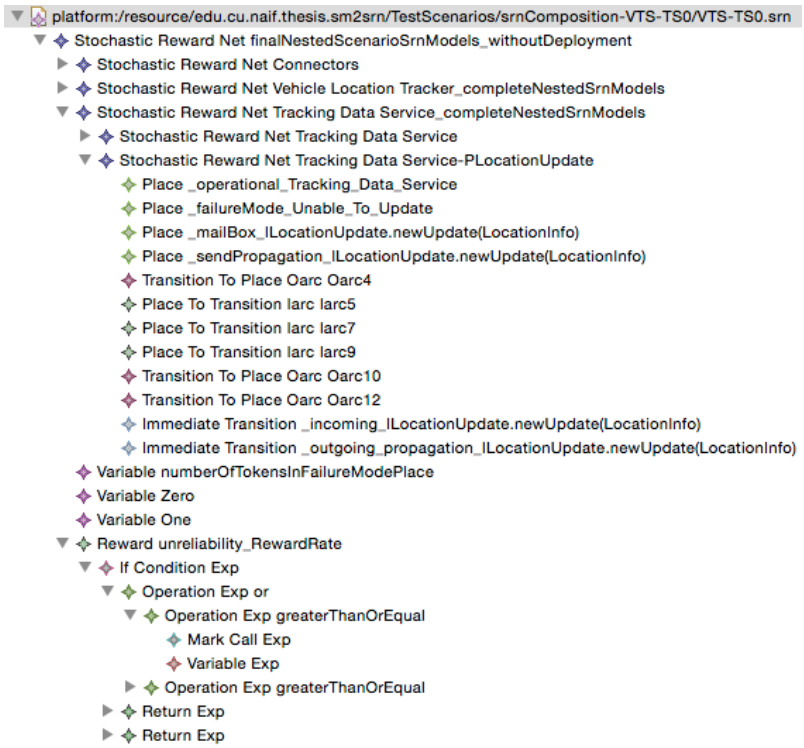


Figure 6.13: Derived SRN Scenario Model of VTS Case Study

6.5 Phase Three: Transformation SRN to CSPL

The derived SRN model from the *sm2srn* transformation represents the SRN subnet structure and marking dependent expressions such as reward, guard, firing rate, and multiplicity. The metamodel of SRN (see Appendix A) has no information about the solution type or any parameter settings required by SPNP solver package. However, to solve the derived SRN model, we have to generate a CSPL code that has all the required parameter settings along with SRN subnet structure. One possible solution is to extend the SRN metamodel to capture all the required information in order to be able to generate the CSPL code. This solution will increase the complexity of both the SRN metamodel and the model transformations. Another solution is to define an intermediate metamodel that captures the CSPL construct and semantics, and to develop a new transformation engine for generating a CSPL model from the derived SRN model. This solution is preferred, since it reduces the complexity by separation of concerns. Also, it makes the transformation chain more modular and easier to maintain, since each transformation engine has one specific job.

CSPL is the input language of the SPNP solver. It describes the structure of SRN model and all other analysis parameter configuration such as specifying solution approach, simulation or analytical. It has six mandatory functions: *options()*, *net()*, *assert()*, *ac_init()*, *ac_reach()*, and *ac_final()*. Each of these functions has specific statements and functions. For instance, the *net()* functions call other functions such as *place()*, *iarc()*, *oarc()* and *imm()*, to create the SRN subnet structure. Another example is that *ac_final()* function must call *solve()* function that specifies the solution type of the model either steady state analysis, transient analysis, or simulation. According to [20]

that has full details about the CSPL language, we developed the CSPL metamodel that captures all CSPL language constructs and semantics, as shown in Appendix A.

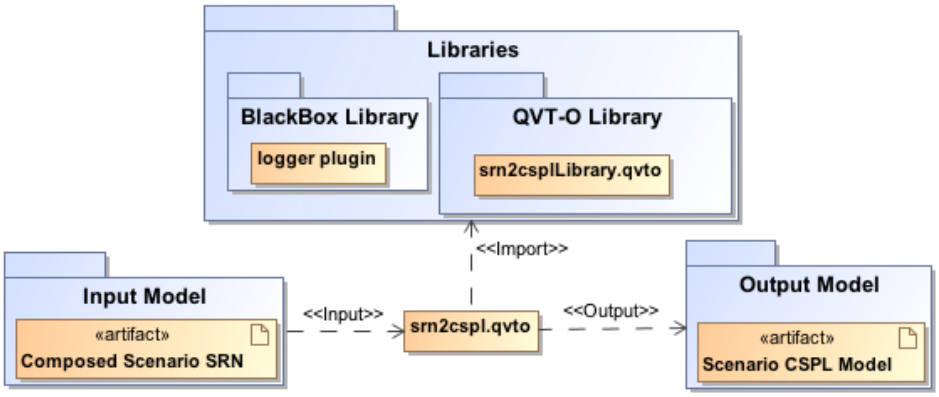


Figure 6.14: *srn2cspl* Transformation Engine Architecture

The *srn2cspl* is an intermediate transformation designed to reduce the complexity in generating CSPL code from the derived SRN model. The architecture of *srn2cspl* transformation is shown in Figure 6.14, and the graphical representation is given in Figure 6.15. This transformation has three main tasks. First, it transforms the SRN model structure to a set of element creation subroutines, called in turn by the *net()* function. It is a one-to-one mapping between each model element (i.e.; place, transition, arc) and the corresponding subroutine that creates it. This task is performed by *createNetFunction* mapping rule. Second, it transforms the marking dependent expressions to CSPL marking dependent functions. In SRN metamodel, we have four marking dependent types: reward, guard, multiplicity, and firing rate. Each of these marking dependents has *srnExpresion* that will be transformed into a CSPL statement. We built four mapping rules based on marking dependent expression types: *reward2markingDependentFunction()*, *firingRate2markingDependentFunction()*, *guard2markingDependentFunction()*, and *multiplicity2markingDependentFunction()*.

The last task of this transformation is to create other mandatory functions. For instance, the *ac_final()* function, which indicates the solution type and calls the reward function to compute the desired output measures. Another example is the *options()* function, which has all SPNP configuration settings. In this mapping, we created all possible options and set their values to NULL, in order to be modified by the user through a properties file, as explained in the next section.

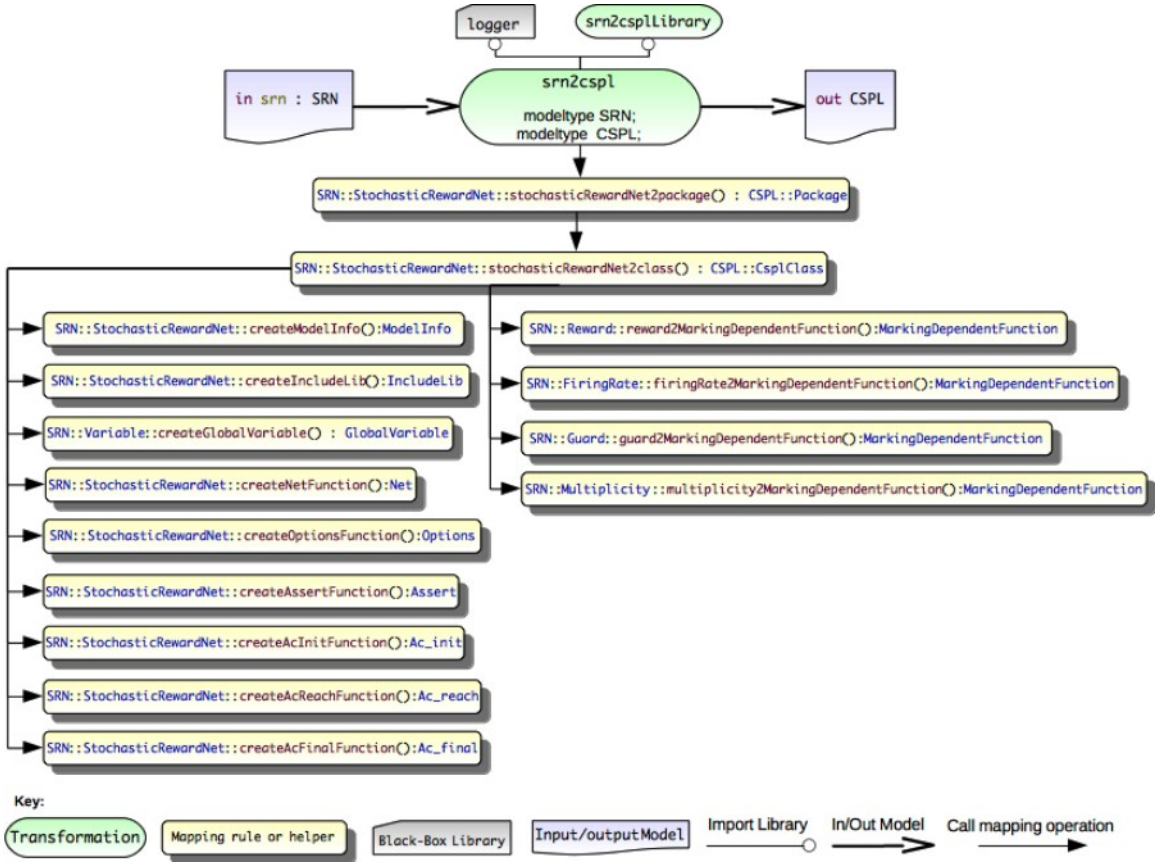


Figure 6.15: Graphical Representation of *srn2cspl* Transformation

6.6 Phase Four: Generating CSPL code

The derived CSPL model from the *srn2cspl* transformation has all the information required to generate CSPL code. This intermediate transformation reduces the gap between the derived SRN model and the actual CSPL code. As a result, the *cspl2text* is

straightforward and the generic model-to-text transformation is designed to transform the CSPL model into CSPL code that can be executed using the SPNP solver.

The *cspl2text* is implemented using Eclipse Acceleo, which is the implementation of model-to-text standard proposed by OMG [130]. Figure 6.16 describes the architecture of this transformation. It takes a CSPL model and it produces a CSPL class file, which is a C-based class file. The class *mainModule.mtl* is the entry class that iterates over *CsplClass* collection property of root package in the input CSPL model to generate a text file for each class by calling *classCsplFile.mtl*.

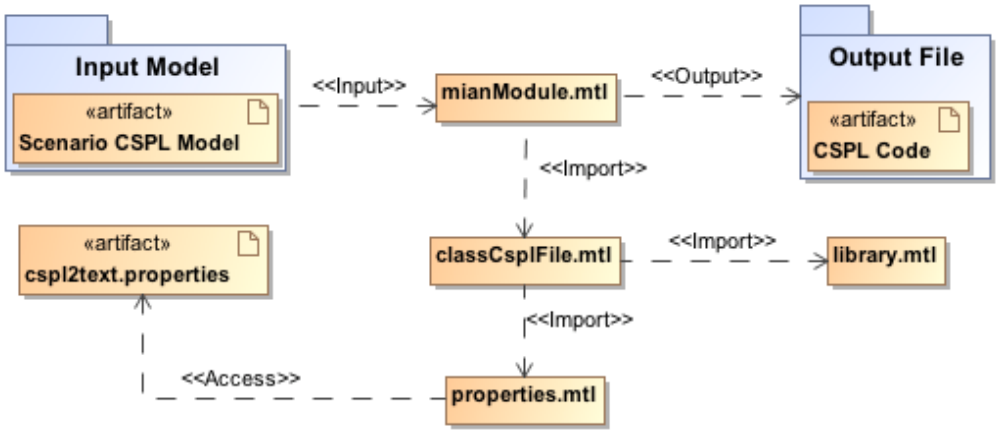


Figure 6.16: *cspl2text* Transformation Architecture

The *classCsplFile.mtl* has all templates to generate a CSPL code file. For example, *genCsplFunctions()* template generates the text for each CSPL mandatory functions such as *net()*, *options()*, and *ac_final()*. In addition, it imports two Acceleo files: *library.mtl* and *properties.mtl*. The first one has common templates such as a a template that generates copyright text and a template that gets the actual folder path to store the generated text file. The second one is created to access *cspl2text.properties* file and read the value of any selected SPNP parameter option.

The properties file is created to preserve and list all possible options of the SPNP solver package defined in [20]. It has a description for each SPNP option parameter as shown in Figure 6.17. Initially, all option values are defined to NULL. The user will use this text file to set only the value of the required options to be included in the *option()* function of the final CSPL code. In fact, the idea of having all SPNP options in an external file that is accessible by the user makes the template code generic and allows the user to modify and select options without changing any transformation templates.

cspl2text.properties

```
# this file has all CSPL options
#
#################################################################################

#################### Generic options ####################
# the path of the CSPL source folder in the generated project
generatedCodeFolder = CsplCode

#################### Available options for intermediate files ####################


# IOP_PR_RSET : this options specify whether the reachability set and graph should be printed for tangible
# Values: VAL_YES, VAL_NO, VAL_TAN
# Default value: VAL_NO
IOP_PR_RSET = VAL_YES

# IOP_PR_RGRAPH : this options specify whether the reachability set and graph should be printed
# Values: VAL_YES, VAL_NO
# Default value: VAL_NO
IOP_PR_RGRAPH =  VAL_NO

# IOP_PR_MARK_ORDER : This option specifies the order in which the markings are printed
# Values: VAL_CANONIC , VAL_LEXICAL , VAL_MATRIX
# Default value:VAL_CANONIC
IOP_PR_MARK_ORDER = NULL

# IOP_PR_FULL_MARK : This option specifies whether the markings are printed in long format,
# Values: VAL_YES, VAL_NO
# Default value: VAL_NO
IOP_PR_FULL_MARK  = NULL
```

Figure 6.17: Snapshot of *cspl2text.properties* File

Chapter 7: Verification and Validation of Model Transformation Chain

In this chapter, we present the validation and verification of the model transformation chain that was first introduced in Chapter 6. The validation of the transformation chain has been carried out by means of unit testing as well as integration testing of the composed transformation engines. We developed a set of test scenarios for each transformation class based on different case studies developed during the work on this thesis. Each transformation engine was tested separately for models that were created according to the CeBAM approach. We also worked to validate the behavior of the transformation engines in identifying modeling errors of the input models and how they are logged in the external log file.

In Section 7.1 we briefly summarize the recent model transformation testing techniques, since this is a new research area. Section 7.2 presents our unit testing approach and lists test scenarios for each unit test. Validating transformation engines behavior for input models that were not conforming to CeBAM user guidelines is presented in Section 7.3. Finally, we apply the transformation chain to the thesis running case study (VTS), and we show all the models generated by each transformation engine.

7.1 Model Transformation Testing Approaches

Model transformation verification and validation techniques are a new and emerging research area. The recent survey in [136] lists and compares some model-based testing approaches. The approach presented in [137] is among the approaches we surveyed. It uses the MDA approach to automate the generation of test cases from platform-independent models. The model transformation techniques were utilized to generate test cases by transforming the input sequence diagram to a general unit test case

model. The next step is to transform this unit test into a platform-specific test case, such as JUnit, using model-to-text techniques.

A recent model transformation testing technique called Model trANsformation Testing (MANTra) was presented in [138]. The idea is to define a methodology to test the QVT-O transformation code using only QVT-O transformation itself. In other words, test cases can be built directly within the QVT-O language and can verify the results from the same development environment without any other tool. The authors defined a set of assertion constructs similar to the JUnit assert statement that was invoked from the QVT-O test case code. In a test case written in QVT-O code, the model transformation class under test was imported. In fact, the QVT-O built-in reuse feature is useful, because it allows accessing the transformation class to be tested. The next step is to create an input model to be used as input data for the test case. The actual transformation result will be compared with the expected value defined in the *assertEquals()* function.

We were not able to utilize this approach, which uses the QVT-O object creation mechanism to create the input model for test cases. In our case, the input models are created according to the CeBAM approach using a UML editor and then exported to the Eclipse modeling environment. Moreover, in our testing approach we need to validate the models generated by each transformation class, instead of validating each mapping rule separately. As mentioned earlier, the testing scenarios developed in the thesis were based on case studies.

7.2 Validating the Transformation Engines

The transformation chain described in Chapter 6 consists of four cascaded transformation engines. These engines were developed using Eclipse modeling tools. We

used Eclipse QVT-O implementation for model-to-model transformation and *Acceleo* for model-to-text transformation. Moreover, we used Ecore tools 2.0 to develop SRN and CSPL metamodels [135] (see Appendix A). The first model transformation engine is *CebamWeaver* that works on UML behavioral and protocol state machines. We selected *MagicDraw* professional edition to model component behaviors [131]. We prepared the modeling workspace by installing MARTE profile and building other required profiles, such as DAM and the CeBAM profiles set. This workspace is used by a software modeler to construct normal components behavior, as well as ports behavior, while the dependability expert builds erroneous behavior according to the CeBAM approach guidelines explained in Chapter 4. These models will be exported from MagicDraw in the format of an Eclipse UML 2.x XMI file and imported to the Eclipse QVT-O workspace to start the transformation chain that will derive the SRN model and then generate CSPL code.

To validate the proposed transformation chain, we have developed a set of test scenarios for each transformation engine. We design test scenarios to ensure that the transformation process has been tested from end to end. Each test scenario consists of a set of test cases designed to validate mapping rule behaviors. These test scenarios use different cases studies that were developed during the work on this thesis. Some test scenarios are performed to test the whole transformation chain while others are designed to test a specific transformation class and its mapping behavior. For instance, test scenarios that validate the derivation of an SRN model will be executed first on *sm2srn* transformation engine and then the output model will be used as input for *srn2cspl* and *cspl2text* transformation engines. Another example of test cases that are designed to test a

specific transformation class and its mapping rules behavior is the transformation of models that have modeling errors and do not follow CeBAM modeling guidelines. The next subsections will explain the scope and goal of each test scenario and present samples of output results.

7.2.1 Transformation Unit Testing

Our objective is to test every model transformation engine and get high validation coverage of mapping rules. We divided the proposed transformation chain into three test units based on the target models as follows:

1. CeBAM approach implementation.
2. SRN model derivation.
3. CSPL code generation.

The first test unit was dedicated to testing *CebamWeaver* transformation engine. As explained in Chapter 6, it is implementing the CeBAM approach that will apply the refactor aspects and erroneous aspect models on internal component behavior. It is also responsible for weaving erroneous aspect models on protocol behavior of component ports. The test scenarios were designed to validate how aspects are applied to the base model, including the stereotypes. The second unit test focuses on validating the *sm2srn* transformation engine, which implements the SRN mapping rules explained in Chapter 5. It uses the complete component behavior derived by the *CebamWeaver* transformation engine along with the UML component model to compose an SRN model of a critical scenario under analysis. In this unit test, we designed test cases to test all the transformation rules, as well as the SRN subnets composition.

As explained in Chapter 6, each transformation engine has a set of transformation classes that work together using the transformation composition feature of QVT-O. However, each of these transformation classes is designed such that it can be used standalone by passing it the required input models. Therefore, some test scenarios in the first and second unit test are designed to validate specific transformation classes and others are designed to test the composed transformation engine. For example, the user can directly invoke and execute *BsmErroneousComposition* transformation class to just apply the erroneous aspect to the internal component behavior. On the other hand, the user can also call *CebamWeaver* as a composed transformation engine to apply the erroneous aspect model and multiple refactor aspects models in one execution.

The *CebamWeaver* and *sm2srn* transformation engines have a verification mechanism that helps in detecting the modeling errors of the input models. Each input model is analyzed to check if it conforms to CeBAM modeling guidelines or if there are some violations. This feature was also considered in designing test scenarios. For instance, a test scenario checks how *BsmRefactorComposition* transformation class handles an input aspect model that is not designed according to CeBAM guidelines. Other test cases are designed to validate how the *sm2srn* transformation detects the component conformance and compatibility modeling errors, as well as the correctness of a component's internal and ports behaviors.

The last test unit combines two transformation engines: *srn2cspl* and *cspl2text*. The goal of these two transformation engines is to generate the CSPL code from the derived SRN model. The input models of these two transformation engines are assumed to be correct and free of modeling errors, such as component conformance and compatibility

issues. Test cases validate the transformation of marking dependent expression into CSPL functions such as reward rate and transition firing rates. Others validate the transformation of SRN net creation commands, as well as other mandatory CSPL methods. The next sections present the test scenarios for each unit, and apply the transformation chain to the running case study VTS.

7.2.2 Testing the CeBAM Implementation

The implementation of the CeBAM approach is a composite transformation. The user invokes the *CebamWeaver.qvt* transformation that takes as input the internal component behavior, one or more refactor aspects, and one erroneous aspect. This transformation will first apply the refactor aspects to the internal behavior and then the erroneous aspect to the refactored model. For component ports, the user will call the *PsmErroneousComposition* class to apply the erroneous aspect to the component protocol behavior. Six test cases are designed to cover different possible situations and to test all transformation classes in the composite transformation. Table 7.1 lists the test scenarios and the included test cases for each transformation class. The following is a summary description of the test scenarios:

- TS#1: This test scenario is designed to validate some mapping rules of the *CebamWeaver* transformation engine, as shown in Table 7.1. It has a set of test cases for each transformation class to validate the weaving of a single refactor aspect model and of the erroneous aspect model with the internal component behavior. Applying these aspects models also involves applying stereotypes to the new woven model elements. Similarly, for component ports this test scenario has a set of test cases to validate the weaving of a single erroneous aspect model to component ports protocol

behavior. We used input models from the VTS case study, which do conform to the CeBAM modeling guidelines. The expected result is a complete component internal and port behavior, including normal and erroneous behavior with all required stereotypes.

- TS#2: This test scenario has set of test cases that validate the behavior of *CebamWeaver* transformation engine for input models that do not conform to the CeBAM approach. The refactor aspect and erroneous aspect models have some modeling errors, such as not applying the *AspectBSM* profile or not using a submachine state in the erroneous aspect. In this test scenario, we use a modified version of the VTS case study with some modeling errors. The expected output of this test scenario is a set of error messages that guide the user to fix the modeling errors. In fact, these error messages are part of the assert statements that were used in verifying the aspect models before applying it to the base model as explained in Chapter 6. Table 7.1 shows the list of test cases applied to each transformation class.
- TS#3: This test scenario validates how the *CebamWeaver* transformation engine weaves multiple refactor aspects at different join points in the base model. A user in this test case will pass only one UML file with a set of refactor models. Before applying the refactor aspect, the *BsmRefactorComposition.qvto* must parse these models and process the refactor aspects one after the other. Moreover, the erroneous aspect has multiple erroneous paths, and some of them have multiple erroneous states. This aspect must be weaved at different join points in the refactored model. Therefore, *BsmErroneousComposition.qvto* must traverse each erroneous path and apply it with applied stereotypes to the refactored model according to the *pointCut*

query. For the protocol behavior of the component port, the erroneous aspect has multiple failure modes that should be woven to the <<*Operational*>> composite state. The expected output is the correct refactored component behavior (internal and ports) with all the erroneous paths. The Emergency Monitoring System (EMS) case study was used in this test scenario. Table 7.1 shows the list of test cases.

- TS#4: This test scenario validates the behavior of *CebamWeaver* transformation engine when a wrong *pointCut* query was used in the aspect models. We used a modified version of EMS case study with *pointCut* query that does not return value. Table 7.1 shows the test scenario and its test cases included. As explained in Chapter 6, we use Eclipse OCL implementation as a standalone black-box module to be called from the QVT-O class to parse and execute the *pointCut* query. As part of CeBAM user guidelines (Chapter 4), we provide the user with template and sample queries that can be used to specify the join point in the base model. However, the user is not limited to these templates, and they can also use customized queries. The expected output of this test scenario is an error message that guides the user to fix the OCL query if the syntax is wrong or no result is returned.
- TS#5: This test scenario is dedicated to validating the *BsmRefactorComposition* class for applying one refactor aspect model on a multiple join point. As explained in Chapter 4, the refactoring aspect will add a new intermediate state and move the transition activity to be a *do* activity in the new intermediate state. The user has the option to design one refactor aspect to be then applied to multiple transitions. In this case, the *pointCut* query specifies the function used as an action for multiple transitions in the base model. The expected output is a correctly refactored model.

Table 7.1: Test Scenarios for Validating *CebamWeaver* Transformation Engine

Test Case	Transformation Unit	Test Scenario					
		TS1	TS2	TS3	TS4	TS5	TS6
Weave single refactor aspect models to single joint point	BsmRefactorComposition	✓					✓
Weave multiple refract aspects models to different join points				✓			
Weave one refactor model to multiple join points						✓	
Apply stereotypes on the newly added elements		✓		✓		✓	✓
Weave refactor aspect model with wrong *PointCut* query					✓		
Weave refactor aspect model that does not conform to CeBAM approach guidelines			✓				
Weave single erroneous aspect model to single join point	BsmErroneousComposition	✓					✓
Weave single erroneous aspect model to multiple join points							✓
Weave erroneous aspect model that has multiple erroneous paths to different join points				✓			
Weave erroneous aspect model that has multiple erroneous states				✓			
Apply stereotypes on the new woven elements		✓					✓
Weave erroneous aspect model that does not conform to CeBAM approach guidelines			✓				
Weave erroneous aspect model that has the wrong *PointCut* query					✓		
Weave an erroneous aspect model that does conform to CeBAM approach guidelines	PsmErroneousComposition		✓				
Weave an erroneous aspect model with wrong *PointCut* query					✓		
Weave a single erroneous aspect model that has only a single failure mode state		✓					✓
Weave an erroneous aspect model that has multiple failure mode states				✓			✓
Apply stereotypes on the new woven elements		✓		✓			✓

- TS#6: This test scenario is designed to validate mapping rules of the *CebamWeaver* transformation engine, as shown in Table 7.1. Input models are from the Field

Monitoring System (FMS) case study, and do conform to the CeBAM guidelines. This test scenario is similar to TS#1, except for one erroneous aspect model with one erroneous path that must be woven at multiple join points without replicating the erroneous state and failure mode. In other words, the *BsmErroneousComposition* transformation weaves the erroneous behavior at the first join point, and then for the other join points, just adds fault activation transitions to the already woven erroneous path. This is performed by checking the existence of the error and failure mode states before weaving the erroneous path. The expected output model is a complete component behavior (port and internal behavior). Table 7.1 shows the list of test cases applied to each transformation class.

7.2.3 Testing the SRN Model Derivation

The *sm2srn* consists of composite transformations that transform the complete behavior (internal and ports) of the components involved in the critical scenario to a single SRN subnet. The *sm2srn* transformation engine is designed to take three input models: the critical scenario component model, all component internal behaviors, and all component ports protocol behaviors. As explained in Chapter 6, the user executes the *srnComposition.qvto* class that requires three input UML models. The first model shows the software architecture (i.e., the components involved in the critical scenario) and the deployment nodes, annotated with DAM and MARTE stereotypes (as presented in Chapter 5). The second and third input models are all components complete internal behavior and all ports complete protocol behaviors. These models are the results of the *CebamWeaver* transformation engine. The user must organize these input UML models so that each internal component behavior and port protocol behavior form a submodel.

The *sm2srn* transformation engine will parse input models and its submodels to derive a single SRN model for the critical scenario. A component UML model is used by the transformation engine to identify each internal component behavior and its ports protocol behavior. It is also used to guide transformation during SRN subnets composition.

The derivation of the SRN model is performed in three sequential steps. First, the main transformation class (*srnComposition.qvto)* calls the *bsm2srn.qvto* transformation class to derive the SRN subnet for the internal behavior of each component. The next step is to call *psm2srn.qvto* transformation class to derive the SRN subnet for all components ports protocol behavior. The last step performed by the *srnComposition.qvto* transformation class is to compose all derived SRN subnets. It starts by composing the derived SRN subnet for each component internal behavior with the derived SRN subnets of each owned port. The conformance checking process is executed to identify any incoming or outgoing requests (normal service or failure propagation) not handled by the internal behavior. After the errors are fixed by the user, the SRN subnets of the components are composed according to provided and required services and the compatibility between components is verified.

All of these transformation classes contain the actual implementations of the transformation rules presented in Chapter 5. Therefore, testing scenarios are designed to cover all transformation rules. Moreover, this implementation includes a verification mechanism that checks whether the input models respect the CeBAM modeling guidelines. If there is any violation of the modeling guidelines, the transformation will stop and the transformation engine provides the user with an error message that will help in fixing the error. A similar approach was considered for designing test scenarios. We

designed five test scenarios to cover all the mapping rules and the possible modeling errors. All the test scenarios are based on case studies developed during the work on this thesis. Each scenario has a set of test cases targeting specific mapping behaviors: some test specific transformation classes, others test the composed transformation engine. Table 7.2 shows the list of all test scenarios and their test cases. The following is a summarized description for each test scenario.

- TS#1: This test scenario is designed to validate mapping rules of the *bsm2srn.qvto* transformation class. We use modified versions of EMS case study that has some normal states with all activities (*entry*, *do,* and *exit*) executed locally and contains an erroneous path. The model is annotated with DAM and MARTE profiles. The expected result is an SRN subnet derived according to the transformation rules in Chapter 5. It starts by deriving the SRN subnet for all states (normal and erroneous) and then composes it with the derived SRN subnet of behavioral transitions (normal and erroneous). Table 7.2 shows the list of test cases included.
- TS#2: This test scenario is dedicated to validating mapping rules of the *bsm2srn.qvto* transformation class. It has a set of test cases that cover some transformation rules such as a normal state with just *entry* activity, a normal state with nonlocal do activity, and another normal state with local do and exit activities. Table 7.2 shows the list of test cases included. We use the Automatic Collision Notification (ACN) case study in this test scenario. The expected result is the correct SRN subnet according to the transformation rules defined in Chapter 5.

Table 7.2: Test scenarios for validating *sm2srn* Transformation Engine

Test Case	Transformation Unit	Test Scenario				
		TS1	TS2	TS3	TS4	TS5
Initial *Pseudostate* and final state	bsm2srn	✓	✓	✓	✓	✓
State with no activity				✓		✓
State with just *entry* activity			✓			
State with local *doActivity*				✓		✓
State with nonlocal *doActivity*			✓			
State with *exit* activity			✓			
State with *entry* and local *do* activities				✓		
State with *entry* and nonlocal *do* activities				✓		✓
State with all local activities		✓				
State with *entry*, *exit* and nonlocal *do* activities					✓	
State with *entry* and *exit* activities					✓	
State with local *do* and *exit* activities			✓			
State with nonlocal *do* and *exit* activities					✓	
Erroneous state		✓	✓	✓	✓	✓
Failure mode state		✓		✓	✓	✓
Transition from initial state		✓	✓	✓	✓	✓
Transition with *CallEvent*		✓	✓	✓	✓	✓
Transition with *SignalEvent*		✓	✓	✓	✓	✓
Transition stereotyped with *<<GaWorkLoadEvent>>*			✓	✓		✓
Fault activation transition		✓	✓	✓	✓	✓
Propagation transition		✓	✓	✓	✓	✓
External Failure Propagation Transition		✓	✓	✓		
Composing derived SRN subnets of states and transitions		✓	✓	✓	✓	✓
Getting attributes values of applied profiles		✓	✓	✓	✓	✓
BSM does not conform to CeBAM guidelines						✓
Operational composite state	psm2srn			✓		✓
Failure model state				✓		✓
Incoming request protocol transition				✓		✓
Outgoing request protocol transition				✓		✓
Incoming failure propagation transition				✓		✓
Local failure propagation protocol transition				✓		✓
PSM does not conform to CeBAM guidelines						✓
Composing incoming request	srnComposition (Conformance)			✓		✓
Composing outgoing request				✓		✓
Composing incoming failure propagation				✓		✓
Composing outgoing failure propagation				✓		✓
Identifying conformance issue				✓		✓
Build nested SRN model	srnComposition (Compatibility)			✓		✓
Composing provided service				✓		✓
Composing required service				✓		✓
Composing failure propagation				✓		✓
Identifying compatibility issue				✓		✓

- TS#3: This test scenario validates the *sm2srn* transformation engine. We use an FMS complete case study to test the whole composed transformation behavior. It includes three components, some with more than one component port. The expected output is a complete composed SRN subnet for the critical scenario. The resulting model is a nested SRN subnet, with an SRN subnet for each component. These subnets are linked to each other using connectors SRN subnets as explained in Chapter 5 and 6. Moreover, it has a reward rate expression and firing rate expressions for timed transitions. The input models do conform to the CeBAM modeling approach, and do not have conformance or compatibility modeling errors, therefore, no errors messages are expected. All test cases included are shown in Table 7.2.
- TS#4: This test scenario is applied to the bsm2srn.qvto transformation class. It covers other transformation rules, such as a normal state with entry and exit activities and another state with local entry and exit activities and nonlocal do activity. Table 7.2 has the list of the test cases included. The expected output is a correct SRN subnet derived according to the transformation rules that were presented in Chapter 5. In this test scenario, we use one component behavioral model from the Automatic Guided Vehicle (AGV) case study.
- TS#5: This test scenario is customized to validate the built-in input model verification mechanism. As explained in Chapter 6, the *sm2srn* transformation engine will check the input model to identify all CeBAM user guideline violations, which are considered as modeling errors. It also identifies the conformance and compatibility modeling errors. These errors will be logged into the external log file to help the user fix the errors. In this test scenario, we use a modified version of the VTS case study.

Table 7.2 shows all test cases included. The expected output is a list of all modeling errors logged into a log file with no SRN model generated. In Section 7.3 we show a sample of these modeling errors and output log files.

7.2.4 Testing CSPL Code Generation

Generating CSPL code for the derived SRN model is performed in two transformation phases. As explained in Chapter 6, the *srn2cspl* transformation engine is an intermediate transformation intended to reduce the gap between the derived SRN model and the actual CSPL code, and thus minimize the complexity of generating CSPL code. The *cspl2text* transformation engine is a generic CSPL code generator, which generates the complete CSPL code from the *srn2cspl* output, with all the required options for setting solver parameters.

Table 7.3: Test scenarios for validating *srn2cspl* and *cspl2text* transformation engines

Test Case	Transformation Unit	Test Scenario	
		TS1	TS2
Generating net subroutine	srn2cspl.qvto & mainModule.mtl & classCsplFile.mtl	✓	✓
Generating mandatory CSPL subroutines		✓	✓
Generating marking dependent functions		✓	✓
Generating global variables		✓	✓
Generating option parameters		✓	✓

In our validation approach, these two transformations are considered as one test unit. We developed two comprehensive test scenarios based on VTS and FMS case studies. These scenarios cover almost all the mapping rules used to generate complete and correct CSPL code. For example, we verified how the reward rate expression was transformed to the CSPL function and used in CSPL *ac_final()* subroutine. Another example is testing the generation of the option parameters that were defined by the user,

as well as global variables. Table 7.3 shows the list of test cases included in both test scenarios.

7.3 Validating the Detection of Modeling Errors

Deriving a correct SRN model from the UML state machine is challenging. For example, the modeler can build behavioral models that do not conform to the CeBAM approach, which will be transformed into incorrect/incomplete SRN models. The user guidelines introduced in Chapter 4 were defined to help users build correct component behavioral models and aspect models, which can be transformed into a correct SRN model. To ensure that correct SRN models are derived, we included verification processes in *CebamWeaver* and *sm2srn* transformation engines to check the input models before starting the transformation process. In the *CebamWeaver* transformation engine, we used the QVT-O built-in assert mechanism to verify the aspect models, as explained in Chapter 6. If any modeling error is detected, then the transformation is terminated and a descriptive error message is displayed to the user. In the *sm2srn* transformation engine we built two customized helpers named *verifyProtocolStateMacheineModel()* and *verifyStateMachineModel()* that check the component complete internal behavior and component port protocol behavior to detect any modeling error that violates the CeBAM modeling guidelines. Component conformance and compatibility errors are also detected during the SRN subnets composition. The implementation is customized to detect and list all errors in all the behavioral models of all the components in the critical scenario, rather than stop at the first error. This will help the user to follow the external log file and fix all modeling errors at once.

As part of validating the transformation engines, we built a set of test scenarios to test the implemented verification process by passing models that do not conform to the CeBAM guidelines. These test scenarios validate the model transformation engine capability to detect modeling errors. The next subsection will show samples of test scenarios executions and the output log files. We use here the VTS thesis running case study introduced in Chapter 4. We intentionally created some behavioral modeling errors as well as component conformance and compatibility errors to test how the implemented model verification mechanism detects them.

7.3.1 Detection of Behavioral Modeling Errors

In the *CebamWeaver* transformation engine, the aspect models are checked using the QVT-O assert mechanism before starting the In-Place transformation that weaves the context-specific aspect models with the base model. The transformation is terminated when it identifies any modeling error. The descriptive error message will help the user to fix the identified modeling errors and then re-execute the transformation. To validate the modeling error identification, we modified the behavior of the refactor aspect and erroneous aspect models of *Vehicle Location Tracker* component by introducing some modeling errors. For example, in TS#2 we do not specify the refactor transition in the refactor aspect model, as the outgoing transition from the source state is not stereotyped with <<*Refactor*>> as shown in Figure 7.1(a). This transition should have a *pointCut* attribute indicating the OCLQuery for finding the joint point. In Figure 7.1(b) we introduced a modeling error in the erroneous aspect model: missing stereotype for an outgoing transition. According to the CeBAM guidelines, this transition is the beginning

of the erroneous path and must be stereotyped either by <<*LocalFaultActivation*>> or <<*ExternalFailurePropagation*>>.

The transformation engine takes these two input aspect models along with the component original behavior, and starts by checking the correctness of the two aspect models. As expected the *CebamWeaver* transformation engine was able to detect both behavioral modeling errors and terminate the transformation with a descriptive error message, as shown in Figure 7.2. All other possible modeling errors are treated in a similar way.

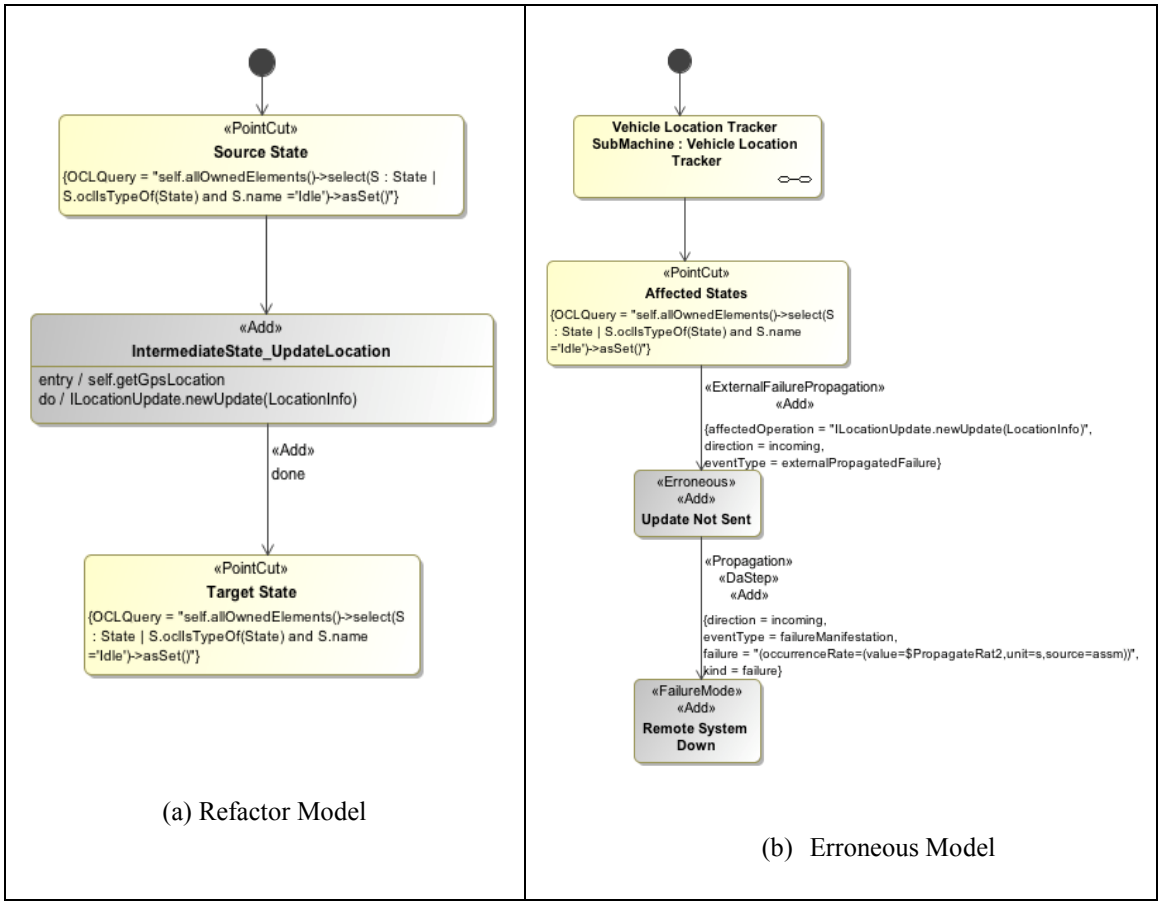


Figure 7.1: Modified Aspect Models of VTS Case Study

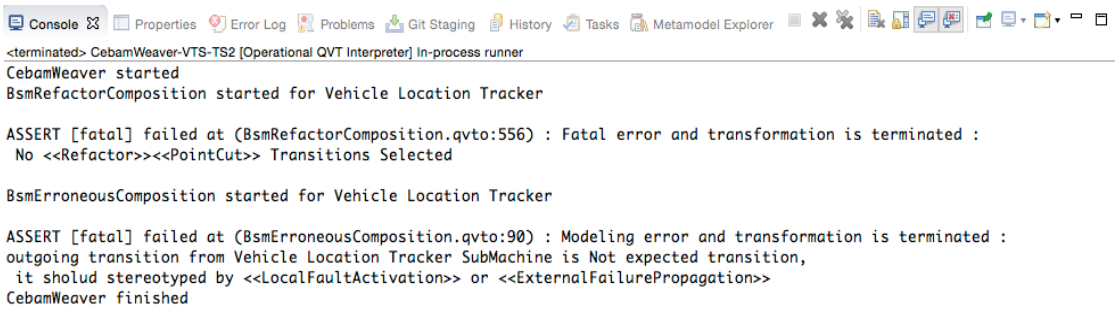


Figure 7.2: *CebamWeaver* Transformation Engine Error Messages

The input models of the *sm2srn* transformation engine include the critical scenario component model as well as complete internal behaviors and ports protocol behaviors derived by the *CebamWeaver*. As presented in Chapter 6, we use the component model to compose the derived SRN subnets of the components according to the critical scenario architecture. Components behaviors are checked to identify any violation of the CeBAM user guidelines. The *bsm2srn.qvto* transformation class is called to derive the SRN subnet for each component. It starts by calling the *verifyStateMachineModel()* helper, which parses the complete component internal behavior to detect any behavioral modeling error.

In TS#5 for the *sm2srn* transformation engine we introduced some modeling errors in *Tracking Data Service* internal behavior, such as adding a UML state with two outgoing transitions triggered by an event of type *Signal* and a *do* activity that is not of type *FunctionBehavior* or *OpaqueBehavior*. However, the component port is correct, and *Vehicle Location Tracker* component also conforms to CeBAM modeling guidelines. After executing the test scenario, the result was as expected: all the modeling errors were detected and included in the descriptive error message and no SRN model was derived. Figure 7.3 and Figure 7.4 show a snapshot of the generated external log files.

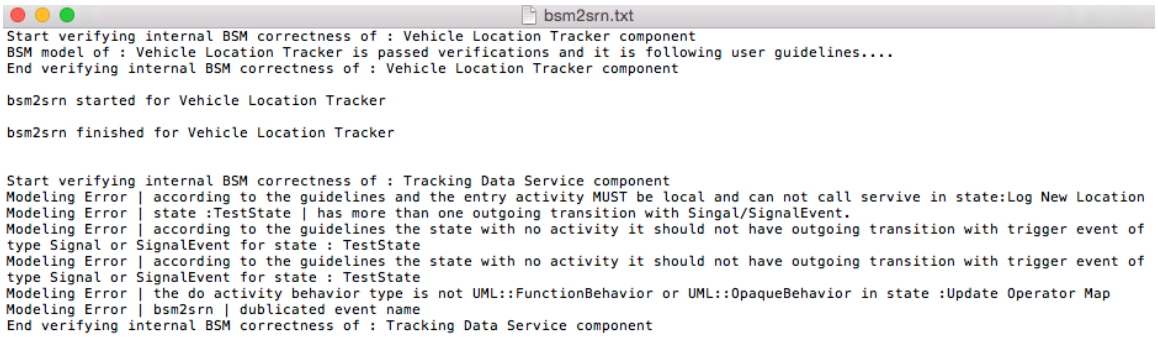


Figure 7.3: Log File Generated from *bsm2srn.qvto* with Modeling Errors

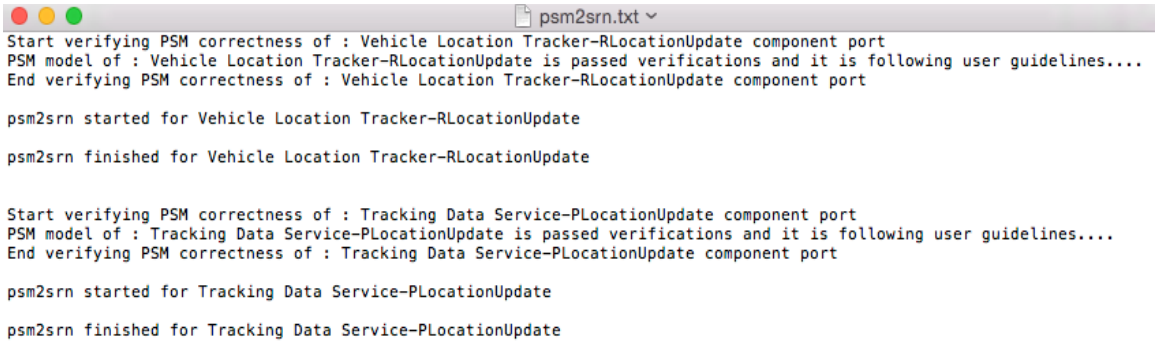


Figure 7.4: Log File Generated from *psm2srn.qvto* with No Modeling Errors

7.3.2 Detection of Conformance and Compatibility Modeling Errors

According to the transformation algorithm described in Chapter 6, the component conformance is checked after deriving SRN subnets of internal behavior and their protocol behavior. During the composition process, the transformation will detect any port incoming message that is not handled by the internal behavior and consider it as a conformance modeling error. For example, incoming failure propagation or service request is modeled in the component port, but it is not handled by internal component behavior. In fact, this will lead to deadlock or absorbing marking in the derived SRN, as explained in Chapter 5. Similarly, any outgoing message (normal or failure propagation) from the internal behavior not matching with the outgoing transition is also detected as a conformance modeling error.

Component compatibility checking during component composition (as explained in Chapter 5), is performed as the last step in *srnComposition* transformation engine. It finds the matching *P_mailbox* places for each required service or outgoing failure propagation. If no match is found, then it is considered as a compatibility error. Such an error leads to an absorbing state in the derived SRN model. In TS#5, we introduce some conformance and compatibility errors, as presented in Chapter 5. For instance, in the port of *Tracking Data Service*, the *logVehicleLocation* outgoing request is not modeled in the port protocol behavior; therefore, a dangling send request SRN place is detected and reported as a conformance modeling.. Similarly, the *reportIssue* provided service is modeled in the component port but is not handled by the internal component behavior.

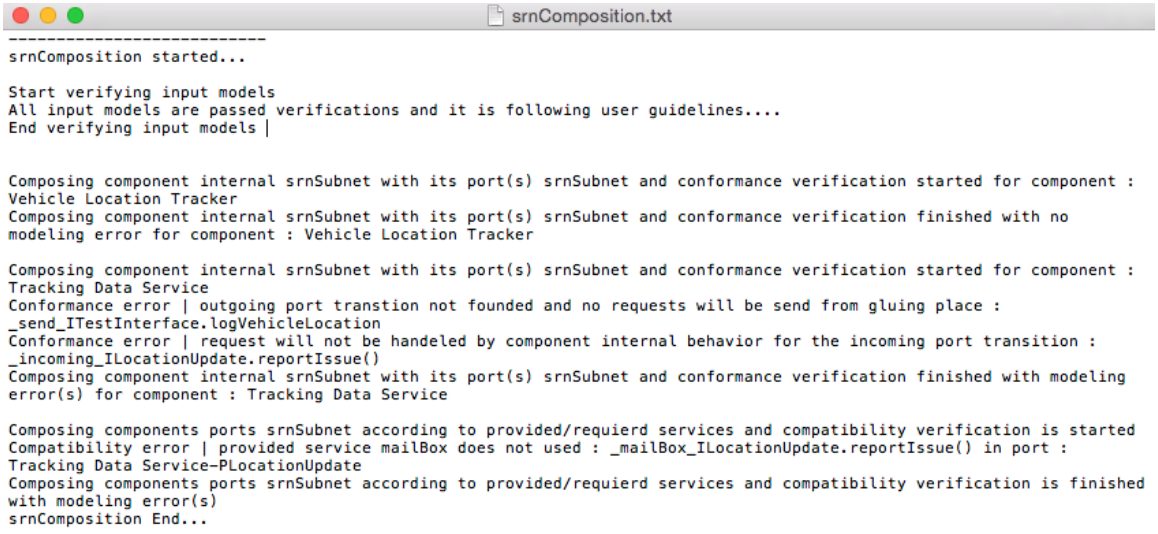


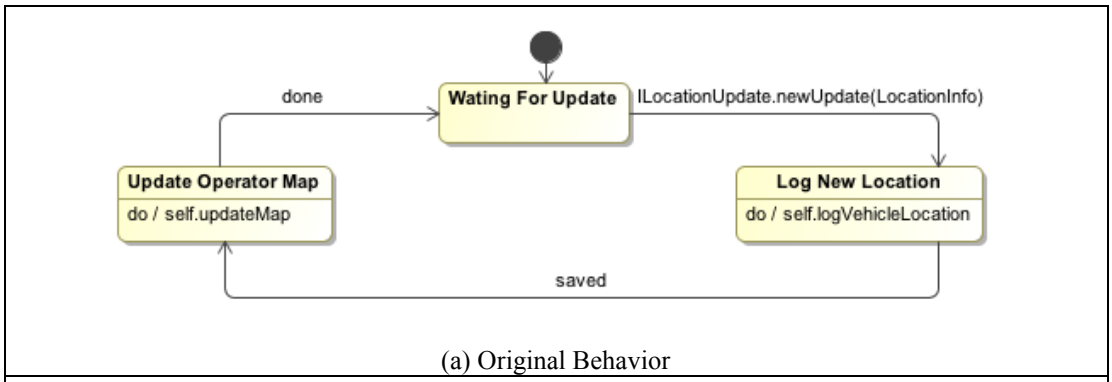
Figure 7.5: Detected Conformance and Compatibility Errors of VTS Case Study

If a provided service is not used by other components in the critical scenario, it is considered as a compatibility modeling error. For example, the *Tracking Data Service* component provides *reportIssue*, but this is not used by any dependent component since there is no UML outgoing transition from any component port that sends requests to this

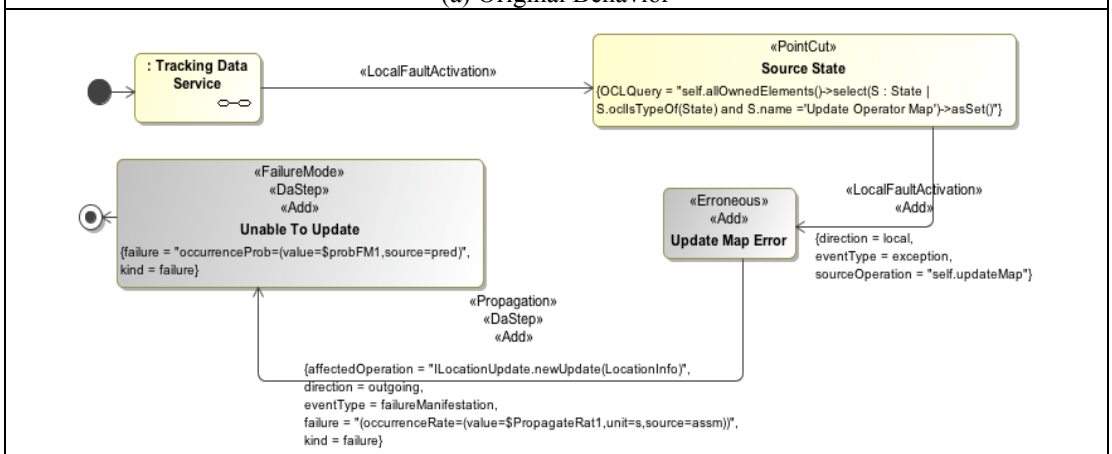
provided service. Therefore, it is considered as a compatibility modeling error. Figure 7.5 shows the list of detected conformance and compatibility modeling errors.

7.4 Applying Transformation Chain to Vehicle Tracking System

In this section, the transformation chain is applied to the VTS running case study after correcting the modeling errors introduced in the previous section. During the thesis work we developed different case studies and applied the transformation chain to all of them. However, only the VTS case study is presented here for space reasons.



(a) Original Behavior



(b) Erroneous Aspect

Figure 7.6: Normal and Erroneous behavior of *Tracking Data Service* Component

7.4.1 Applying CeBAM

The *Tracking Data Service* component provides *newUpdate* service as part of the implementation of the *ILocationUpdate* interface. It receives requests through its port

PLocationUpdate. Figure 7.6 shows the shows the original internal behavior of the component and the erroneous aspect model. The internal behavior has one erroneous aspect that captures the fault activation from the state *Update Operator Map*. Note that the erroneous aspect has only one erroneous path compared to the model presented in Chapter 4.

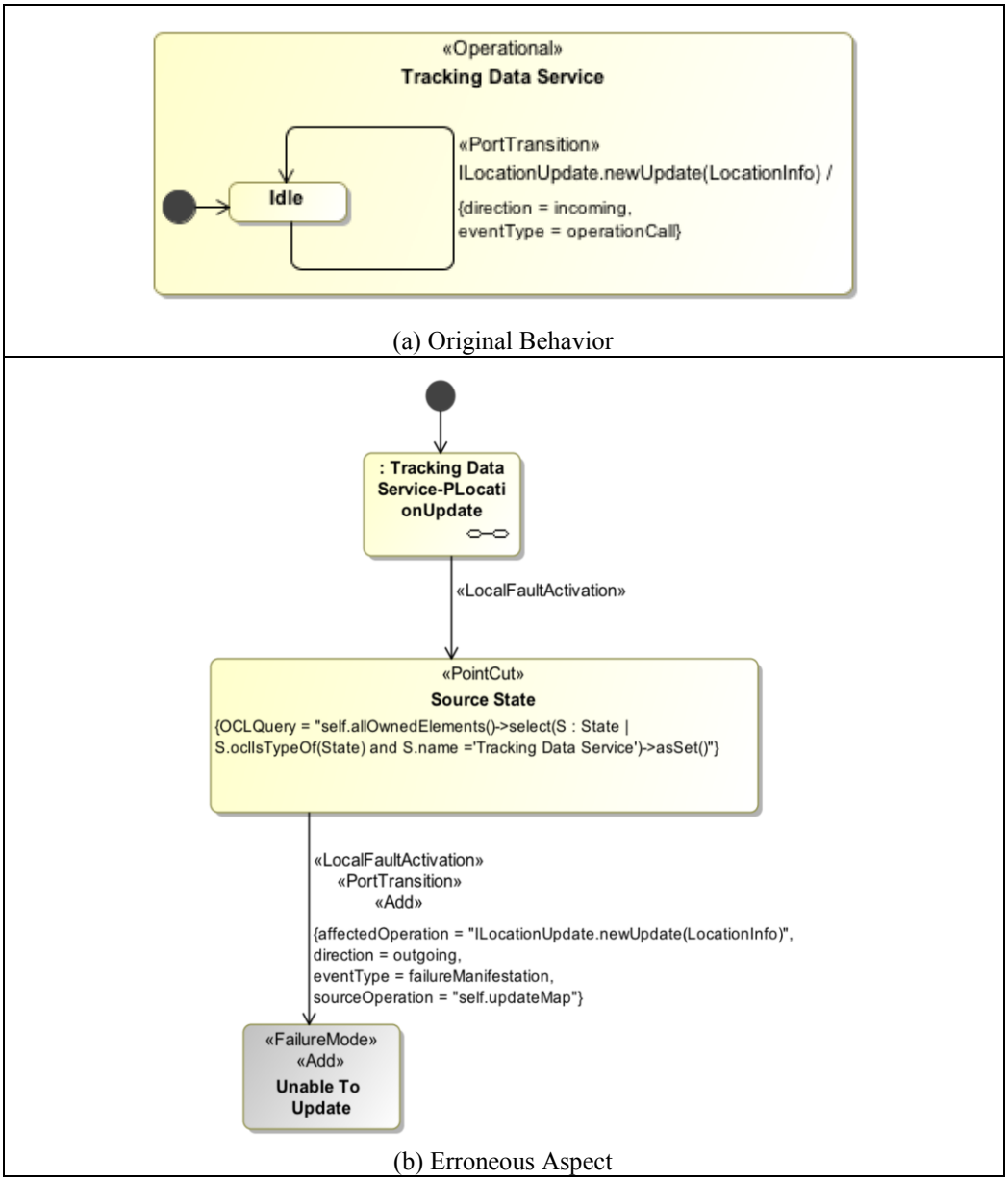
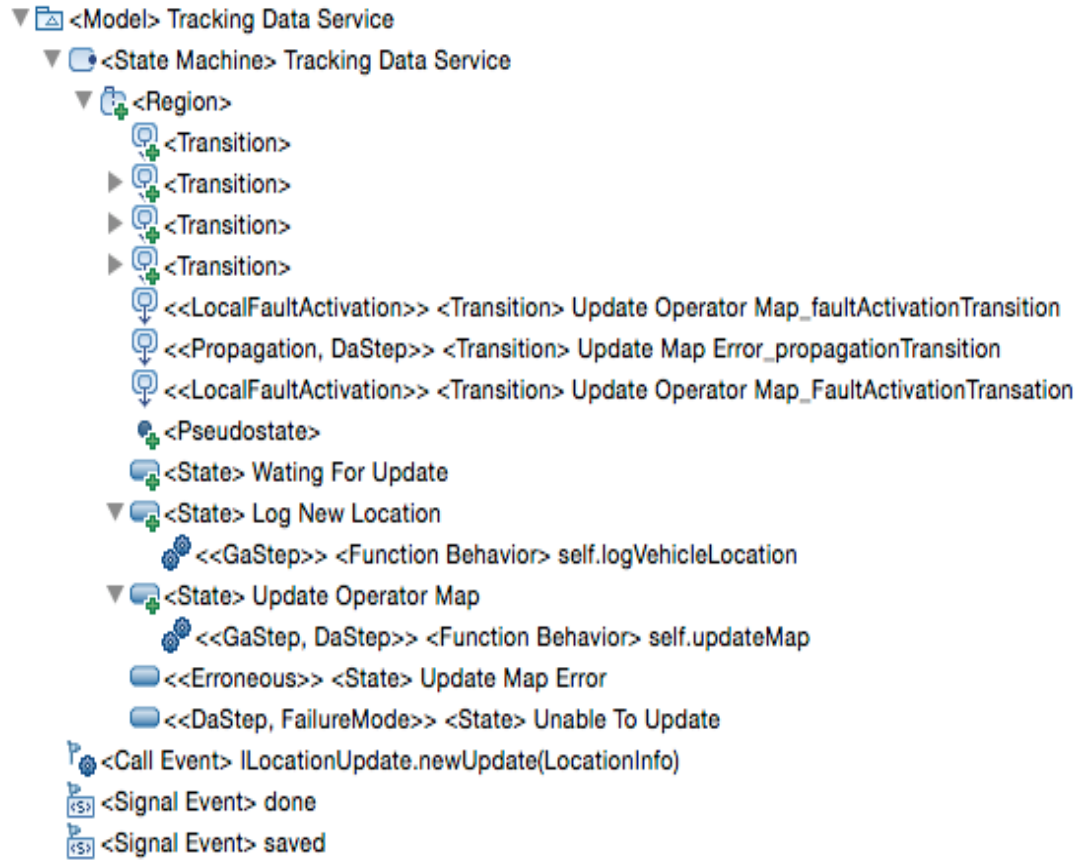


Figure 7.7: *PLocationUpdate* Port Protocol Behavior



(a) Internal Behavior after Applying Erroneous Aspect

(b) Port Protocol Behavior after Applying Erroneous Aspect

Figure 7.8: Complate Behavior of *Tracking Data Service* Component (internal and port)

An internal manifested failure mode is propagated to ports and then to the other dependent components, as explained in Chapter 4. In some cases, multiple internal failure modes are manifested only as one failure mode in the component port. For example, multiple propagation transitions that have the same value for the *affectedOperation* attribute are linked to one port failure mode. The notion of failure mode encapsulation was explained in Chapters 4 and 5, and it is implemented in the *sm2srn* transformation engine. Figure 7.7 shows the original component port behavior and the erroneous aspect model. The second component is *Vehicle Location Tracker,* and all its original behaviors and aspects have already been presented in Chapter 4. We use the *CebamWeaver* transformation engine to apply both the refactor and erroneous aspects to all components, and examples of the result models are shown in Figure 7.8.

7.4.2 Deriving the SRN model

The *sm2srn* transformation engine is designed to take three input models: the critical scenario component model, all component internal behaviors, and all component ports protocol behaviors. The component model is used as transformation and composition guidance since it has architectural information about the critical scenario that helps to compose the final SRN subnet. The user is asked to organize these input models in a specific way to be parsed by the transformation engine. For example, the internal behavior of all components is passed in one UML file, with a submodel that holds complete component internal behavior. In a similar way, each protocol behavior of each component port is placed in one submodel under one UML file.

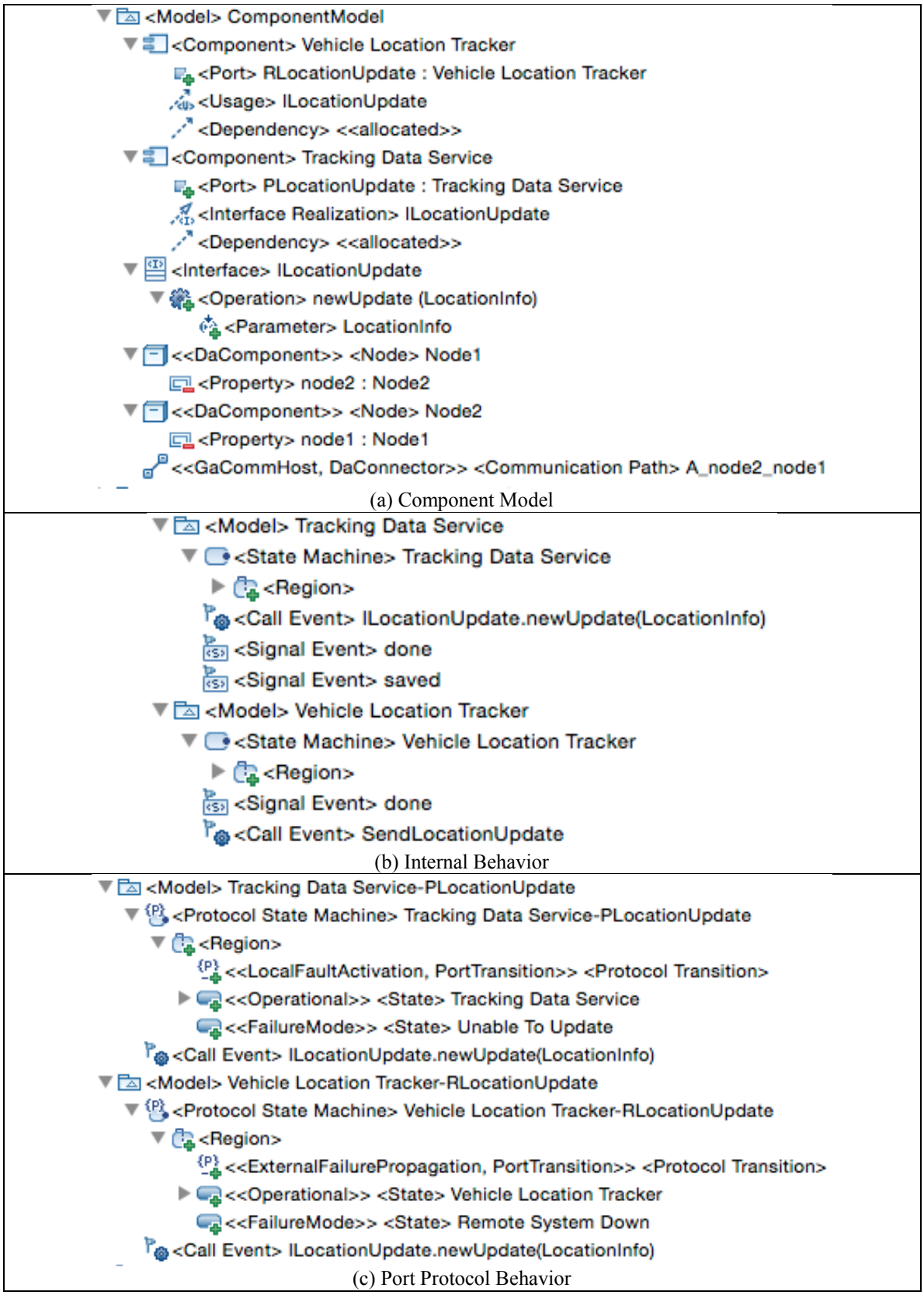


Figure 7.9: Input Models for *sm2srn* Transformation Engine

Figure 7.9 shows the input model of the VTS case study and how it is organized. The transformation engine queries these input models and generates an SRN subnet for each component internal behavior and ports protocol behavior. Each derived subnet is in an srnSubnet nested from the main SRN subnet, as explained in Chapter 6. These srnSubnets are connected and linked to each other and checked for conformance and compatibility, as explained in the previous section. Figure 7.10 shows the derived SRN subnet for the VTS case study and all marking dependent expressions such as reward rate.

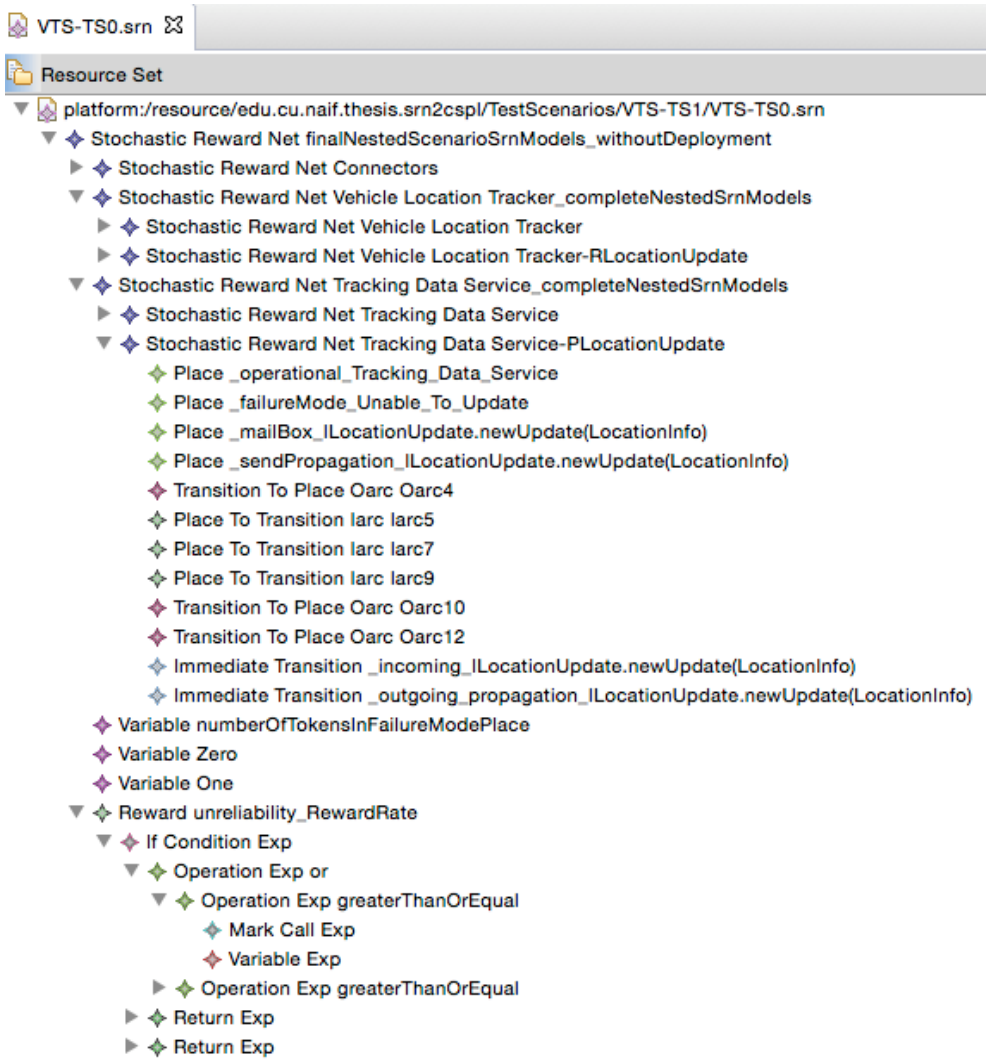


Figure 7.10: Derived SRN Model of VTS Case Study

7.4.3 Generating CSPL code

As already explained in the previous sections, generating the CSPL code is performed by the *srn2cspl* and *cspl2text* transformation engines. The derived SRN model is passed to the intermediate *srn2cspl* transformation. It transforms marking dependent expressions to marking dependent functions. For instance, a reward expression generated by the *sm2srn* transformation engine based on the derived failure modes places is transformed to a marking dependent function in the target CSPL model. This transformation also generates all CSPL mandatory functions and other required variables. For example, the *net()* function holds all CSPL statements for creating the SRN structure while the *final()* function contains the statements for solving the model by using the derived reward rate function.

The *cspl2text* engine is a model-to-text transformation implemented to generate the CSPL code based on the CSPL model derived by *srn2cspl*. As explained in Chapter 6, a direct transformation from SRN to code could be very complicated without the *srn2cspl* intermediate model-to-model transformation that reduces the gap between the derived SRN model and the actual CSPL language used by the SPNP solver.

Figure 7.11 and Figure 7.12 show snapshots from the derived CSPL model and CSPL code, respectively. The user is required to manually add closing arcs before passing the generated CSPL code as input to the SPNP solver. This is necessary for closing the Petri nets model, to obtain a K-bounded SRN with a finite state space. These arcs are SRN output transition arcs that connect the exit SRN transitions with the workload source (a place derived from a UML transition stereotyped with *<<GaWorkloadEvent>>)*. These exit transitions represent the end of the scenario, and

can be easily located in two steps. First, for all component internal behaviors identify all UML transitions representing the end of the scenario processing, either normal or erroneous. The next step is to determine the corresponding derived SRN transitions for all UML transitions found in the first step.

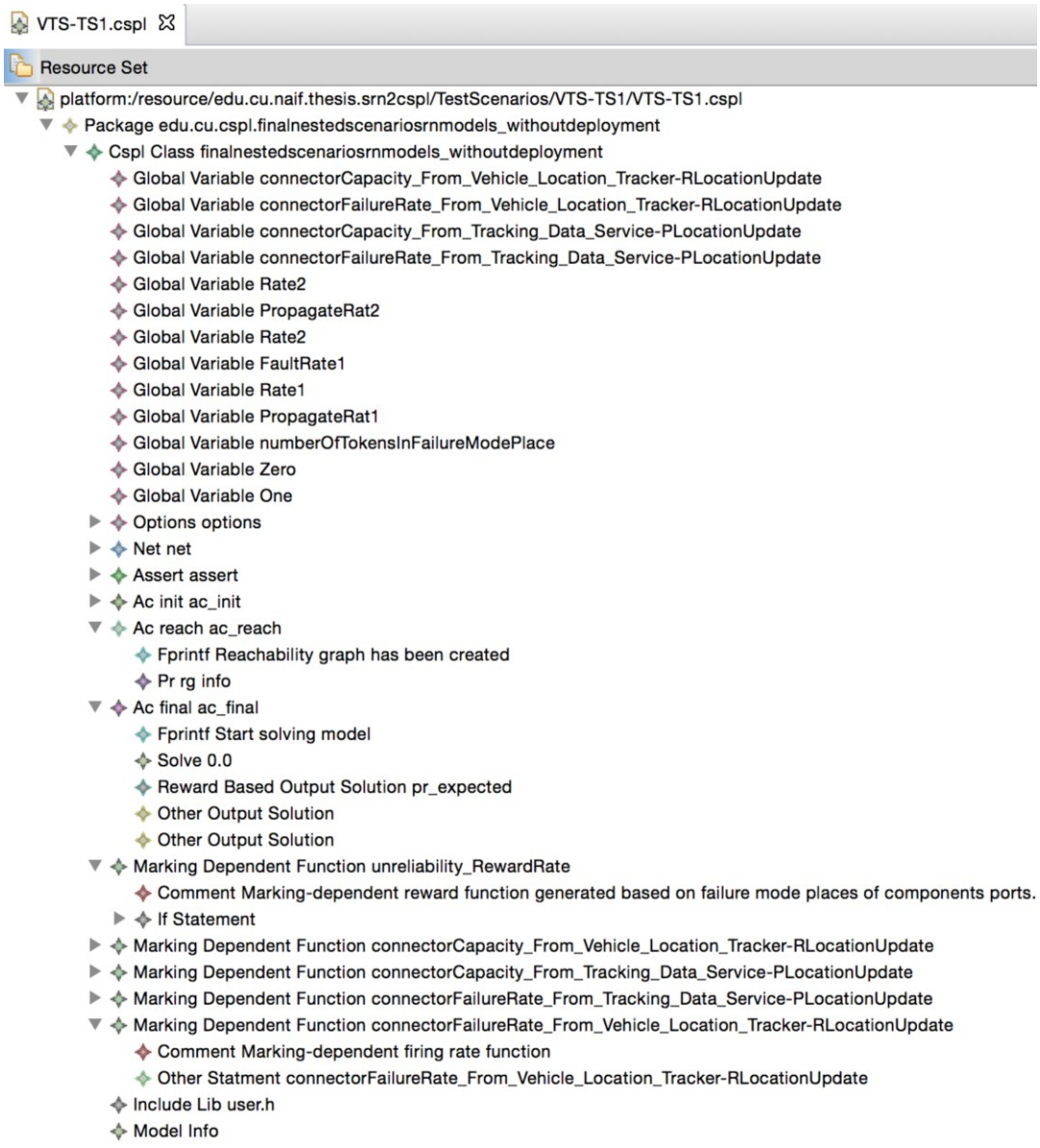


Figure 7.11: CSPL Model from the Derived SRN Model

finalnestedscenariosrnmodels_withoutdeployment.c

```
int  assert() {
   /* Now in our case we do not have specific condition to verify! later we may have one!! */
   return RES_NOERR;
}


void  ac_init() {

   pr_net_info(); /* information on the net structure */
   fprintf(stderr, "Case Study from my PhD theis");
}


void  ac_reach() {

   pr_rg_info(); /* information on the reachability graph */
   fprintf(stderr, "Reachability graph has been created");
}


void  ac_final() {

   fprintf(stderr, "Start solving model");

   solve(INFINITY);
   /* Reward Based Output Solution*/
   pr_expected("expected unreliability_Reward Rate is : " , unreliability_RewardRate );

  /*Other Output Solution*/
   pr_std_average();
   pr_mc_info();
}


double  unreliability_RewardRate() {
   /* Marking-dependent reward function generated based on failure mode places of components ports. */
    if(   return   mark("P2_failureMode_Remote_System_Down") >= numberOfTokensInFailureModePlace  ||
                   mark("P2_failureMode_Unable_To_Update") >= numberOfTokensInFailureModePlace  )
        return( One );
    else
        return( Zero ); ;
}

double  connectorCapacity_From_Vehicle_Location_Tracker-RLocationUpdate() {
   /* Marking-dependent firing rate function  */
    return connectorCapacity_From_Vehicle_Location_Tracker-RLocationUpdate ;
}
```

Figure 7.12: Snapshot of Generated CSPL Code for VTS Case Study

Chapter 8: Modeling Fault Tolerance Tactics with Reusable Aspects

This Chapter presents the Single Version Fault Tolerance Aspect Modeling (SvFTAM) approach that models Single Version-Fault Tolerance tactics (SV-FT) as generic reusable aspects. The main contribution of this chapter is threefold. First, we introduce the SvFTAM approach that applies fault tolerance tactics to the UML software architecture and behavior using the AOM approach. In a case study, we show how to model SV-FT tactics and then present an automated process for aspect instantiation and composition with the basic UML model, which represents not only the normal behavior, but also the erroneous behavior and failure propagation following the CeBAM approach introduced in Chapter 4. Secondly, we discuss what new transformation rules are necessary to map the elements of a fault-tolerance tactic to SRN, in order to extend the original SRN analysis model with the respective fault-tolerance tactic, giving the checkpoint transformation and hardware failure propagation as an example. Third, we illustrate how to approach the analysis by solving and comparing the derived SRN models before and after applying fault tolerance.

The chapter is organized as follow. Section 8.1 presents an overview and shows the VTS case study after correcting modeling errors identified in Chapter 7. Section 8.2 describes SvFTAM by modeling three SV-FT tactics. Section 8.3 illustrates the process of aspect instantiation and composition, whose effect is to refactor the case study model by adding the spare checkpoint tactic to the original design. Section 8.4 discusses how to approach the analysis of the derived model and what transformation rules are necessary to map the elements of the fault-tolerance tactic to SRN.

8.1 Overview

Software fault tolerance techniques can be categorized in three groups. The first is design diversity or multi-version techniques, where many replicas with different implementations are developed from the same specifications, but by different teams and different programming languages. Examples are the recovery block and N-version programming. The second group, data diversity, uses identical replication, but each replica will be executed with different data generated by data re-expression mechanisms from the original data. Example of this category is N-copy programming [11], [99]. SV-FT is the third category, based on redundant software modules that can detect erroneous state and apply recovery actions, such as restarting or switching to a redundant spare module deployed on a different node. Exception handling, checkpoint, restart, and process pairs are examples of SV-FT techniques [104].

Increasing redundancy by identical replication is a common approach for fault tolerance in hardware. According to [11] this approach is not applicable to software, which is deterministic and thus each replica receives and processes the same data; instead, design diversity needs to be used. However, the work in [12], [13] introduces a new thinking in software fault tolerance based on environmental diversity as opposed to design diversity. Software bugs are classified into two categories: Bohrbugs and Mandelbugs. The former is manifested consistently under known conditions and should be fixed during testing, while the later is hard to reproduce and it has complex error propagation. In [12] it is shown that the failures caused by Mandelbugs are more predominant. Restart, reconfigure and reboot are techniques employed to recover from Mandelbugs. In practice, availability tactics presented in [114] depend on redundancy or

retry of a single version. Indeed, design diversity is not widely adopted in practice due to its high cost and effort, being used only for mission critical systems.

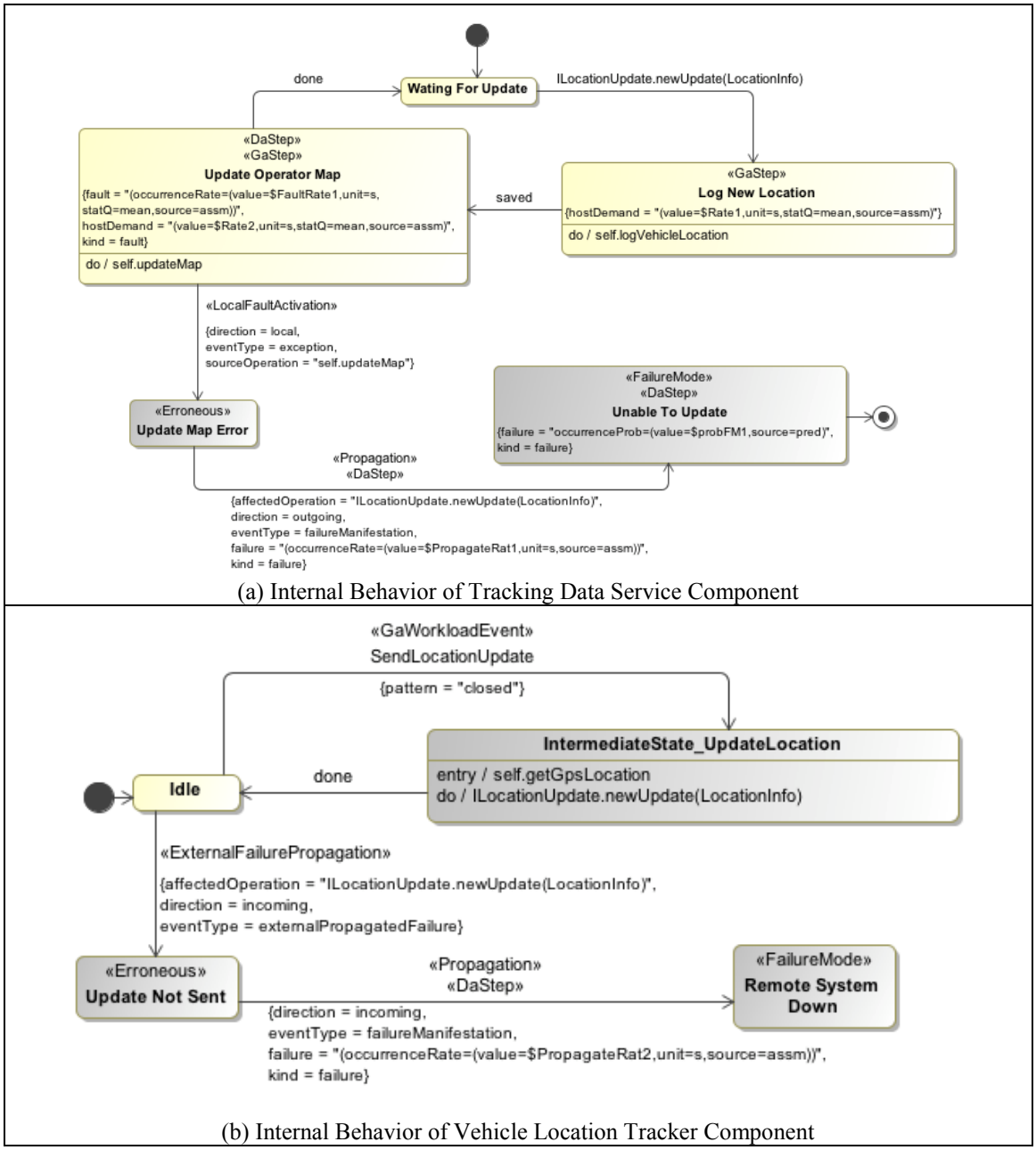


Figure 8.1: VTS Case Study after Correcting Modeling Errors and Applying CeBAM Approach

Adding any kind of fault tolerance mechanism is expected to improve the system reliability and availability, but the effects are non-trivial due to the dependency of such

mechanisms on the software context [2]. In fact, each fault tolerance technique needs to be customized and tailored to the application using it. Our proposed approach aims to provide quantitative data for supporting an easy comparison of different fault tolerance tactics, in order to select the best solution for a given system.

In this chapter, we address the above issues by introducing the SvFTAM approach that models single version fault tolerance tactics as generic reusable aspects. Our work has been inspired by the new vision of software fault tolerance introduced in [12], [114]. We illustrate our approach with four reusable fault tolerance tactics: spare with checkpoint, standby spare, restart and retry. A generic aspect is instantiated and then its parameters are bound to the application context. The resulting context-specific aspect models are then composed with the original design model.

Applying fault tolerance to software architecture usually requires adding new components and modifying the existing ones, therefore the overall architecture changes and becomes harder to maintain. To overcome this issue, we replace a component without fault tolerance with a single composite component which contains the original component (possibly replicated) and a fault tolerance manager, preserves the original interfaces and has fault tolerance capabilities to recover from internal manifested failures and to prevent failure propagation.

To explain the SvFTAM approach we use the running case study (VTS). We correct all modeling errors identified by *CebamWeaver* and *sm2srn* transformation engines as shown in Chapter 7. As mentioned earlier we focus on periodically sending vehicle location update as the most critical scenario. Any failure in this scenario will affect the system availability and reliability. The architecture model is already presented

in Chapter 4. Figure 8.1 shows the internal behavior models of components involved in the scenario. For the sake of simplicity, we model only one failure mode (represented by the states Update Map Error and Unable to update). Note that other models such as port protocol behavior are already shown in Chapter 4 and 5. However, applying SvFTAM will not modify the port behavior of the original components that have passed the conformance and compatibility verification. The VTS system reliability and availability is an important non-functional property. Therefore, to achieve high reliability and availability, we have to avoid any single point of failure (such as the Tracking Data Service component) by adding a fault tolerance mechanism to the initial design, as described in the following sections.

8.2 Single Version Fault Tolerance Aspect Modeling

Single version fault tolerance tactics [25], [114] have been widely used as best practices to improve system reliability and availability. However, there is a lack of modeling approaches able to represent these tactics in a generic reusable form, along with dependability annotations, which can be used in early software design phases. One of the main challenges is to customize and tailor the generic models to the software context in order to get accurate results in terms of dependability improvements. In order to model SV-FT tactics as reusable models we use AOM [118] to describe the structure and behavior of the selected tactics. In SvFTAM the generic structure and behavior of fault tolerance tactics is captured according to the CeBAM approach [116]. In order to reuse these generic tactics, we employ model transformations to automate the instantiation of context-specific aspects and to compose them with the base model.

In any fault tolerance mechanism, four basic actions are taken to tolerate an erroneous state. First, the error detection action determines the erroneous state. Next, during the processing phase, two actions focus on assessing the damage, identifying the cause of the error and restoring the system to its normal state. The last action uses the recovered state to continue the service and the normal operation [11], [25], [99]. All reusable single version fault tolerance tactics in our approach are modeled according to CeBAM [116] that captures normal and erroneous behavior. The detection mechanism in our approach is started by the primary component, which sends a notification message (i.e., raising an exception), to the fault tolerance manager. In such a case, a recovery action will be taken by switching the control to the redundant component.

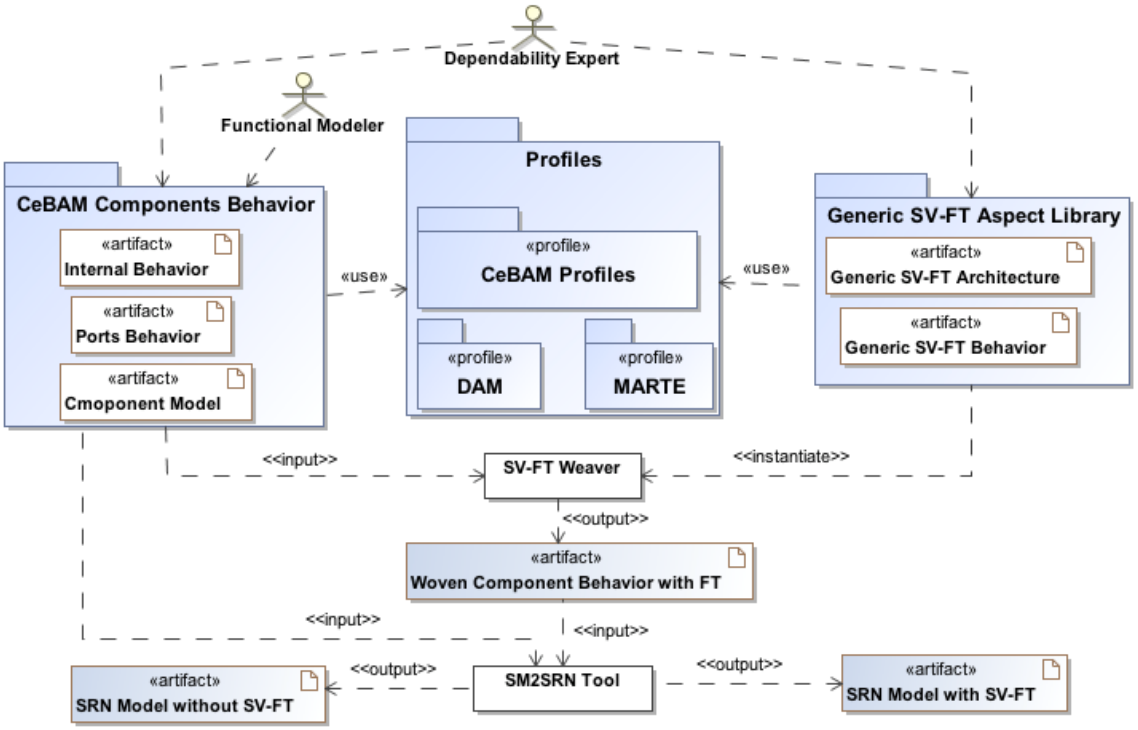


Figure 8.2: SvFTAM Overall Approach

We created the *AspectComponent* UML2 profile to model fault tolerance tactics as generic reusable aspects annotated with dependability attributes. We consider single

version fault tolerance tactics as crosscutting concerns, which can be applied to different components in the system. A point cut is a query that identifies the join point(s) in the base model where the aspects should be composed. To fully automate the process of selecting and composing aspects, we specify a point cut as an OCL query string in the *AspectComponent* profile. During the composition process, the OCL query is passed to an OCL parser implemented as a QVT-O black-box module, for parsing and executing the OCL query [139]. As shown in Figure 8.2, the generic aspect template of a selected tactic is instantiated and the template parameters (which have names starting with "|" symbol) are bound to context values, thus obtaining an application-specific aspect that can be composed with the base model.

As illustrated in Figure 8.2, the original design (architecture and behavior) is built by the developer according to the user requirements. A dependability expert augments the initial design in two phases. First, he/she captures the erroneous behavior and applies the required dependability attributes using the CeBAM, MARTE, and DAM profiles [4], [116], [117]. As shown in Chapters 4 and 6, erroneous behavior is modeled as a context specific aspect, which is composed with the normal behavior using model transformation techniques. The next phase - which is the focus of this chapter - is to refactor the software architecture and behavior by applying fault tolerance tactics to the most critical components. Different fault tolerance reusable aspects can be selected from a predefined library. A SV-FT weaver automates the process of obtaining the context-specific aspect and composing it with the base model. The composed model (architecture and behavior) will be passed to another transformation chain (*sm2srn*) to derive the SRN analysis model. Note that we present only the design and the architecture of SV-FT weaver

transformation and we apply it manually to the case study. However the QVT-O implementation of SV-FT weaver transformation can easily extend the transformation engines presented in Chapter 6 and Chapter 7.

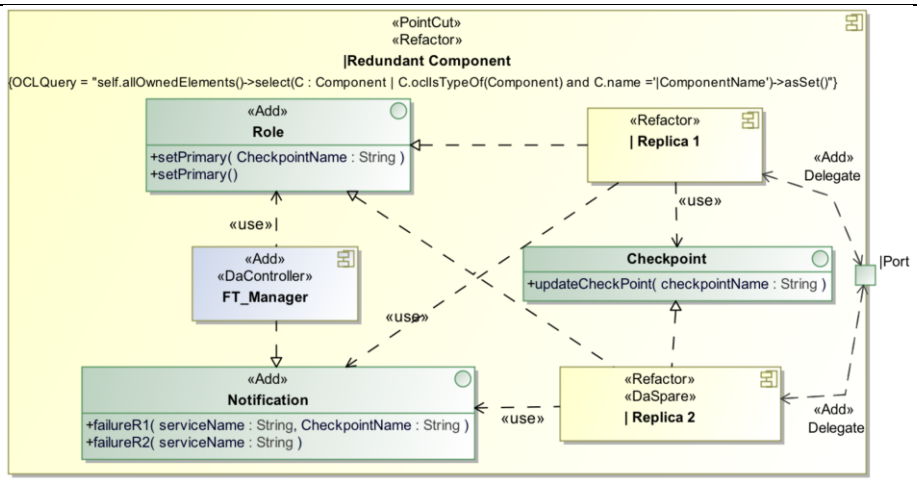
Adding fault tolerance will introduce more complexity in the software models. To mitigate this issue, SvFTAM replaces the most critical component in the scenario with a composite component, preserving the original interfaces; the new component embeds a set of components working together to provide fault tolerance capabilities. In the following examples, it contains two identical replicated components offering functional services, which are copies of the replaced simple component. These replicas are deployed on different nodes and their internal behavior is refactored to support fault tolerance actions (i.e., failure notification). In addition, the new component contains a fault tolerance manager component dedicated to managing fault tolerance behavior by detecting failure notifications and switching between replicas in case of hardware and/or software failure.

8.2.1 Spare with Checkpoint Tactic

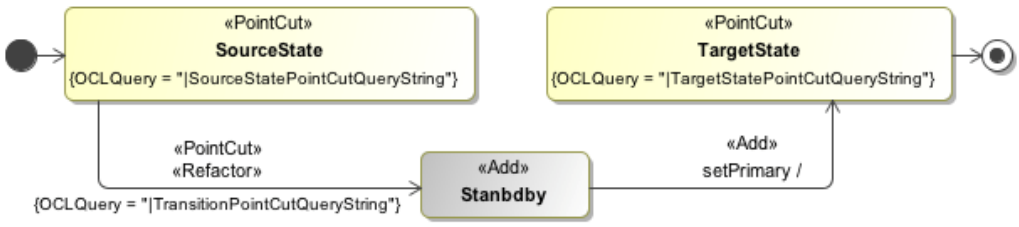
This tactic, also known as *warm spare,* was originally described in [114]. It has an active component that periodically updates the state of the redundant spare using a checkpoint mechanism. The architecture changes include a new auxiliary component for detecting failure, as well as modifications to the component internal behavior to send checkpoint synchronization to the other replica. Our proposed SvFTAM approach will limit the changes to just one component and keep all the other dependent components unchanged. Figure 8.3(a) captures the structure of this tactic as a generic reusable aspect model.

As already mentioned, the aspect model is a single composite component including two replicas that provide the functional services specific to the application. The fault tolerance manager component manages the fault tolerance behavior, detecting failures and switching requests between replicas. The primary replica will send a checkpoint update to the secondary replica. In case of a failure, the primary component will notify the fault tolerance manager about the manifested failure. As a recovery action, the fault tolerance manger will switch the control to the second component, which will resume the execution from the most recent updated checkpoint. The fault tolerance behavior is executed inside the composite component and does not affect the other components from the original software architecture. In modeling this tactic's structure, we use the *AspectComponent* profile that is a part of CeBAM profiles set, as well as the DAM profile for dependability annotations [4].

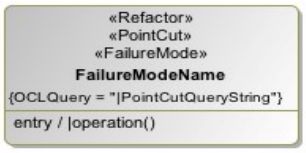
The main composite component stereotyped *PointCut* is used by the transformation tool to identify the critical component, as specified in the *OCLQuery* attribute. It is also stereotyped as *Refactor*, which will guide the transformation to replace the selected component in the base architecture model with a composite component. Note that the port(s) of the replaced component will not change and we just add delegation connections from the main port to each replica to pass the incoming or outgoing messages. Therefore, there are no changes in the other original component(s), which are unaware of the component replacement which is the effect of the aspect application. New activities and actions must be added to the original behavior of both replicated components to support fault tolerance. In order to automate refactoring the internal behavior of the replicated component we develop four generic aspects models as shown in Figure 8.3(b, c, d, e).



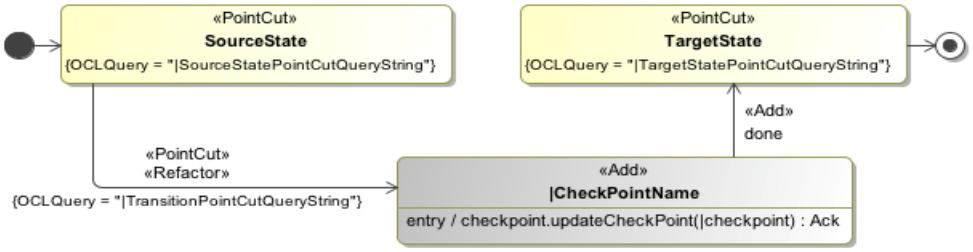
(a) Composite component aspect of spare with checkpoint tactic



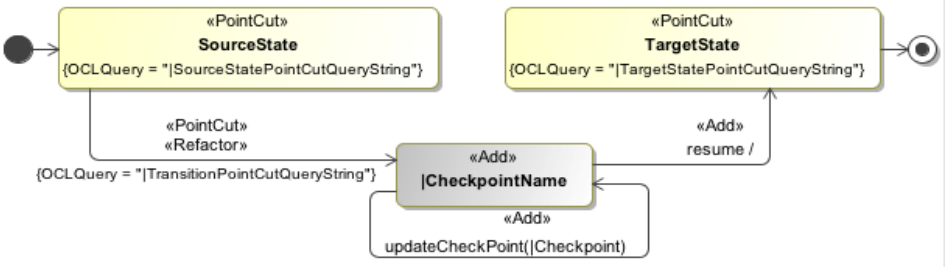
(b) Standby Refactor Aspect



(c) Failure Mode Refactor Aspect



(d) Sender Checkpoint Refactor Aspect



(e) Receiver Checkpoint Refactor Aspect

Figure 8.3: Spare with Checkpoint Tactics: Structural and Behavior Aspects Models

First, the *standby refactor* aspect is instantiated once to modify the behavior of both replicas to start initially in standby mode and remain in that mode until receiving a *setPrimary* message from the fault tolerance manager. This aspect model has two states stereotyped with *pointCut* that are used to identify the source and target states in the base model. As explained earlier, the string value of the *OCLQuery* attribute will be passed to the OCL parser, which returns the join point in the base model. A similar technique is used for the transition stereotyped by *Refactor* to either replace a model element or to modify its attributes. It starts by identifying the transition in the base model using *OCLQuery* and then modifies its destination to the new added state (i.e., *Standby*). In addition, any model element stereotyped with *Add* will be treated by the transformation as a new behavior that needs to be added to the base model. Figure 8.3(c) shows the second refactor aspect, dedicated to refactoring failure modes states by adding an entry operation, which is used to notify the fault tolerance manger about the failure. In consequence, the control is either switched to the spare, or failure propagation is allowed in the case of failure in the spare component.

The *Role* interface is implemented by each replica to provide a service invoked by the fault tolerance manager to set the primary component. The main difference between *setPrimary()* and *setPrimary(ChcekpointName)* operations is that the former will be called during the initialization of the component, while the later will be called after a failure to resume the service from the last updated checkpoint. In fact, the implementation of these interfaces depends on the fault tolerance manager behavior. For instance, in our case the first replica will be always the primary component, until a failure

occurs, in which case the second replica will resume the service from the most recently updated checkpoint.

In this tactic, the primary component will keep the spare component updated by sending periodic state updates. Figure 8.3(a) shows the checkpoint interface implemented by the second replica, which is the checkpoint receiver. In order to capture this behavior, we need to refactor the internal behavior of both replicas, by adding a checkpoint state in similar join points. However, the first replica will act as checkpoint sender, while the second will act as receiver. Therefore, we develop two refactor checkpoint aspects. It is the responsibility of the dependability modeler to insure that the checkpoint states are added in similar join points using OCL queries. Figure 8.3(d) shows the sender refactor aspect, which has an entry action for sending a checkpoint synchronization request to its counterpart.

The call is synchronous, as the checkpoint sender waits for an acknowledgment from the receiver. A special transformation rule that maps the checkpoint behavior to SRN is presented in section 8.4. The fault tolerance manager has two main tasks: it acts as a failure detector receiving failure notifications from the replicas and it decides which component is the primary and when to switch the control to the standby component in case of unrecovered failure. Both replicas start in the standby mode, until the fault tolerance manager sends the *setPrimary* message to the first replica. The primary replica keeps the standby replica updated by sending checkpoint updates, as already mentioned. Any incoming message will be passed to both replicas, but only the primary component will handle the request and the standby component will simply ignore it. A failure will propagate to other outside components only if the second replica fails, too. Software

repair behavior is not considered in this tactic. However, the failed component due software failure will be repaired only if the hardware restarted. The fault tolerance manager behavior and the number of checkpoints can be customized according to the software context. For instance, it can detect the failure of the primary component due to hardware failure and then it switches to the second active redundant component.

8.2.2 Standby Spare Tactic

The structure of the standby spare tactics has two identical redundant components: one of them active and the second in standby mode. If the primary replica fails, then the fault tolerance manager will switch the control to the standby component to start handling the request form the beginning. Indeed, such behavior is suitable for systems with high reliability requirements [114]. The structure is modeled as a reusable aspect template by following similar concepts as in the previous tactic (see section 8.2.1), but without adding checkpoint interface between the two replicas. For the behavior we apply the same standby refactor aspect and failure mode aspect from Figure 8.3(b) and Figure 8.3(c), respectively. During fail over to standby component, the fault tolerance manager will call the *setPriamry(serviceName)* method implemented by the second replica to change its mode to primary and to start over the processing for the failed service from the beginning based on the service name passed as a parameter. A special transformation rules is implement to capture this behavior in the analysis model.

8.2.3 Retry and Restart Tactics

In principle, the previous tactics depends on replicating identical software components on different deployment nodes, without modifying the software behavior. It helps to avoid the effects of Mandelbugs by masking the failure and trying to process the

request again on a different node hosting another instance of the same software component. However, the retry tactic depends on an error detection mechanism that identifies the erroneous state and retries the failed action before the failure mode manifested. This behavior can be modeled using CeBAM by including the retry action in the erroneous aspect model.

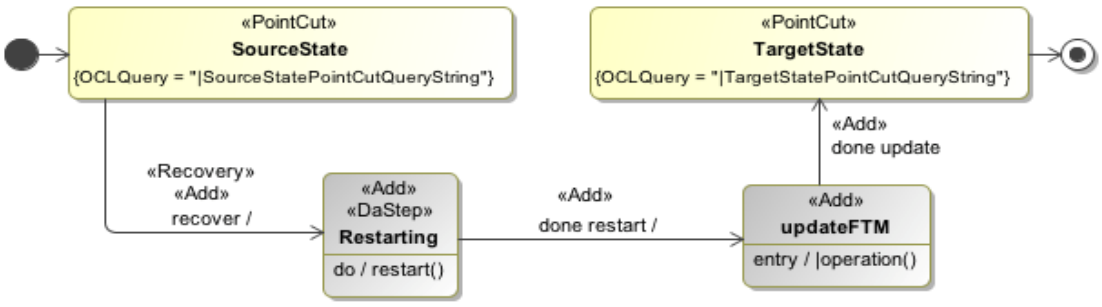


Figure 8.4: Restart Aspect

The restarting tactic is a similar single version FT tactic that forces the software component to restart if any local failure mode is manifested. The restart behavior is modeled as a separate aspect that can be applied to the component internal behavior as shown in Figure 8.4. In this aspect the source state point cut will be the failure mode state while the target point cut state will be the initial state of the component internal behavior.

8.3 Aspect Composition

The proposed process of using aspects for refactoring the software architectural and behavioral models in order to add fault tolerance capabilities is illustrated in Figure 8.5. As discussed in the previous sections, single version fault tolerance tactics can be modeled as generic reusable aspect models, which are then instantiated and their parameters bound to application specific values, producing context specific aspect models.

We choose to specify the point cuts as OCL queries, represented by string attribute of the stereotype *PointCut*. For instance, Figure 8.3(a) shows a parameterized OCL query expression as a string. Before the instantiation, the modeler will provide values for the template parameters to build up the complete query (e.g., the component name). During the aspect composition, the refactoring engine will invoke the *OCLParser* to parse and execute the OCL expression for finding the join points.

In the case of the VTS system we refactor the base model (architecture and behavior) by adding the fault tolerance tactic "spare with checkpoint". At the architecture level we just need to replace the *Tracking Data Service* component with a composite component that supports fault tolerance, keeping all the other dependent components unchanged. The modeler is expected to provide values for the template parameters such as component name and OCL query parameters. The refactoring engine will use the *AspectComponent* profile stereotypes as composition directives to guide the refactoring process. It starts by looking for *PointCut* stereotype and passes the *OCLQuery* string to the *OCLParser* black-box code to parse and execute the OCL expression for identifying the join points in the base model. In this case, the query should return at least one component (i.e., *Tracking Data Service*).

The next step in the transformation algorithm is to replace the join point element with a composite component (instantiated as a context specific aspect) which is stereotyped with *Refactor*. Note that this replacement will preserve the existing ports and connections with the other original components. An internal redundant component stereotyped with *Refactor* is in fact a copy of the replaced component in the original model. Every model element stereotyped with *Add* will be added as a new model element

in the final woven model. Figure 8.6(a) shows the refactored architectural model obtained as a result.

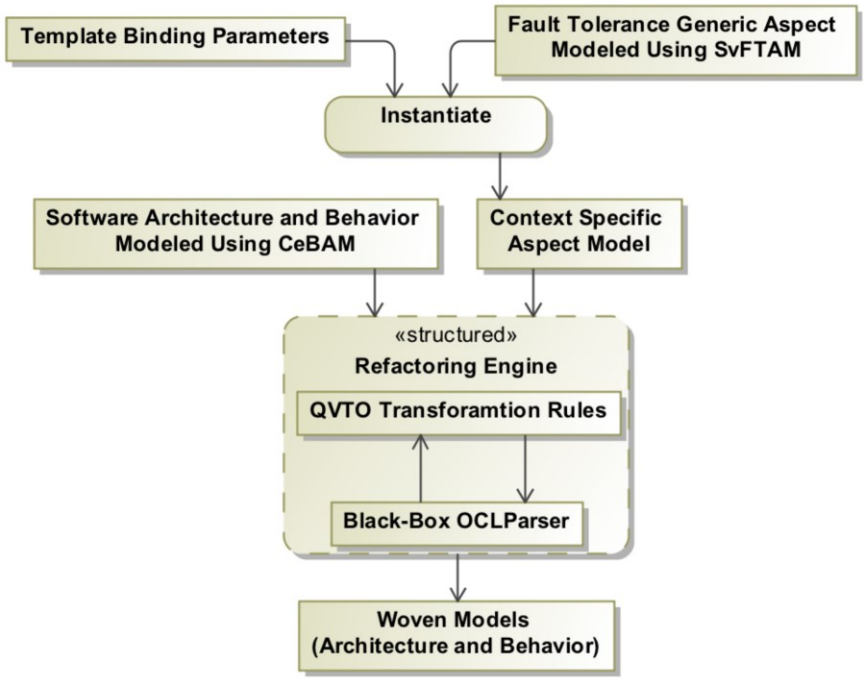
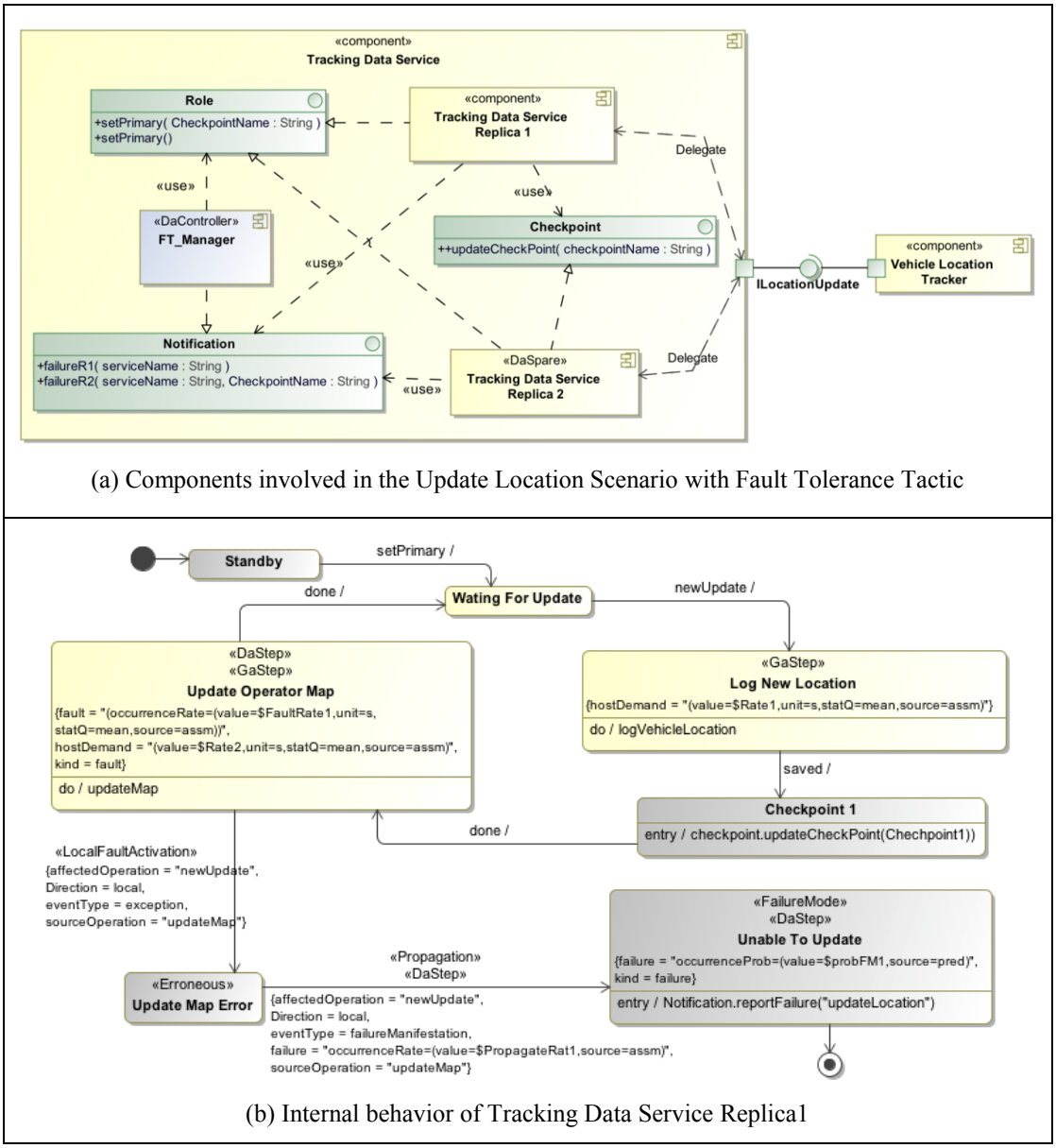


Figure 8.5: Refactoring Aspect to Add Fault Tolerance

The internal behavior of both replicated components is similar. It is refactored by three aspects. The first aspect will add a *standby* state after the initial *PseudoState* and a new transition triggered by the event *setPrimary* sent by the fault tolerance manager. The second aspect will refactor all failure mode states by adding an entry operation to notify the fault tolerance manager about the failure manifestation. The last aspect is the checkpoint aspect, which will be added in the same joint point to each replica, but with different behavior. *Tracking Data Service Replica1* acts as a checkpoint sender, while *Tracking Data Service Replica2* acts as a receiver. To ensure that the checkpoint is added in the same join point in each replicas we verify the result of *PointCut* OCL query during composition. Figure 8.6(b) shows the internal behavior of the contained primary redundant component. The behavior of the second redundant component will be identical

to the primary component, except for the checkpoint state, which receives updates rather than sending them.



(a) Components involved in the Update Location Scenario with Fault Tolerance Tactic

(b) Internal behavior of Tracking Data Service Replica1

Figure 8.6: Applying Spare with Checkpoint Tactics to VTS Case Study

The dependability model of a system describes the failure and repair process of each component and also captures the failure propagation between software components, as well as failure propagation from a hardware node to the hosted software. In practice,

software engineers model the normal behavior of the system and ignore the erroneous behavior due to its complexity. In our previous work [116], [140] we introduced a modeling approach that utilizes Aspect Modeling to separately model component erroneous behavior annotated with dependability information using the MARTE and DAM profiles [4], [117] (see Chapter 4 and 5). We extend this approach here to model fault tolerance patterns as generic aspects that can be applied automatically to any design. The purpose is to help developers to quantitatively compare different design alternatives, in order to estimate the improvement brought by different fault tolerance tactics in terms of reliability and availability.

8.4 Dependability analysis

We developed a tool based on QVTO and Acceleo [130], [139] that perform a set of model-to-model and model-to-text transformations to derive an SRN dependability analysis model (Chapter 6). The proposed analysis approach is carried out through the following steps: 1) apply erroneous aspect to the normal behavior of each component; 2) iteratively derive SRN models for each component and compose them according to the system architecture; 3) verify conformance and compatibility between software components by analyzing the structure of the derived SRN; 4) extend the SRN model by adding deployment SRN subnet; 5) generate CSPL code from SRN model (where CSPL is the input language to the SPNP solver [113]). The SvFTAM patterns can be instantiated and applied to the original design after applying erroneous behavior aspects.

DAM profile allows software engineers to specify the output dependability measures of interest [4]. These measures are computed by solving the derived analysis model to get results that can be interpreted to improve the system design. We are

interested here in unreliability and instantaneous availability of the system. According to our approach, the derived SRN model describes the healthy states along with erroneous behavior and failure propagation. Considering unreliability, i.e., the probability that the system has failed by time *t*, we just focus on failure mode states of the system. On the other hand, for the instantaneous availability we need to compute the probability that the system does not arrive to any failure mode by time t. Both of these measures are computed using transient analysis. For instance, if *Pi(t)* is the probability of the system being in state *i* at time *t*, then the unreliability is computed by summing the probabilities that the system is in any state *i* whose corresponding marking contains at least one token in a failure mode place [19].

One of the advantages of SRN is the ability to define a reward rate function for the system states of interest. To compute the unreliability of the system, we define a reward rate function as:

URi = if (#(P_failureMode_i) >= 1 || #(P_failureMode_j) >= 1) 1 else 0.

This function includes all possible failure modes in the system. The unreliability is determined by performing first transient analysis to compute *Pi(t)* of each state and then compute the reward rate function given above. The reliability is given by *R(t)=1-UR(t)*. The instantaneous availability and mean time to failure measures are computed in a similar way, by defining a reward rate for each measure of interest. In SRN, rewards can be used in conjunction with both transient and steady state analysis.

8.4.1 SRN Derivation Rules

SRN is a variety of stochastic Petri Net that has some interesting features such as reward rates and marking dependency. In SRN any tangible marking can be associated

with a reward rate. Moreover, the marking dependency is an essential characteristic of SRN that allows for defining model parameters as a function of the number of tokens in particular places [18]. Marking dependency was introduced for the convenience of the modeler, as it may simplify the model specification. For instance, arc multiplicity, transition guard and firing rate can be defined with marking-dependent feature to simplify the graphical model.

As mentioned in the introduction, the ultimate objective of this thesis is to automatically derive dependability analysis models from annotated UML software models, in order to predict dependability properties (such as reliability and availability) of the software architecture in the early development stages. In our previous work [140] (Chapter 5) we proposed a set of transformation rules for deriving SRN analysis model from an annotated UML software model without fault tolerance capabilities and without considering the fault assumption of the deployment nodes.

In this chapter we propose an aspect-based approach for extending a software model with fault tolerance tactics. Since we intend to derive also the SRN model of such a system, we need new transformation rules for translating from UML to SRN the elements of the fault tolerance tactic added to the original software model. For instance, if we consider the "spare with checkpoint" tactic, we need transformation rules for the following elements of the tactic: setting the primary component, checkpoint synchronization, failure notification, and switching control to other replica. Among these transformation rules, the one for failure propagation from a deployment node to the hosted software behavior and checkpoint synchronization are the most complex rules, so we will briefly discuss both in this section.

The initial derived analysis model represents only the software Platform Independent Model (PIM) since it does not include the failure specification from the hardware nodes. The derived SRN model needs to be refactored to derive Platform Specific Model (PSM). We use UML component and deployment diagram to add a SRN subnet for each node as shown in Figure 8.7(b). Each software component is allocated to a hardware node, therefore a hardware failure propagates to the software, causing the loss of any request that is being processed or cached. In such a case, the two SRN subnets must be synchronized in a way that the software SRN subnet model goes to the failure mode if the hardware node is down and it goes back to the initial state if the hardware node is repaired. In order to represent this synchronization without any additional complexity, we make use of a SRN feature that can simplify our model, namely guards to model failure propagation from the SRN hardware subnet to the allocated software subnet.

In Figure 8.7 we have two SRN subnets: subnet (a) is the derived SRN model of the *Tracking Data Service* component internal behavior and subnet (b) the failure and repair behavior model of the hardware node hosting this component. To model the failure propagation between hardware node and hosted software we add a set of guarded transitions as shown in Table 8.1. For instance, the *t6_node1Down* transition that has [*Gf*] guard is fired only if the hosted node SRN subnet is in failure mode (i.e. *P2_node1Down* has a token). During the normal operation of the software the SRN subnet will be in one state at a time; if the hardware node goes down the guarded transition attached to that state will be enabled causing the loss of the request and switching to failure mode. Moreover, once the hardware node recovers, the guarded transition *t11_node1Up* is fired,

allowing the software SRN subnet to be started again from the initial place to model software starting up after hardware recovery.

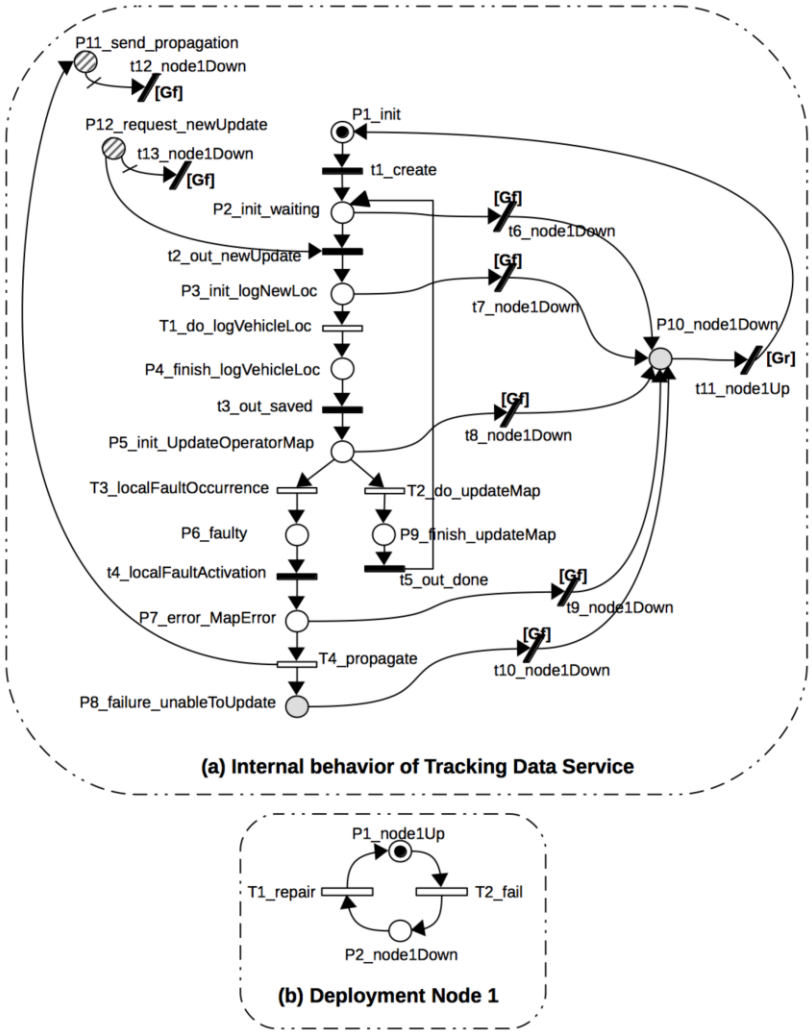


Figure 8.7: Derived SRN Model for Tracking Data Service with Deployment

Table 8.1: Hardware Failure propagation SRN guards

Guard Name	Function
Gf	if (#(P2_node1Down) == 1) 1 else 0
Gr	if (#(P1_node1Up) == 1) 1 else 0

The striped places in subnet (a) will be used to compose the component internal behavior with its port protocol state machine. These places represent the event pool of the

internal state machine and it may contain many tokens that represent incoming and outgoing requests. Failure of the host node will trigger the flushing out transition of all pending requests. This is modeled by a guarded immediate transition with an input arc with marking dependent multiplicity, to be enabled once the host node is down.

As explained in Section 8.2, a checkpoint may be added in different places of the main behavior of the replicas in order to synchronize their data state. In software, a checkpoint synchronization message is periodically sent from the primary replica to the spare replica along with the checkpoint name as a parameter. We need to map this semantic to the SRN model, to have a generic checkpoint mechanism without introducing extra complexity in the analysis model. Tokens in SRN models do not carry any information, so we cannot use them to carry parameters. However, we use SRN guards again for synchronizing transitions. Figure 8.8 shows a generic SRN model for checkpoint synchronization using a set of transition guards.

The transition *t_entry_updateCheckpoint1R1* of the sender component (Replica1) will send a checkpoint synchronization request by depositing a token in the shared place called *P_request_checkpoint*. We assume that the communication between replicas is immediate and never lost. In the software model the type of checkpoint message is synchronous, therefore the sender component will wait for an acknowledgment from the replicated component. To map this semantic, we add the guard [*Gs1*] to *t_Ack1R1* transition of the sender component that will prevent it from firing until a token is deposited in the *P_init_Checkpoint1R2* place of the receiver component. Another approach to model the acknowledgement could use a return path from Replica1 to Replica2, but this would add more places and transitions to the model.

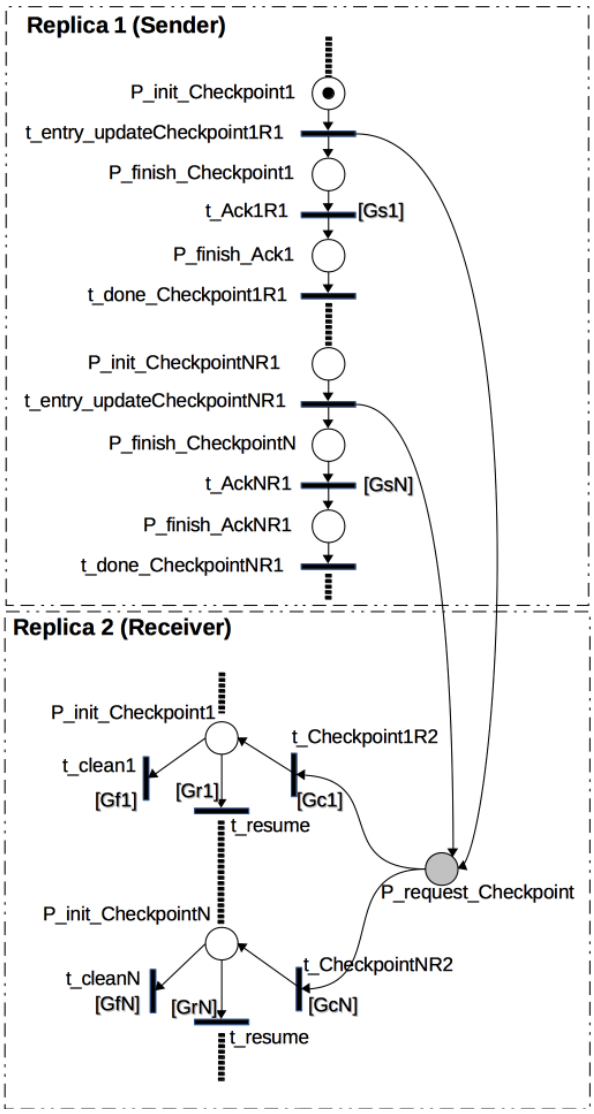


Figure 8.8: Checkpoint Synchronization SRN Model

Table 8.2: Checkpoint Synchronization SRN Guards

Guard Name	Function
Gs1	if (#(P_init_Checkpoint1) == 1) 1 else 0
Gc1	if (#(P_finish_Checkpoint1)== 1) 1 else 0
Gr1	if (#(P_inti_Primary) == 1) 1 else 0
Gf1	if (#(P_init_Checkpoint2) == 1 or .. #(P_init_CheckpointN) == 1) 1 else 0

In the receiver (Replica2), a token will be placed in *P_request_checkpoint* place that represents the receiving of a checkpoint update message, but this token does not carry information indicating which checkpoint place should take it. In order to deposit the

received token in the correct checkpoint place that matches the place in the sender component, we add a *t_checkpoint$_i$R2* transition (where *i=1,N*) for each checkpoint place in the receiver component. Each transition has a guard that checks the marking of the corresponding checkpoint in the sender component, as shown in Figure 8.8 and Table 8.2. Now just one *t_checkpoint$_i$R2* transition will fire to receive the message. Transition *t_clean$_i$* (where *i=1,N*) and its associated guard is added for each checkpoint transition to flush out the token once a new checkpoint update was received from the sender component and for every new request processed by the primary replica.

In case of failure manifested in the primary replica, the fault tolerance manager will change the state of the second replica to primary and then a *t_resume$_i$* transition will be enabled to complete the execution of the request from the last updated checkpoint.

8.4.2 Setting SRN parameters

We use different UML diagrams to model the system. Component diagram(s) along with deployment diagram capture how the software components are composed and what are the provided and required services. Moreover, it shows the deployment of the software component instances on hardware nodes (see Figure 4.1(a) and Figure 8.7). Behavioral state machine describes component internal behavior, while extended protocol state machine describes the provided and required services along with failure propagation. The component diagram will be used to guide the composition of the derived SRN model from each component state machine. MARTE and DAM profiles are used in our approach to augment the UML design with annotation dependability specifications that will be mapped to SRN parameters [4], [117].

Each deployment node is transformed to an SRN subnet that models the failure and repair of each node in the system, as shown in Figure 8.7(b). The annotations applied on this model are *DaComponent*, *DaConnector* and *GaCommHost*. *DaComponent* is applied to a hardware node to describe the aspect failure and repair. The transition rate of *T2_fail* is mapped to the *failure.occurenceRate*, and the rate of *T1_repair* to repair tagged-value. *DaConnector* and *GaCommHost* both describe connector specification: that the first captures the failure rate of the connector, while the second describes the connector capacity that is used to compute the transfer time between software components.

In our approach, the behavior model encompasses normal and erroneous component behavior. According to CeBAM, all transitions are atomic without any action. Figure 8.6(b) show the internal behavior state machine of *Tracking Data Service* component. This model is transformed to a SRN model as shown in Figure 8.7(a). The striped places will be used to compose the component internal behavior with the port's protocol state machine. In fact, all of these places together model the event pool of the state machine. *DaStep* and *GaStep* are applied on state activities such as *entry, do, exit* to annotate them with the processing demand, as well as with fault activation occurrences rate if an error propagation chain is attached to the state. For example, "*Update Operator Map*" state has *do/ updateMap* activity and an output transition to the error propagation state starting with "*Update Map Error*" state until the failure mode is manifested. In this state we use *hostDemand* attribute of *GaStep* stereotype to specify the processing time of the *updateMap* activity. This activity will be transformed into a timed transition called *T2_do_updateMap* as shown in Figure 8.7(a), whose rate is mapped to the value of *hostDemand*. Additionally, *DaStep* stereotype is applied to the same activity

(*updateMap*) to annotate the fault activation rate. According to the transformation rules presented in [140] (see Chapter 5) the fault activation of an activity is transformed to a time transition. In VTS case study, the timed transition *T3_localFaultOccurrence* represents the fault activation occurrence rate of *updateMap* activity and its rate is mapped to *occurrenceRate.* In some cases, a state is annotated with *GaStep* only, since the error propagation is not modeled (e.g., "*Log New Location*" state).

DaStep is applied to two other model elements of the component internal behavior state machine: a) to the propagation transition specifying the time between the switching to erroneous state and the failure manifestation (translated to the SRN timed transition *T4_propagate*); and b) to the failure mode state specifying the occurrence probability of failure model state).

8.4.3 Analysis of Results

We used the SPNP tool for solving the derived SRN model [113]. For the case study of this thesis we use the analytical solver to compute the unreliability using transient analysis. The values assigned to the input parameters of the *Tracking Data Service* component and its deployment node is shown in Table 8.3. Similar parameters are assigned to each similar component and connectors. All timed transitions have exponential distributions. The generated SRN model is based on the following assumptions:

- Components fault occurrences are independent;
- Failure modes of each component are mutually exclusive, therefore, if the component fails with a given failure mode then it is considered failed and no other failure mode may occur until it is repaired;

- Once a local fault occurs with a given rate, it switches to the erroneous state immediately;
- The fault tolerance manager is deployed on a node that has negligible failure rate;
- Communication between the internal components of the composite component is immediate with no delay;
- Network failure is recoverable.

Table 8.3: Parameters of Tracking Data Service component

SRN Transition	DAM Parameter	Assumed Rate
T1_do_logVehicleLoc	$Rate1	1/(15 s)
T2_do_updateMap	$Rate2	1/(40 s)
T3_locaFaultActivation	$FaultRate1	1/(2700 s)
T4_propagate	$PropagateRate1	1/(10 s)
T1_repair	$RepairFreq1	1/(300 s)
T2_fail	$FailFreq1	1/(604800 s)

In this example we focus on unreliability (failure probability) analysis, trying to answer two questions: first, what impact have the different SvFTAM tactics on the system unreliability; and second, what is the best SvFTAM tactics to be applied for the particular system in terms of reliability improvement. As explained in the previous sections, the analysis model is automatically derived before and after applying fault tolerance tactics and then solved to get quantitative data for comparison. Figure 8.9 shows the unreliability (failure probability) results of VTS case study before and after applying SvFTAM tactics. It is clear that the system failure probability is lower after applying any SvFTAM tactics. This answers the first question and the quantitative results will encourage the designers to consider such tactics for improving the reliability of the system. In addition, the collected results will guide the developers in the comparison of

different SvFTAM tactics and help them select the best one in terms of reliability improvement. For instance, for the values chosen for the failure and repair rates, the retry tactic is the best option for the VTS case study. For different failure and repair rates, the comparison results may be different.

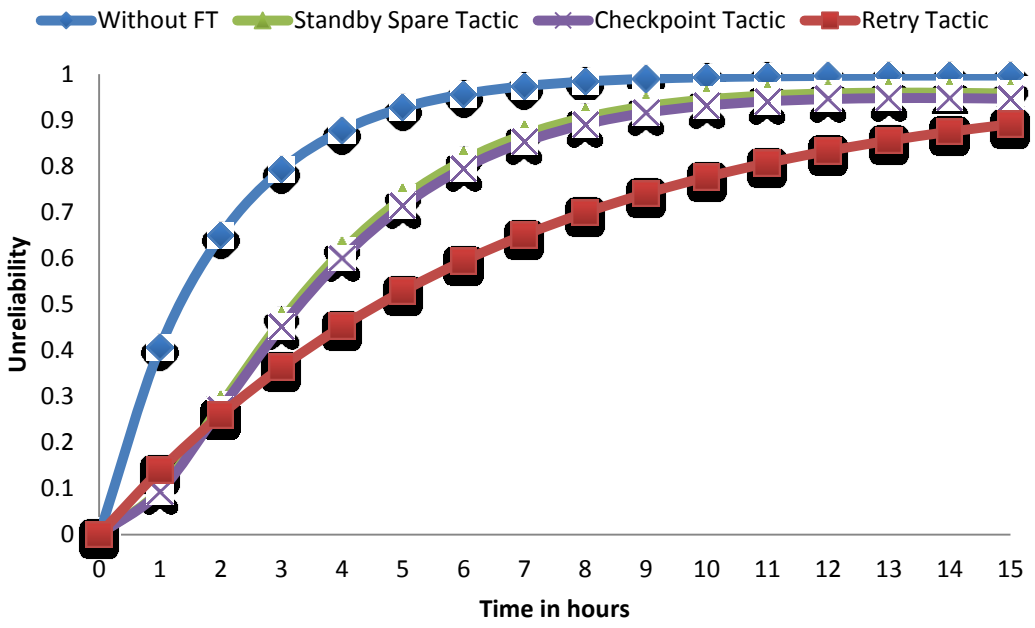


Figure 8.9: Unreliability of VTS Case Study with SvFTAM Tactics Compression as Function of Time

Checkpoint tactic shows a slight improvement compared to the standby spare tactic. The difference is not very significant in this particular case study, due to the size and simplicity of the VTS case study, which has only a single checkpoint synchronization between the two redundant components. According to these values a noticeable improvement will be clearer in a bigger system that has multiple checkpoints. Standby spare tactic is ranked low compared to other tactics. In this tactic after switching to the second redundant component, the failed request starts from the beginning and fault may again reappear in any operation. On the other hand, retry and checkpoint tactics continue from the last failed operation.

Chapter 9: Decomposition and Reduction Techniques for the Derived SRN Model

The previous chapters present our approach to automating the derivation of SRN model from annotated software behavioral models. We employ Model-Driven Architecture (MDA) techniques namely model-to-model transformations and model-to-text transformations to derive the SRN model from annotated software architecture and behavioral models. However, solving derived SRN model analytically remains challenging for intermediate and large software that involve many components. The major problem with the analytical solution of SRN model is a large state space of underlying Markov Chain. This problem known as state space explosion makes the analytical solution impossible for realistic systems [15], [122], [141]. In such case, simulation is the only way to solve the derived SRN model. The major disadvantages of the simulation are long execution time and estimated results.

In this chapter, we provide a decomposition and reduction technique for the derived SRN model to compute approximate system reliability and availability measures. Our approach is inspired by the work of [15], [123] where iterative SPN decomposition techniques are introduced to compute performance measures. We adopt their idea of constructing auxiliary models and solving them using the delay equivalence approach. However, our approach uses the decomposition of the software into components to guide the decomposition of the derived SRN model. It first identifies the SRN subnets of each component internal behavior, then applies SRN structure reduction rules to construct a new reduced and aggregated SRN model of the component internal behavior. Next, a set of SRN auxiliary models is constructed. Each one has an original component internal

SRN model, and the other components are represented by aggregated subnets. Auxiliary models are solved iteratively based on the delay equivalence between the original SRN subnet and the corresponding aggregated subnet from another auxiliary model. In the last step, we build an approximate model that contains all aggregated subnets to be solved analytically in order to compute the scenario reliability and other measures.

We develop an algorithm to identify the eligible SRN subnets for reduction. This algorithm uses the SRN compositional model elements between the component internal behavior and the attached component port subnet to identify the service execution path. Accordingly, a suitable structure reduction rule will be identified to construct the aggregated subnet. Each aggregated subnet will preserve the original subnet behavior such as liveness and boundedness. In order to compute system reliability, we build an approximate model that has all aggregated SRN subnets constructed in the auxiliary models, and it has the deployment SRN subnets with failure propagation from the hardware node to the hosted software component. In fact, system reliability and other measures can also be computed from any auxiliary model, as each represents a different approximation of the entire system states. However, we selected to build an approximate model that preserves symmetry by having only reduced SRN subnets of all components.

This chapter is organized as follows. In section 9.1 we give an overview of the decomposition and reduction technique. In section 9.2 we introduce a new illustrative case study. Reduction rules are presented in section 9.3. Section 9.4 has the detailed explanation of the proposed technique applied to the illustrative case study. Derivation formulas for all reduction rules is provided in section 9.5. Sections 9.6 and 9.7 show the derivation of formulas for the case study and experimental results.

9.1 SRN Decomposition and Reduction Technique Overview

Our ultimate goal is to find a design technique to compute approximated reliability and availability measures for larger derived SRN models and to alleviate the state space explosion problem. The proposed SRN decomposition and reduction approach is inspired by the iterative delay equivalence approach presented in [15], [142].

Our approach has three phases as shown in Figure 9.1. In the first phase, we identify the candidate SRN subnets for reduction. According to our transformation technique each software component has one SRN subnet that represents the component internal behavior, derived from the annotated component UML behavioral state machine model. In this subnet, each service has an execution path. It starts from receiving a request from the component port SRN subnet and ends by sending an acknowledgment, propagating failure, or going back to the component initial state. Reduction rules are not applied to the whole internal component derived SRN subnet. Instead, they are applied to the service execution path. We propose an algorithm to identify these paths as candidate subnets to be aggregated using reduction rules such as SISO, SM or And-Or that will be explained in section 9.3. In the identification process, we are utilizing SRN compositional elements introduced by our transformation rules (see Chapter 5) such as places *P_sent* and *P_rquest*. The result of this phase is a reduced component internal behavior that preserves the component original behavior. The derived SRN subnets of components ports and connectors are not changed. These subnets are simple and cannot further be reduced.

In the second phase, we build auxiliary models based on the number of software components. These models will be solved in parallel in different iterations to compute the

new constant rates of the aggregated transitions. Little's law and delay equivalence concepts are employed to derive the solution formulas. Moreover, the solution technique is adopted from [15]. In the last phase we construct the approximate model to obtain reliability and availability measures of the system or scenario. This model has the deployment node subnet and it has the failure propagation from hardware node to the hosted software, as already explained in Chapter 8. The subsequent sections will present these phases in great details with equation derivation. Also, we apply the proposed approach to a case study, and compare reliability results between the original derived SRN model and the approximate model.

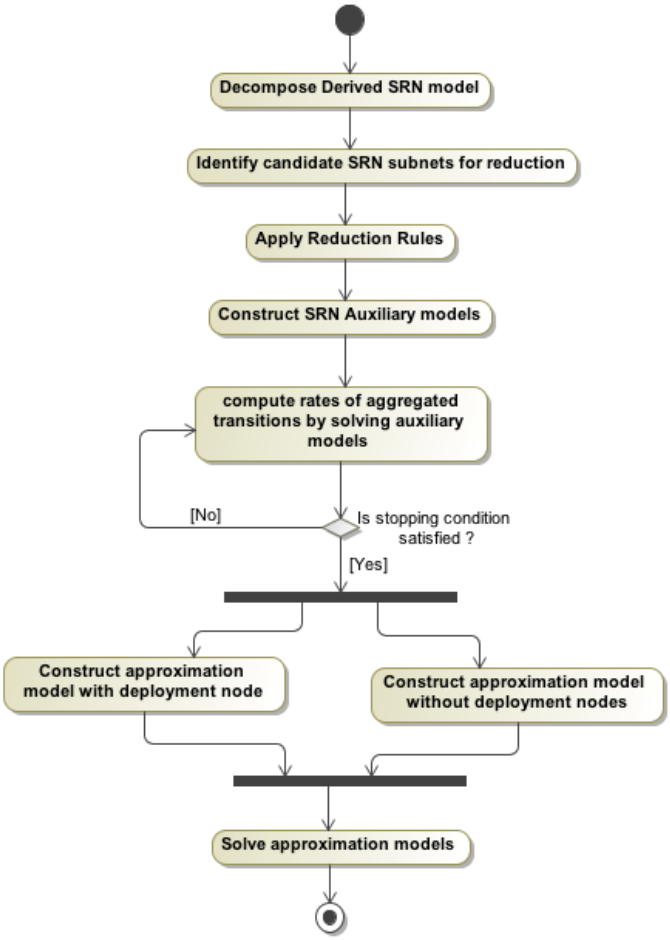


Figure 9.1: SRN Reduction Approach

9.2 Illustrative Case Study

To show the impact on state space reduction of the derived SRN model on a fairly large critical scenario, we developed a new case study called Field Monitoring System (FMS). We use it to show the application of reduction rules on the derived SRN model for each component. Also, we use it to illustrate how to construct auxiliary models as well as approximate model. This case study has three components working together to report collected data from the field to the operator as shown in Figure 9.2.

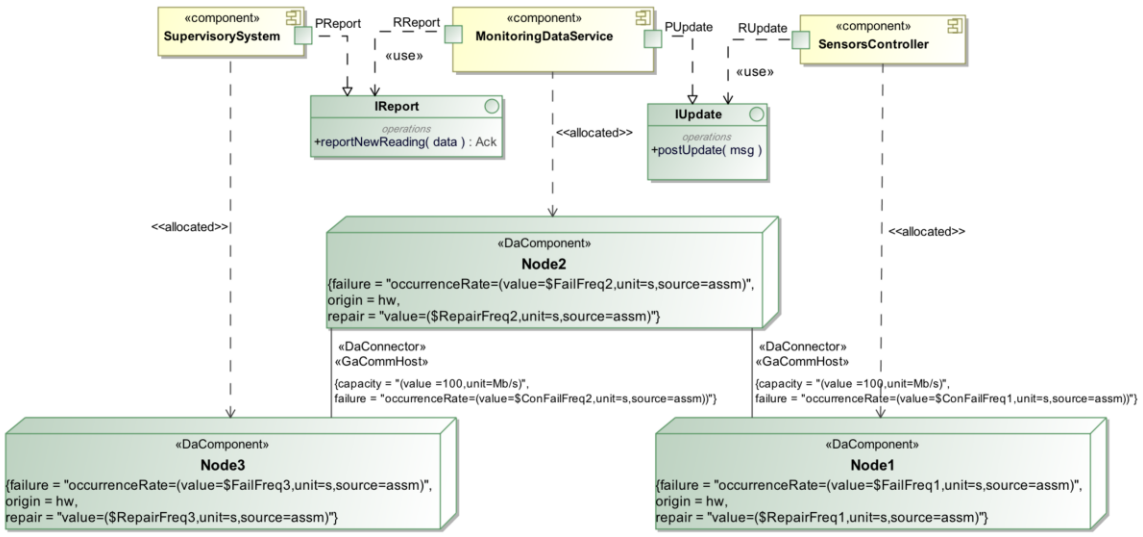


Figure 9.2: Field Monitoring System Architecture

The Sensors Controller component collects data from distributed sensors and pass it to the monitoring data service component using the *postUpdate(msg)* provided service. The reported data will be formatted and sent to the supervisory system using the *reportNewRading(data)* to be displayed to the operator. We assume that every component is deployed on different machines that have different failure rates. We use the CeBAM approach to model component internal behaviors and port protocol behavior. In Appendix C, we show each component internal behavior and ports protocol behavior after applying the CeBAM approach. However, in this chapter we show the derived SRN model for

each component and how we apply the decomposition approach. In addition, we show the derivation of the constant rate formula to solve the auxiliary model using the SPNP solver.

9.3 SRN Reduction Rules

Reduction rules define the ways of aggregating a PN subnet into a simpler one without losing the original subnet properties. A PN model is constructed from a set of the elementary subnets (subclasses of PN) such as Single Input Single Output (SISO), State Machine (SM), and And-Or subnets. A large PN model is sometimes impossible to solve analytically. A solution is to reduce it to a simpler model. To construct a reduced model, first we identify all possible elementary subnets and then apply the reduction rules on each subnet. The resulted reduced model preserves the structural and behavioral properties of the original system [15], [123].

According to [16], a PN is K-bounded if the number of tokens in each place does not exceed a finite number K for any marking. A K-vector y of nonnegative number can be called a place invariant or S-invariant if $y \cdot \mathbb{T} = 0$, where $\mathbb{T}$ is the incident matrix of the Petri net. Therefore, a place $\boldsymbol{P}$ is covered by an S-invariant if $y > 0$ with $y(\boldsymbol{P}) > 0$. If all places are covered by S-invariant y, then we call it conservative and $y(\boldsymbol{P})$ we call it weight of the place. PN is *alive* if every transition in the net is ultimately friable by progressing through some firing sequence for every marking. A PN subnet is *regular* if its input is places and output is transitions. A Subnet is *empty* if there are no tokens inside it and is *effectively empty* if no more tokens can flow out by firing any transition inside it. These behavioral and structural properties are valid for the derived SRN model since the number of the tokens will not exceed the number of jobs of the closed system that was

captured using <<*GaWorkloadEvent*>> stereotype. Moreover, it is *deadlock* free since the conformance and compatibility verification is performed during the composition of the complete system SRN model.

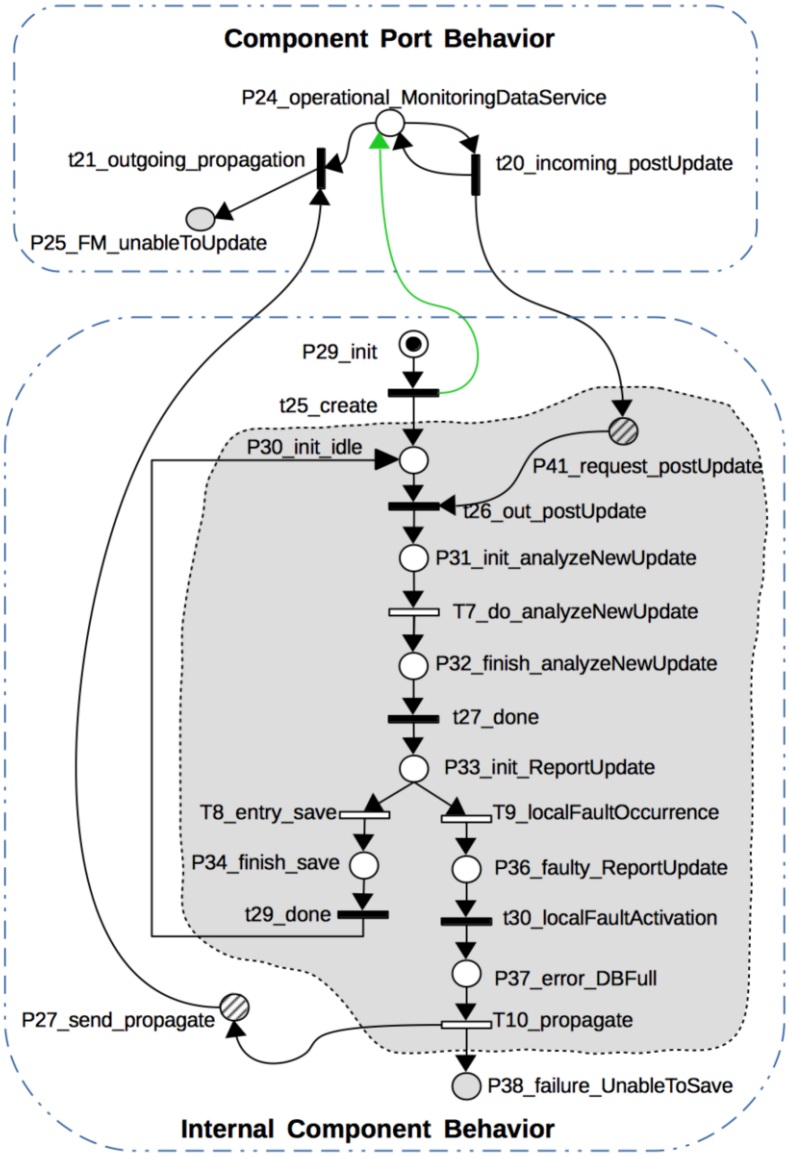


Figure 9.3: Component Port and Internal Derived SRN Subnets

Figure 9.3 shows the SRN model of a single component that consists of two subnets: the component port subnet and the component internal behavior subnet. Places *P41_request_postUpdate* and *P27_send_propagate* are compositional elements added to

capture sending and receiving requests between the component internal behavior and its port. The highlighted subnet in the internal behavior has two input places, *P41_request_postUpdate* and *P30_init_idle,* and two output transitions, *t29_done* and *T10_propagate*. This subnet is *regular* since its inputs are places and output are transitions. It is also an *empty* subnet because there is no token inside it. Furthermore, this subnet represents the service execution path, starting from the receiving request into the *P41_request_postUpdate* place, which represents the event pool semantics and it is K-bounded. It ends in either failure propagation through transition *T10_propagate* to be propagated to the other component, or in the initial place if the fault is not manifested. All service paths subnets in our approach are *regular* and *empty*. The UML State Machine property to be in one state $s \in S$ at a time leads by construction to the corresponding SRN property that a request token will be in one place $p \in P$ at a time, where P is the set of places derived from S. Therefore, the generated SRN subnet of the service execution path is 1-bounded. This means that the places in the derived SRN subnet of the service execution path are covered by an S-invariant. How to identify such a subnet and match it with the reduction rule is explained in detail in the subsequent sections.

One of the advantages of our SRN derivation approach is using the knowledge about the software architecture to compose the system SRN model. We defined a set of compositional rules to glue the component internal behavior with its ports and to compose the components' provided and required services through connector subnets. We utilize this feature to identify the SRN subnet that can be reduced without losing the original correct behavior. From the literature, we have found different structural reduction rules such as those presented in [14]-[16], [123]. We studied these rules and

selected those that can be applied to our derived SRN model. The following subsections present the selected rules that are used in the proposed reduction and approximation approach.

9.3.1 Rule 1: Single Input Single Output Subnet Reduction

The Single Input Single Output (SISO) subnet is a regular and empty subnet. As shown in Figure 9.4 it has one input place and one output transition. This subnet is flow conserving since each token passed to its input place will eventually come out. This subnet can be aggregated into one place and one timed transition [15], [16]. The input place is preserved since it is connected to an outside source transition, while the internal behavior of the subnet (service request execution time) is replaced by one timed transition. It represents the time spent in the original subnet to process the incoming request.

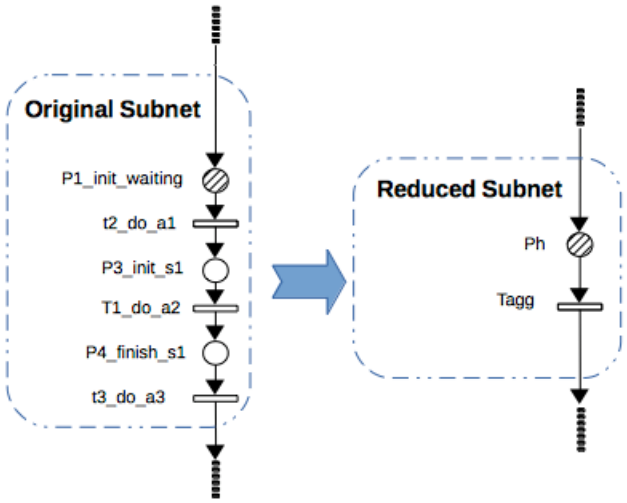


Figure 9.4: Single Input Single Output Subnet Reduction Rule

9.3.2 Rule 2: State Machine Subnet Reduction

By definition, every transition of a State Machine (SM) subnet has only one input place and one output place [16]. The entire subnet is regular, since it has one input place

and one or more output transitions. According to [123] such a subnet can be reduced to just one input place and several output transitions. As shown in Figure 9.5 the reduced structure preserves the input place that is connected to one or more source transition(s). It also preserves all the output transitions and replaces them with timed transition attached to the single input place. Each of these timed transitions represents the execution delay of each path of the corresponding path in the original subnet.

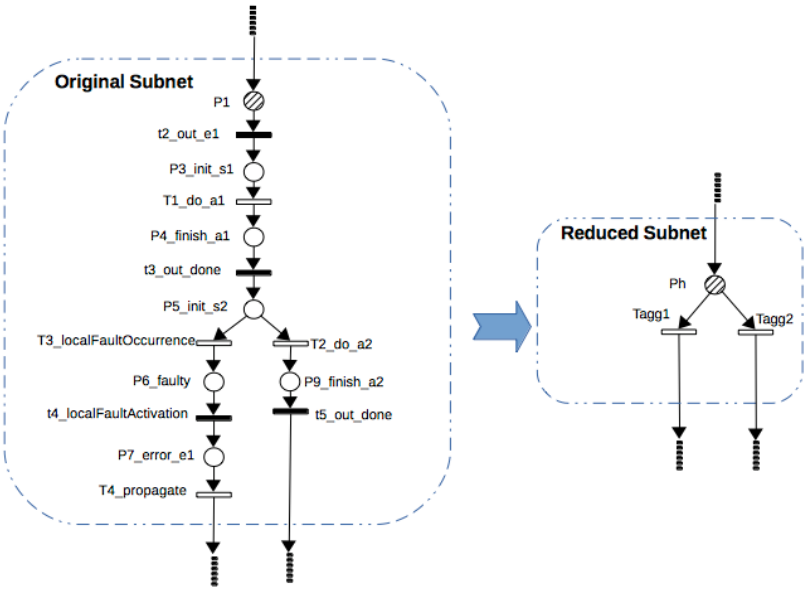


Figure 9.5: State Machine Reduction Rule

9.3.3 Rule 3: And-Or Subnets Reduction

And-Or subnets are a more general class of subnets, including four types: *and:and*, *and:or*, *or:or*, and *or:and*. They have a black box equivalence reduction as explained in [123]. The naming is based on the logical relationships between the subnet input arcs, as well as on the subnet output arcs. In our case, we are interested in *and:and* as well as in *and:or* subnets, since the request will be processed if the two input places of the service execution path have tokens. These tokens are sent out from only one output transition that represents either normal behavior or failure propagation.

The *and:and* subnet has two input places and only one output transition attached to one or more place outside the subnet. Figure 9.6 shows the structure of reducing the *and:and* subnet. The input places are preserved and the internal subnet behavior is replaced by a timed transition, corresponding to the output transition in the original subnet. The *and:or* subnet, shown in Figure 9.7, is reduced in a similar way, except that we replaced the internal behavior with one place attached to two output timed transitions. Each of these timed transitions represents the end of the execution path of the original subnet (normal or failure propagation).

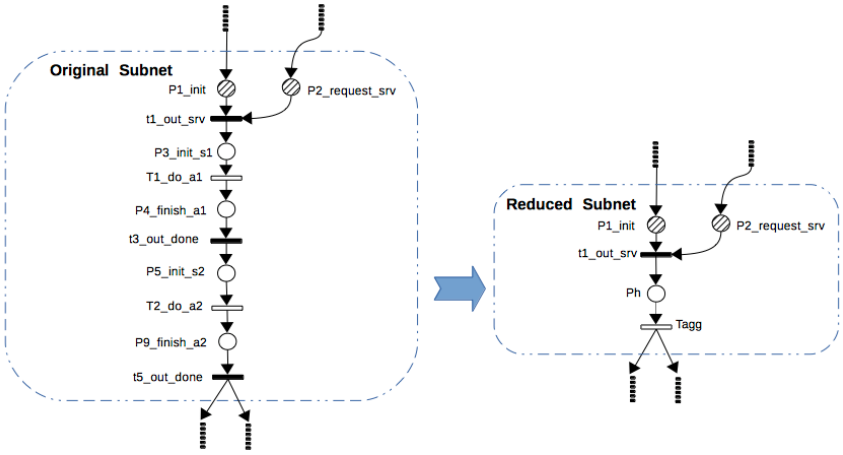


Figure 9.6: and:and Subnet Reduction Rule

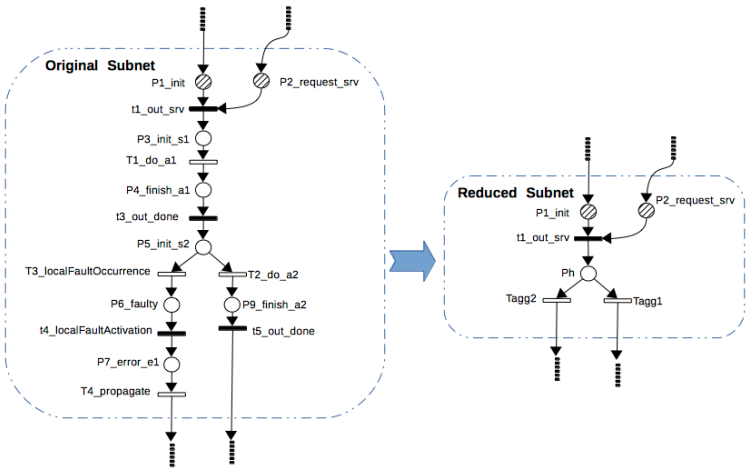


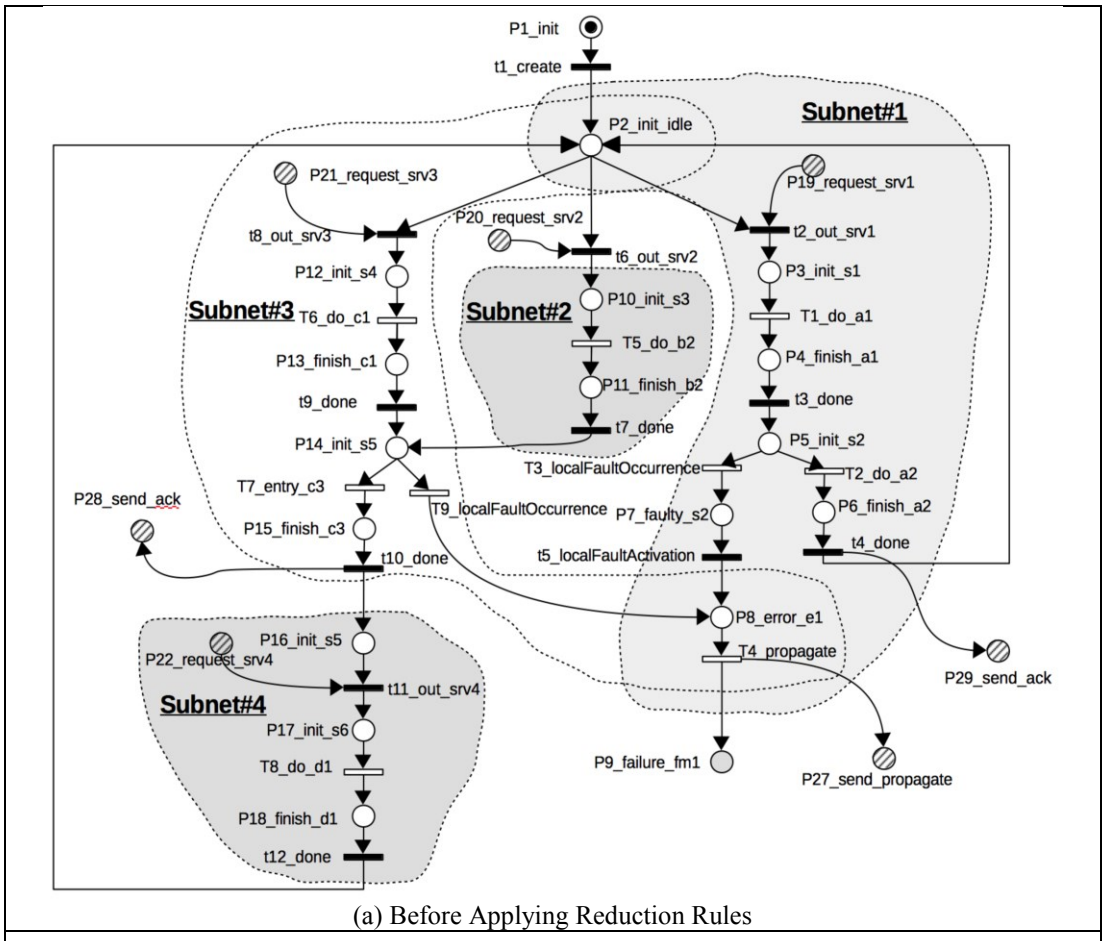
Figure 9.7: and:or Subnet Reduction Rule

9.4 SRN Decomposition and Reduction Approach

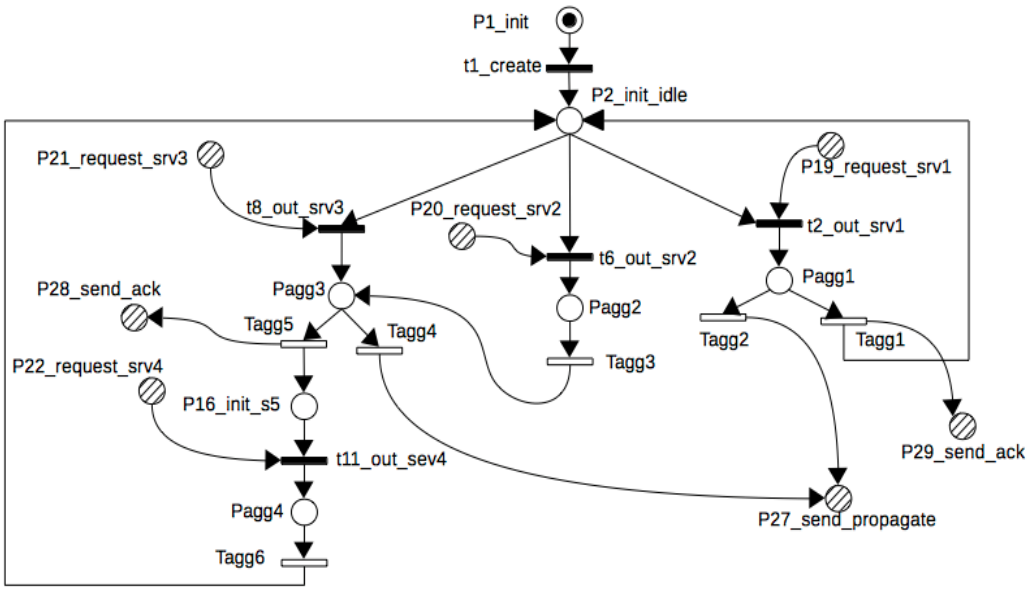
In this section, we present the decomposition and reduction for the derived SRN model. We begin by identifying the candidate SRN subnets and then matching them with a reduction rule presented in the previous section. The next step is to construct the auxiliary models and solve them by using delay equivalence. In the last step we construct a scenario approximate model and we include SRN subnets of deployment nodes to compute an approximation of the unreliability of the system.

9.4.1 SRN Subnet Identification and Auxiliary Models Construction

The software component may provide or require any number of services. According to CeBAM, the component internal behavior captures all possible component states either normal or erroneous (Chapter 4). For each provided service, we have a service path that consists of a series of states and events. As shown in Figure 9.8 some services start from a common initial state *P2_init_idle,* such as *"srv1"*, *"srv2"* and *"srv3"* and other services from an internal state, such as *"srv4"*. The component will be in one state at any point in time; therefore only one service can be executed at a time. On the other hand, the component port is capturing incoming and outgoing messages and modeling two types of state, either operational state or failure mode states. In other words, the component port is just a gate that passes messages from/to the component internal behavior, without caching any message. It encapsulates the status of the component. The derived component port SRN model is abstracted and cannot further be reduced. However, the internal component behavior is the focus of the reduction process.



(a) Before Applying Reduction Rules



(b) After Applying Reduction Rules

Figure 9.8: Component Internal Behavior SRN Subnet with Multiple Services Paths

BEGIN ALGORITHM ***(SRN subnet identification and reduction algorithm)***

FOREACH *(component)*
Identify execution path of each service starting from P_request place
Identify the shared behavior between service execution paths
BEGIN CASE *(applying reduction rule)*

CASE *(only shared initial place)*
Decompose service path
Apply the proper reduction rule
Omit all failure model places
Connect all services from the shared initial place
CASE *(sharing erroneous behavior path)*
FOREACH *(service path)*
Decompose service path
Replicate the shared erroneous behavior path
Apply the proper reduction rule
Omit all failure model places
Connect all services from the shared initial place
Merge the shared P_send_propagate place
END FOREACH

CASE *(sharing part of normal behavior path)*
FOREACH *(service path)*
Identify the longest service path
Identify arc that connect two paths
Decompose service path paths
Apply the proper reduction rule
Omit all failure model places
Connect all services from the shared initial place
Add the arc that connect both paths
END FOREACH

CASE *(sharing normal and erroneous path)*
FOREACH *(service path)*
Identify the longest service path
Identify arc that connect two paths
Decompose service path paths
Replicate the shared erroneous behavior path
Apply the proper reduction rule
Omit all failure model places
Connect all services from the shared initial place
Merge the shared P_send_propagate place
Add the arc that connect both paths
END FOREACH
END CASE
END FOREACH
END ALGORITHM ***(SRN subnet identification and reduction algorithm)***

According to our SRN transformation architecture, the derived SRN model is constructed by a chain of automated transformations based on the software components structure, presented in Chaapers 5 and 6. Our objective here is to apply structural reduction rules to construct a new aggregated SRN model with a smaller state space, which may be solved analytically. The reduction process should retain interconnections of the component internal behavior subnet and its ports subnets, as well as the original software behavior. As a result, our reduction approach focuses on aggregating the component internal SRN subnet only, while keeping the connectors and ports subnets unchanged. Moreover, the internal component SRN subnet is not reduced as a whole, but it is divided into a set of subnets, each representing a software service execution path. Theses paths start from receiving the request through the component port subnet and end by sending a reply message, going back to the initial state, or propagating a manifested failure.

The process of splitting the derived internal component SRN model to a set of subnets to be reduced without losing the original behavior and other subnet properties is hard and complex. However, in our approach we utilize our UML-to-SRN transformation rules to systematically identify the candidate service execution path subnets for reduction. According to the compositional transformation rules, each service execution path starts from the *P_request* and ends to one or more *P_send*, *P_send_propagates* places. We call these places "compositional" (or gluing) places. Therefore, the splitting process is performed by identifying these places to find the boundaries of each service execution path; after that, we study the subnet structure to select the best reduction rule for simplifying the subnet (see the above SRN subnet identification and reduction algorithm).

Figure 9.8 shows an example of derived SRN model for the component internal behavior that has multiple service paths. During the subnet identification process, we may find different possible ways of subnet splitting starting from the common starting place *P2_init_idle*. We compare them and select the best one in terms of size of the resulted reduced subnet. After applying our subnet identification algorithm to this particular example, we have identified four subnets each representing a service path. Three services start from the shared place *P2_init_idle*, except for one service that starts from the *P16_init_s5* place. Additionally, we have common behaviors that are shared between some service paths such as the erroneous behavior between service "*srv1*" (*subnet#1*) and service "*srv3*" (*subnet#3*) as well as the normal behavior shared between services "*srv2*" (*subnet#2*) and "*srv3*" (*subnet#3*). In such case, we treat each service path subnet alone to identify the proper reduction rule and then combine them in reduced subnets according to the original subnet behavior. For instance, the *and:or* reduction rule can be applied to *subnet#1* and *subnet#3* in Figure 9.8. However, the shared erroneous behavior (error and failure states and failure propagation) will be exercised by both services. Therefore, such behavior should be preserved after reduction. The erroneous behavior will be replicated for each subnet (*subnet#1* and *subnet#3*) to be applied to the reduction rule. These two subnets will be combined and the shared elements will be merged such as *P27_send_propagate* and common starting place *P2_init_idle*.

Another case in this example is the shared normal behavior between *"srv2"* and "*srv3"* services. After evaluating the different identified subnets, we have found the best option that covers a larger subnet to keep *subnet#2* starting from the *P10_init_s3* place and ending in the output transition *t7_done* and *subnet#3* boundaries between gluing

places with the component port. After applying the reduction rule for each subnet, we must preserve the output transition from *subnet#2* to the *subnet#3* to preserve the original component internal behavior.

As explained earlier, the subnet identification process starts by identifying the gluing places with the component port subnets namely *P_request and P_send.* These places are used to identify the input places and output transition for each service path subnet. However, in some cases the service path could start from the *P_request* place and not send out any request such as an acknowledgment. In such case, once the service is executed a transition will return the token either to the initial common starting place i.e.; *P2_init_idle* or to the failure mode place. This transition will be considered as a subnet output transition. For example the sub*net#4* is starting from the input places *P16_init_s5* and *P22_request_srv4* and it ends at the output transition *t12_done* since it is connected to the common initial place *P2_init_idle*.

The next phase is to build a set of SRN auxiliary models to be solved in parallel to compute the constant rates of the aggregated transitions. The concept of subnet delay equivalence is applied to solve these models, as explained in the next section. Auxiliary models are constructed according to the software component decomposition. Our transformation techniques derive and construct the system SRN model based on the software components structure. As a result, building SRN auxiliary models becomes easier since the boundaries of each component SRN subnet are clearly identified. Each auxiliary model will be smaller than the original model since it has only one original component SRN subnet, and other components subnets will be aggregated. The idea is to have a complete component SRN model appear only in one auxiliary model. A number of

auxiliary models are based on the number of the components involved in the scenario. For instance, in the FMS case study we have three auxiliary models as shown in Figure 9.9, Figure 9.10, and Figure 9.11.

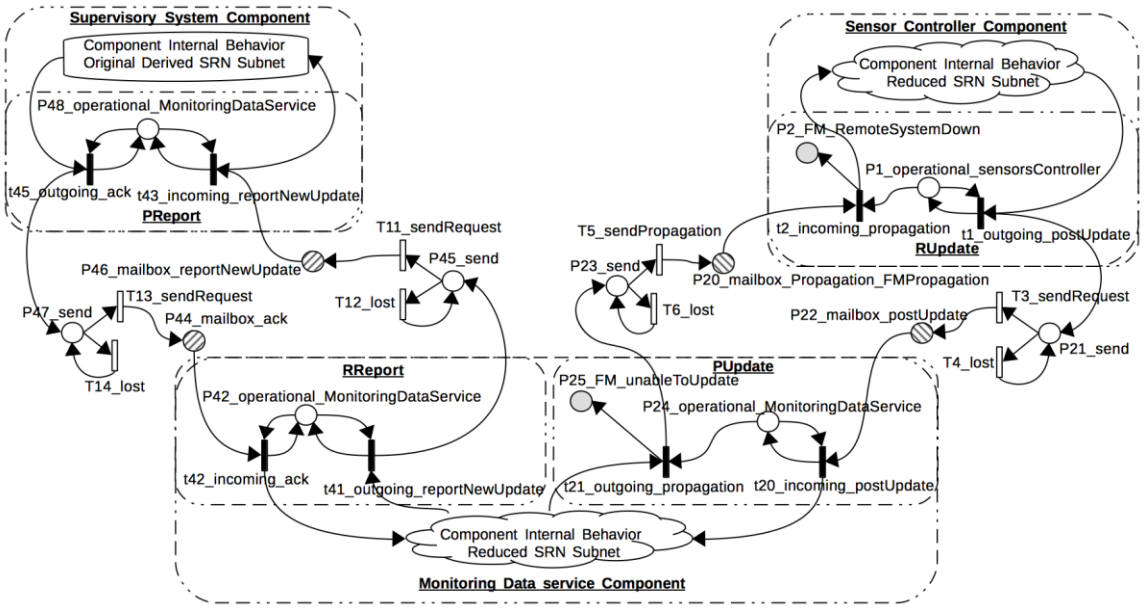


Figure 9.9: First Auxiliary Model of FMS Case Study

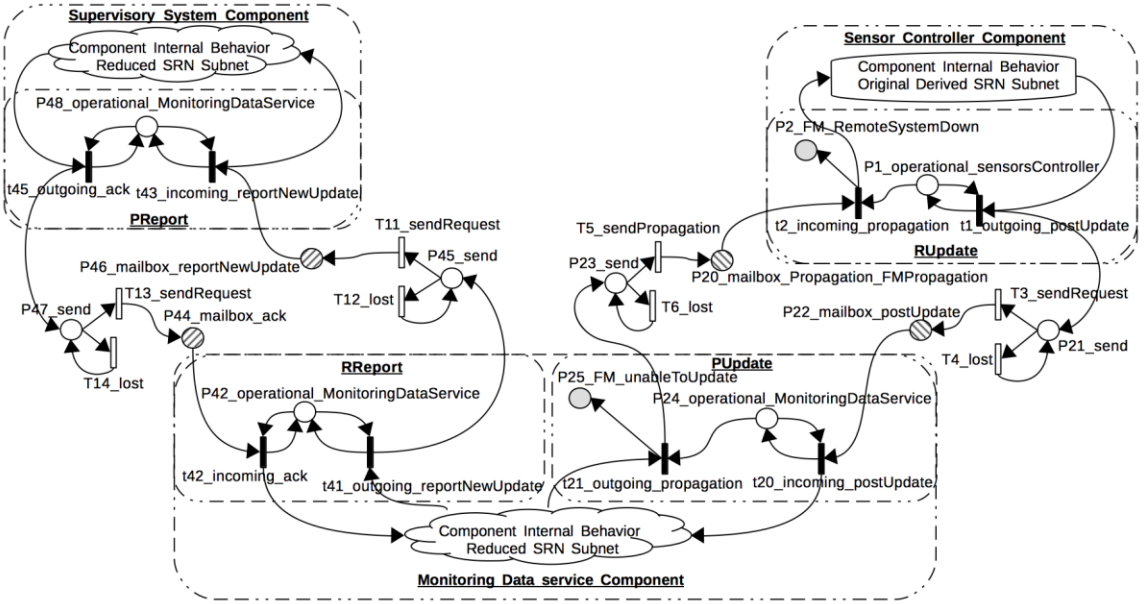


Figure 9.10: Second Auxiliary Model of FMS Case Study

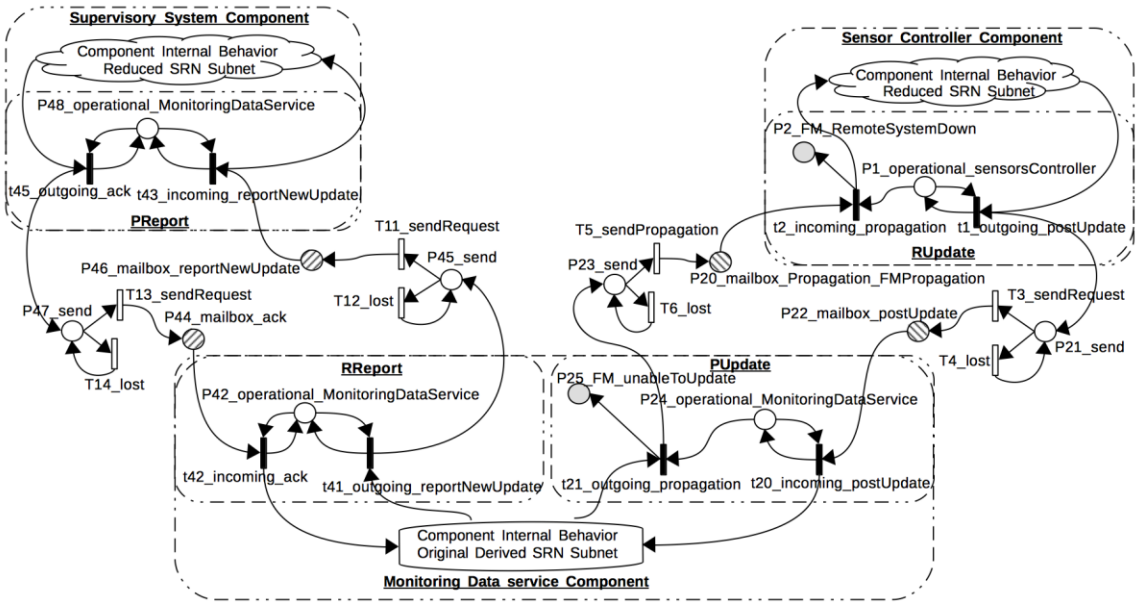


Figure 9.11: Third Auxiliary Model of FMS Case Study

9.4.2 Iterative Subnet Delay Equivalence

Our aim is to avoid solving a large original model by solving smaller auxiliary models instead. We first compute the rates of the aggregated transitions and then construct an approximate model that has a smaller state space compared to the original derived SRN model. As illustrated in Figure 9.9, Figure 9.10, and Figure 9.11 each auxiliary model contains one complete SRN subnet of one software component and aggregated subnets of other dependent components. These auxiliary models are solved iteratively based on the delay equivalence between the aggregated subnet and its corresponding original (not reduced) subnet. In other words, for each reduced subnet the rate of the aggregated transition is derived from its original subnet in another auxiliary model. Aggregated rates are tuned and adjusted iteratively. In each iteration, we solve all auxiliary models in parallel using a SPNP tool to calculate new rate values of the aggregated transitions to be then used in the next cycle. The stopping condition in our

approach depends on the percentage difference of calculated rate between two successive iterations. A similar approach is presented in [14], [15].

As mentioned earlier, each subnet represents a service execution path in the component internal behavior. We assume that S-invariant covers all subnets; therefore, we can apply Little's law in calculating the service subnet delay (execution time) as well as the rate tuning process of aggregated transitions. To make the aggregated subnet delay equivalent to its original subnet, we consider the delay of processing a request once it has reached the subnet input places and until it leaves through an output transition. Little's law is applied to obtain the subnet delay as follows:

$$\text{Service Time} = \frac{\text{Utilization}}{\text{Throughput}}$$

Utilization is mapped to the number of traversable tokens (requests) that arrive at the subnet and that leave the subnet after finishing the processing. In our case, the mean subnet delay of the original service execution path can be found by applying Little's law as follows:

$$\mathrm{D} = \frac{\overline{\mathrm{N}}}{\lambda}$$

where $\overline{N}$ is the mean weighted sum of traversable tokens (requests) in the original subnet and λ is the throughput of the original subnet. In a similar way we can obtain the delay of the corresponding aggregated subnet as follows:

$$\mathrm{d_{agg}} = \frac{\overline{\mathrm{n}}_{\mathrm{agg}}}{\lambda_{\mathrm{agg}}}$$

Where $\bar{n}_{agg}$ is the mean weighted sum of traversable tokens (requests) in the aggregated subnet, λ_{agg} is the throughput of the aggregated subnet. Subnets delay equivalence is

enforced between the aggregated subnet in one auxiliary model and the corresponding original subnet in the other auxiliary model as follow:

$$D = d_{agg}$$

$$\frac{\overline{N}}{\lambda} = \frac{\bar{n}_{agg}}{\lambda_{agg}} \quad \textbf{(9. 1)}$$

On the left hand, the values of $\overline{N}$ and λ of original subnets in the auxiliary models can be determined by a customized reward rate function while solving the model using the SPNP tool. In similar way we obtain the $\bar{n}_{agg}$of the reduced subnet. Throughput of the reduced (aggregated) subnet can be computed by the following:

$$\lambda_{agg} = \mu_{agg} \cdot \mathrm{P}(\#p_h > 0) \quad \textbf{(9. 2)}$$

Where μ_{agg} is the constant rate of the aggregated subnet, $\mathrm{P}(\#p_h > 0)$is the probability of the place been non-empty. To find the constant rate of the aggregated transition (μ_{agg}) we substitute 9.2 in 9.1 as following:

$$\mu_{agg} = \frac{\bar{n}_{agg}}{\mathrm{P}(\#p_h > 0)} \cdot \frac{\lambda}{\overline{N}} \quad \textbf{(9. 3)}$$

To obtain the new rate of the aggregated transition, equation 9.3 will be applied in every iteration. Using the same derivation approach, we apply the delay equivalence in all reduction rules to come up with an equation similar to 9.3 that can be used by the SPNP. Section 9.5 presents the formula derivation for each reduction rules to compute the constant rates of aggregated transitions.

9.4.3 SRN Approximate Model Construction

We are interested in a measure which can determine the system failure rates (unreliability), defined as the probability that the system is in a failure mode at various points in time. Another example is finding the system's availability, which is the

probability over time that the system is in a healthy state and none of the failure modes states is active. Such measures are obtained from the entire system states and therefore, we build an approximate model, which combines all the components reduced subnets.

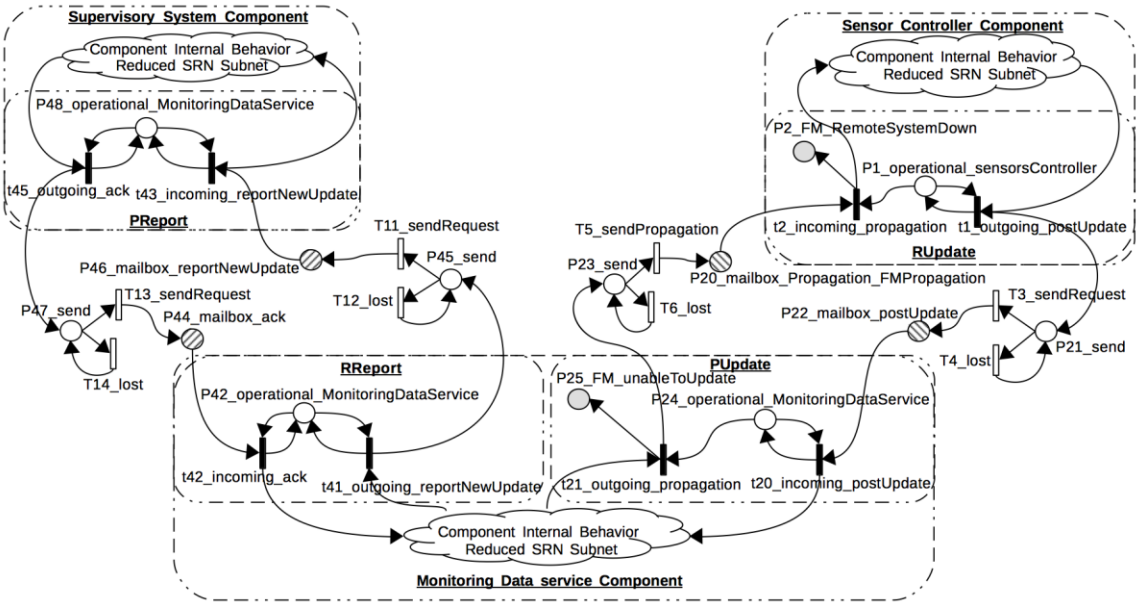


Figure 9.12: Approximate Model without Deployment Nodes of FMS Case Study

In our automated dependability analysis, we model component ports behavior to capture incoming and outgoing messages. This modeling technique helps us in several ways: e.g., to verify components conformance and compatibility and to model failure propagation between components. The ports in our approach encapsulate component internal behavior by reflecting the healthy state and the set of failure modes. During the reduction process, all failure mode places in the internal behavior get aggregated in the reduced net, but whether a component is in normal or failure mode is still reflected in the port model. As a result, the approximate model preserves all failure modes places despite the reduction.

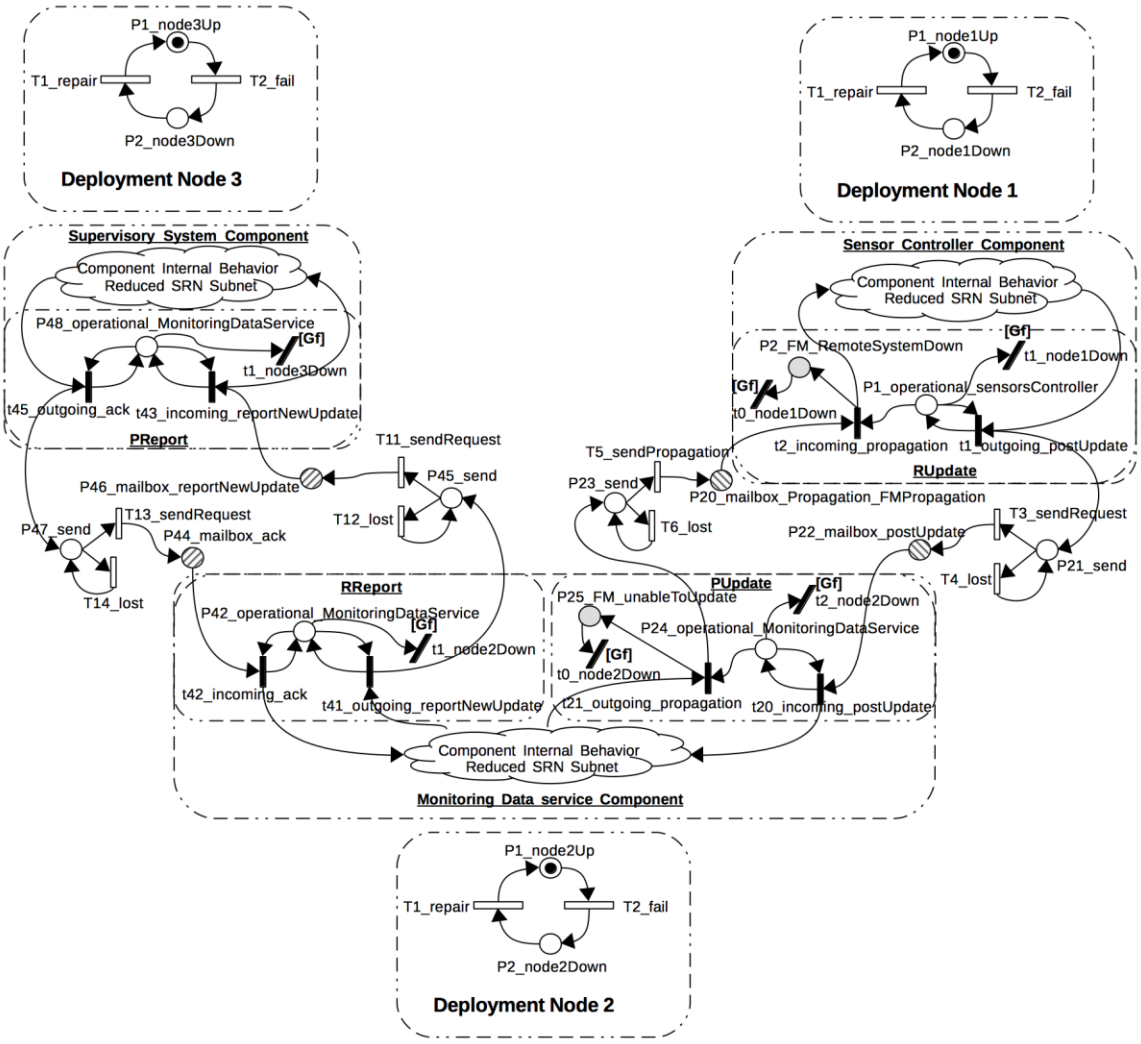


Figure 9.13: Approximate Model with Deployment Nodes of FMS Case Study

The reduction approach is applied to the Platform Independent Model (PIM) that is derived from the software model without considering the deployment. In fact, adding deployment nodes to SRN subnets to get the Platform Dependent Model (PDM) will increase the complexity of the SRN model, and it is not beneficial for the reduction process. The SRN deployment node subnet is abstracted and can not be further reduced. Moreover, our focus for reduction was only on the SRN subnets of the component internal behavior since they can be reduced compared to other subnets that cannot be reduced (i.e., ports, connectors, or deployment node). Therefore, we first construct the

PIM approximate model as shown in Figure 9.12. Then we apply PIM-to-PSM transformation rules to add the deployment nodes SRN subnet to the approximate model as illustrated in Figure 9.13. Note that the system reliability and other measures can also be computed from any auxiliary models since it the entire system states. However, we prefer to build an approximate model to preserve symmetry by having a model with only reduced SRN subnets of all components. To model failure propagation from deployment node to the hosted software component, we apply the transformation rule presented in Chapter 8 based on SRN guards.

9.5 Constant Rate Iterative Formula Derivation and Tuning Process

In this section, we illustrate the derivation of the equation for each reduction rule. Such an equation will be used in solving auxiliary models. Additionally, we explain the tuning process to obtain the constant rate of the aggregated transitions. The following is a list of useful notations:

D : Original subnet delay;

d_{agg} : Reduced subnet delay;

w_i : Weight of place i;

$\bar{N}$: Mean number of weighted traversable token of the original subnet;

$\bar{n}_{agg}$: Mean number of weighted traversable tokens of the reduced subnet;

λ : Original subnet throughput;

λ_{agg} : Reduced subnet throughput.

$\mathrm{P}(\#p_h = i)$: Probability of having i tokens in place p_h

9.5.1 SISO Subnet

The original subnet is regular and can have maximum one token at a time. The total subnet delay can be obtained by applying Little's law as follows:

$$D = \frac{\overline{N}}{\lambda}$$

All places can hold maximum one token. The mean number of weighted traversable tokens is obtained by:

$$\overline{N} = \sum_{i} w_i \cdot \mathrm{P}(\#p_i = 1)$$

Where i is the place number in the original subnet. We set $w_i = 1$ for every place in the traversable path $w_i = 0$ otherwise.

The value of λ is equal to the throughput of any timed transition in the original subnet since we have only one path from input place to the output transition. We compute values of $\overline{N}$ and λ of derived SRN model by using a customized reward rate function implemented by the SPNP tool.

According to the reduction rule of SISO shown in Figure 9.4, the resulted subnet is regular and empty. It has only an input place and one aggregated timed transition that represents the original subnet request processing delay. Throughput of the reduced subnet is given by:

$$\lambda_{agg} = \mu_{agg} \cdot \mathrm{P}(\#\, p_h > 0) \quad \textbf{(9. 4)}$$

The number of traversable tokens can be computed as weighted sum of input place:

$$\bar{n}_{agg} = w_h \sum_{i=1}^{max} i \cdot \mathrm{P}(\#p_h = i)$$

The input place can have maximum one token and we have only one traversable path. Therefore, we set $w_h = 1$. Now the number of traversable tokens is obtained by:

$$\bar{n}_{agg} = \mathrm{P}(\#p_h = 1) \quad \textbf{(9. 5)}$$

The Subnet delay can be calculated by applying Little's law as follow:

$$d_{agg} = \frac{\bar{n}_{agg}}{\lambda_{agg}}$$

By enforcing the delay equivalence between the reduced subnet and the corresponding original subnet we get the following:

$$\frac{\bar{n}_{agg}}{\lambda_{agg}} = \frac{\bar{N}}{\lambda}$$

Substitute equations 9.4 and 9.5 to get

$$\frac{\mathrm{P}(\#p_h = 1)}{\mu_{agg} \cdot \mathrm{P}(\#\, p_h > 0)} = \frac{\bar{N}}{\lambda}$$

Then the constant rate of the aggregated transition is given by:

$$\mu_{agg} = \frac{\mathrm{P}(\#p_h = 1)}{\mathrm{P}(\#\, p_h > 0)} \cdot \frac{\lambda}{\bar{N}}$$

Since input place p_h can have at most one token then $\mathrm{P}(\#p_h = 1) = \mathrm{P}(\#\, p_h > 0)$, therefore, the constant rate of the aggregated transition is given by:

$$\mu_{agg} = \frac{\lambda}{\bar{N}} \quad \textbf{(9. 6)}$$

9.5.2 SM Subnet

The SM subnet has one input place and multiple output transitions. Therefore, in our case we have multiple output paths, i.e., normal and erroneous paths. The throughput of the original subnet via i^{th} output path is given by:

$$\sum_i \lambda_i$$

Total original subnet throughput is obtained by summing throughputs of all output paths.

$$\sum \sum_{i} \lambda_i$$

Total subnet delay of the original subnet is:

$$D = \frac{\bar{N}}{\sum \sum_i \lambda_i}$$

According to CeBAM and the transformation rules presented in Chapter 5, all places in SM subnet can hold at most one token. The mean number of weighted traversable tokens via i^{th} output path is obtained by

$$\bar{N} = \sum_{i} \sum_{j} w_j \cdot \mathrm{P}(\#p_j = 1)$$

where j is the place number in the original subnet. We set $w_j = 1$ for every place in the traversable path, $w_j = 0$ otherwise. We implement a reward rate function in the SPNP tool to compute the subnet throughput and mean number of weighted traversable tokens via i^{th} output path.

According to the SM reduction rule illustrated in section 9.3.2, the number of output paths (arcs) of the reduced subnet is similar to the original subnet. In our case, one path represents the normal execution of the received request while the other path represents an erroneous path for the manifested failure mode. Throughput of the aggregated subnet via i^{th} output transition is give by:

$$\lambda_{agg_i} = \mu_{agg_i} \cdot \mathrm{P}(\#p_h > 0) \quad \textbf{(9. 7)}$$

The aggregated subnet will have only one input place p_h, therefore the weighted traversable tokens is:

$$n_{agg} = \mathrm{P}(\#p_h = 1) \quad \textbf{(9. 8)}$$

We apply the delay equivalence via i^{th} output transition between the reduced subnet and the corresponding original subnet

$$\frac{\bar{n}_{agg}}{\lambda_{agg_i}} = \frac{\bar{N}}{\sum_i \lambda_i} \quad \textbf{(9. 9)}$$

By substituting 9.7 and 9.8 in 9.9 we obtain:

$$\mu_{agg_i} = \frac{\bar{n}_{agg}}{\mathrm{P}(\#p_h > 0)} \cdot \frac{\sum_i \lambda_i}{\bar{N}} \quad \textbf{(9. 10)}$$

9.5.3 And-Or Subnets

In proposed the approach, we have two cases of And-Or subnet. The first case is *and:or* where we have two input places and multiple output transitions. The second case is *and:and* subnet that has two input places and one output transition. In both cases, all places can hold at most one token except one of the input places ($p_{\_request}$) that represents the event pool of service execution path. According to the transformation rules presented in Chapter 5, this place receives all the requests and caches them so that each is processed sequentially.

The first case is *and:or* subnet. For the original subnet the throughput via i^{th} output path is similar to SM subnet and given by:

$$\sum_i \lambda_i$$

Total original subnet delay is given by:

$$D = \frac{\bar{N}}{\sum \sum_i \lambda_i}$$

Where $\sum \sum_i \lambda_i$ is the total throughput of the original *and:or* subnet. The mean number of weighted traversable tokens via i^{th} output path is obtained by:

$$\bar{N} = \sum_i \sum_j w_j \cdot \mathrm{P}(\#p_j = 1) + \sum_n^{max} n \cdot \mathrm{P}(\#p_{\_request} = n)$$

where j is the place number in the original subnet and n is the number of tokens in event pool place ($p_{\_request}$). We set $w_j = 1$ for every place in the traversable path $w_j = 0$ otherwise. The original subnet throughput and weighted traversable tokens are computed using a reward rate implemented in SPNP tool.

According to the reduction rule in section 9.3.3, the number of output transitions in the original subnet and reduced subnet are equal. Subnet throughput of the reduced subnet via i^{th} output path is computed in a similar way of SM subnet as follows:

$$\lambda_{agg_i} = \mu_{agg_i} \cdot \mathrm{P}(\#p_h > 0) \quad \textbf{(9. 11)}$$

The reduced subnet has two input places connected to immediate transition and then connected to another place, which is the input place output transitions see Figure 9.7. So weighted traversable tokens are

$$n_{agg} = \mathrm{P}(\#p_{\_init} = 1) + \sum_n^{max} \mathrm{P}(\#p_{\_request} = n) + \mathrm{P}(\#p_h = 1) \quad \textbf{(9. 12)}$$

We apply the delay equivalence via i^{th} output transition between the reduced subnet and the corresponding original subnet as follows:

$$\frac{\bar{\mathrm{n}}_{\mathrm{agg}}}{\lambda_{\mathrm{agg_i}}} = \frac{\bar{\mathrm{N}}}{\sum_{\mathrm{i}} \lambda_{\mathrm{i}}}$$

Using 9.11 the constant rate of the i^{th} the output transition is obtained by:

$$\mu_{agg_i} = \frac{\bar{n}_{agg}}{\mathrm{P}(\#p_h > 0)} \cdot \frac{\sum_i \lambda_i}{\bar{N}} \quad \textbf{(9. 13)}$$

The second case is *and:and* subnet. The throughput λ is given by the throughput of the output timed transition because we have only one path from the input places to the output transition. The mean number of weighted traversable tokens is obtained by

$$\bar{N} = \sum_{i} w_i \cdot \mathrm{P}(\#p_i = 1) + \sum_{n}^{max} n \cdot \mathrm{P}(\#p_{\_request} = n)$$

Where i is the place number in the original subnet and n is the number of tokens in event pool place ($p_{\_request}$). We set $w_i = 1$ for every place in the traversable path $w_i = 0$ otherwise.

In this subnet, we have only one output transition, then subnet throughput is equal to the throughput of output transition. We obtain values of $\bar{N}$ and λ by using a customized reward rate function implemented by the SPNP tool.

In the reduced subnet (see Figure 9.6) we have only one output transition. Therefore, the subnet throughput is equal to the throughput of the output transition

$$\lambda_{agg} = \mu_{agg} \cdot \mathrm{P}(\#p_h > 0) \quad \textbf{(9. 14)}$$

Similar to *and:or* subnet, the mean weighted traversable tokens is

$$n_{agg} = \mathrm{P}(\#p_{\_init} = 1) + \sum_{n}^{max} \mathrm{P}(\#p_{\_request} = n) + \mathrm{P}(\#p_h = 1) \quad \textbf{(9. 15)}$$

If we apply delay equivalence between the reduced and original subnets then:

$$\frac{\bar{n}_{agg}}{\lambda_{agg_i}} = \frac{\bar{N}}{\lambda}$$

Using 9.14 the rate of the aggregated transition is obtained by:

$$\mu_{agg} = \frac{\bar{n}_{agg}}{\mathrm{P}(\#p_h > 0)} \cdot \frac{\lambda}{\bar{N}} \quad \textbf{(9. 16)}$$

9.5.4 Aggregated Transitions Rates Tuning Process

The objective is to obtain the constant rate of the aggregated transitions that is equivalent to the corresponding original subnet delay. To achieve that, auxiliary models are solved iteratively. In each cycle, we solve all constructed auxiliary models in parallel. The initial rate of the aggregated transitions is set to the average of rates in the execution path. The tuning process is performed using the derived delay equivalence equations of each reduced subnet type. The iteration is stopped if the difference between two successive rates is falling into a predefined threshold.

```
BEGIN ALGORITHM (Tuning process of aggregated transitions constant rates)

        FOREACH (component)
            Apply SRN subnet identification and reduction algorithm
        END FOREACH

        Build Auxiliary models

        FOREACH (auxiliary model)
            Set the initial rates of aggregated transitions
        END FOREACH

        DO UNTIL (stopping condition is satisfied)
            Simultaneously solve all auxiliary models

            FOREACH (auxiliary model results)
                    Collect throughputs and mean number of tokens in each subnet
                    Calculate new rates using delay equivalence equations
                    Update aggregated transition rates
            END FOREACH

        END DO

END ALGORITHM (Tuning process of aggregated transitions constant rates)
```

The successive substitution method used in the tuning process is similar to [15]. In each iteration, we collect some measures such as the probability that certain places are nonempty and transitions throughputs. These values will be then used in the derived

delay equivalence equations to compute the newly tuned rate values for the aggregated transitions. In the next iteration, we substitute the obtained rates from the previous cycle and repeat the tuning process until the stopping criteria is satisfied. This process is described in the in algorithmic syntax. In the SPNP tool, we use transient analysis method and accumulative measures settings. To reduce the calculation complexity of the derived equations in the previous section, we implement it using CSPL as functions used by the SPNP solver package while solving constructed auxiliary models.

9.6 Applying Reduction Rules and Formulas Derivation on Illustrative Case Study

In this section, we show the reduction rules application on each component derived SRN model from FMS case study. Additionally, we show in detail the derivation of constant rate formulas used by SPNP solver during the tuning process.

Figure 9.14(a), shows the derived SRN model of sensors controller component internal behavior by following the transformation rules presented in Chapter 5. After applying the tuning process of aggregated transitions constant rates algorithm of subnet identification, we found only one *And:And* subnet (*subnet#1*). Figure 9.14(b) shows the reduced subnet. Note that after applying reduction rules, the original behavior and other subnet properties do not change. For the original subnet, we use the subnet from the second auxiliary model as shown in section 9.4. The original subnet initially is empty and it processes one request at a time.

Delay of original subnet is obtained by:

$$D = \frac{\overline{N}}{\lambda}$$

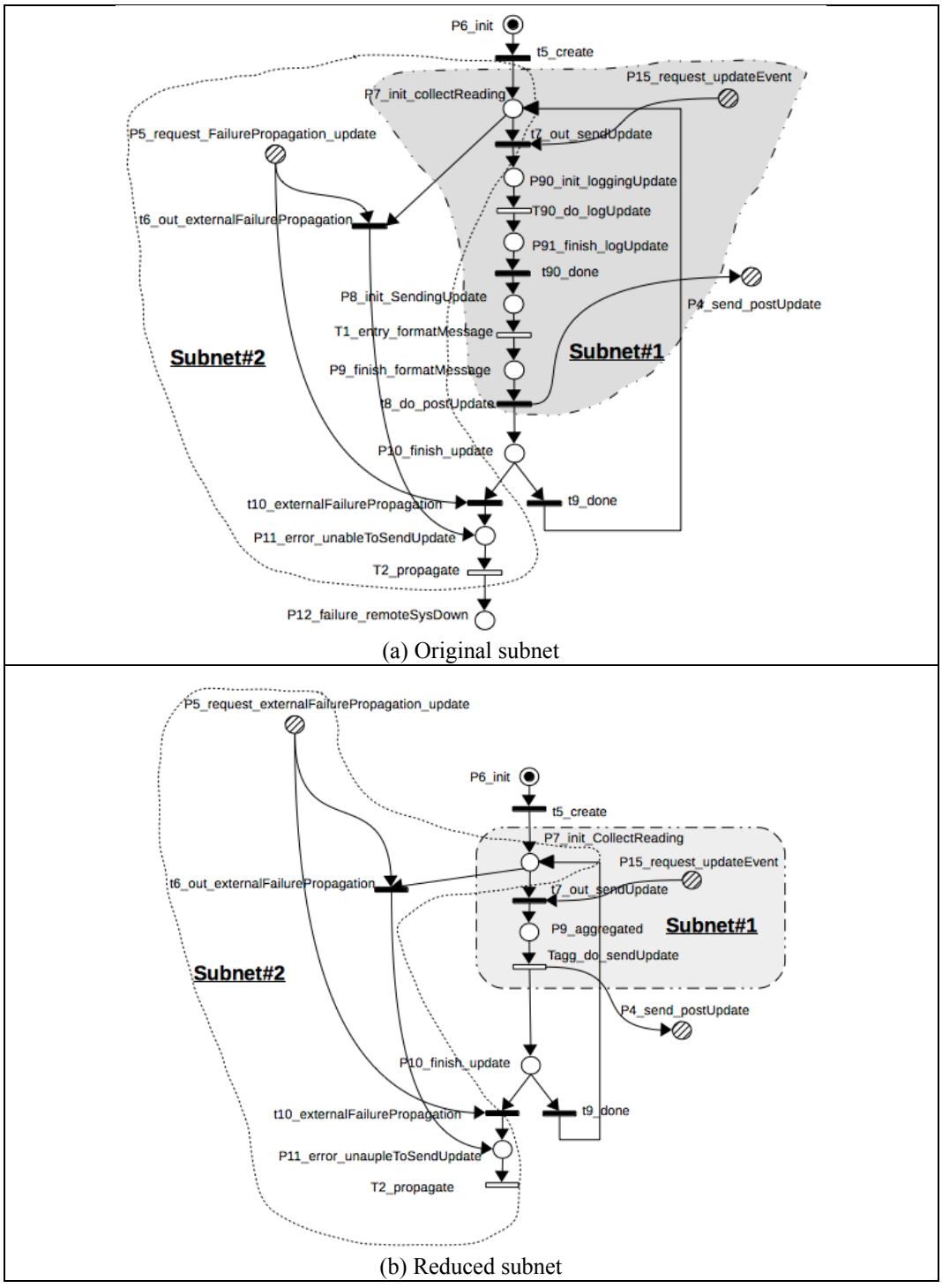


Figure 9.14: Internal Behavior SRN Subnet of Sensor Controller Component

In the original subnet, we have only one traversable path from the entry places to the output transition as shown in Figure 9.14(a). In this path, the places can have at most

one token except $p15_{\_request\_updateEvent}$. This place is representing the event pool of the execution path and it can hold n tokens. Additionally, this subnet will sent out two tokens since the output transition $t8_{\_do\_postUpdate}$ has two output arcs. Thus, the mean number of weighted traversable tokens in this subnet is:

$$\bar{N} = \mathrm{P}(\#p7_{\_init\_collectReading} = 1) + \sum_{n=1}^{max} n \cdot \mathrm{P}(\#p15_{\_request\_updateEvent} = n) + \mathrm{P}(\#p90_{\_init\_loggingUpdate} = 1) + \mathrm{P}(\#p91_{\_finish\_logUpdate} = 1) + \mathrm{P}(\#p8_{\_init\_sendingUpdate} = 1) + 2\,\mathrm{P}(\#p9_{\_finish\_formatMessage} = 1)$$

Since we have only one path, then the throughout of original subnet is the same throughout of the *T1_entry_formateMessage* transition and it is given by:

$$\lambda = \mu_{T1\_entry\_formatMessage} \cdot \mathrm{P}(\#p8_{\_init\_sendingUpdate} > 0)$$

The aggregated subnet shown in Figure 9.14(b) has three places, two of them are the original entry places. It has only one aggregated transition that represents the original subnet delay. Moreover, the output arcs are matching the original subnet to preserve the original behavior. The Subnet delay can be computed from the first auxiliary model as follows:

$$d_{agg} = \frac{\bar{n}_{agg}}{\lambda_{agg}}$$

The Mean number of weighted traversable tokens is obtained by:

$$\bar{n}_{agg} = \sum_{n=1}^{max} n \cdot \mathrm{P}(\#p15_{\_request\_updateEvent} = n) + \mathrm{P}(\#p7_{\_init\_collectReading} = 1) + 2\,\mathrm{P}(\#p9_{\_aggregated} = 1)$$

The Subnet throughput is given by the throughput of *Tagg_do_sendUpdate* as follows:

$$\lambda_{agg} = \mu_{Tagg\_do\_sendUpdate} \cdot \mathrm{P}(\#p9_{aggregated} > 0)$$

In order to obtain the constant rate of $T_{agg\_do\_sendUpdate}$ in the reduced subnet we apply a delay equivalence between the original *and:and* subnet in the second auxiliary model and its reduced subnet in the first auxiliary model as follows:

$$d_{agg} = D$$

$$\frac{\bar{n}_{agg}}{\mu_{Tagg\_do\_sendUpdate} \cdot \mathrm{P}(\#p9_{aggregated} > 0)}$$

$$= \frac{\bar{N}}{\mu_{T1\_entry\_formatMessage} \cdot \mathrm{P}(\#p8_{\_init\_sendingUpdate} > 0)}$$

Then,

$$\mu_{Tagg\_do\_sendUpdate} = \frac{\mu_{T1\_entry\_formatMessage} \cdot \mathrm{P}(\#p8_{\_init\_sendingUpdate} > 0)}{\mathrm{P}(\#p9_{aggregated} > 0)} \cdot \frac{\bar{n}_{agg}}{\bar{N}}$$

All these equations are implemented in the SPNP tool and they will be used in every iteration to tune the value of the constant rate of *Tagg_do_sendUpdate* in the reduced subnet.

Figure 9.15(a) shows the derived SRN model of monitoring the data service of the component internal behavior. After applying the subnet identification algorithm, we found that it has *and:or* subnet (*subnet#1*). We use the original subnet in the third auxiliary model shown in section 9.4 to compute the delay using the following formula:

$$D = \frac{\bar{N}}{\sum_i \lambda_i}$$

The place $p41_{\_request\_postUpdate}$ is modeling the event pool and it can hold n tokens. Requests are processed sequentially since the service execution path will process one request at a time. The request under processing has two possible paths: normal or

erroneous. Each path has its own mean number of weighted traversable tokens, throughput and delay.

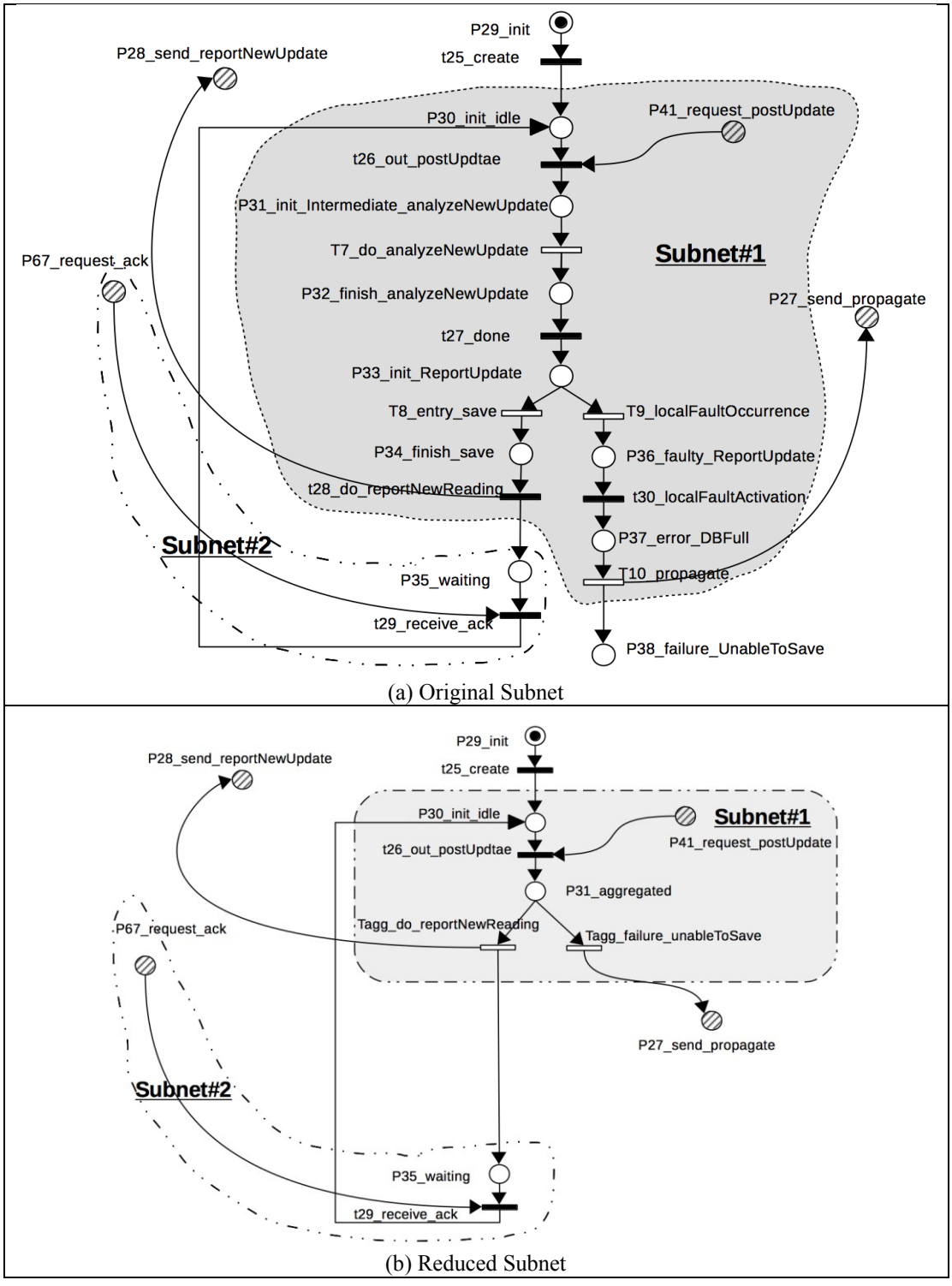


Figure 9.15: Internal Behavior SRN Subnet of Monitoring Data Service Component

For the normal path, the weighted traversable tokens and throughput are computed using the following calculation:

$$\lambda_{normal} = \mu_{T8\_entry\_save} \cdot \mathrm{P}(\#p33_{\_init\_reportUpdate} > 0)$$

$$\bar{N}_{normal} = \sum_{n=1}^{max} n \cdot \mathrm{P}(\#p41_{\_request\_postUpdate} = n) + \mathrm{P}(\#p30_{\_init\_idle} = 1)$$
$$+ \mathrm{P}(\#p31_{\_init\_intermediate\_analyzeNewUpdate} = 1)$$
$$+ \mathrm{P}(\#p32_{\_finish\_analyzeNewUpdate} = 1)$$
$$+ \mathrm{P}(\#p33_{\_init\_reportUpdate} = 1) + 2 \cdot \mathrm{P}(\#p34_{\_finish\_save} = 1)$$

For the erroneous path the weighted traversable tokens and throughput are computed using the following

$$\lambda_{\mathrm{erroneous}} = \mu_{T10\_propagate} \cdot \mathrm{P}(\#p37_{\_error\_DBFull} > 0)$$

$$\bar{N}_{erroneous} = \sum_{n=1}^{max} n \cdot P(\#p41_{\_request\_postUpdate} = n) + P(\#p30_{\_init\_idle} = 1)$$
$$+ P(\#p31_{\_init\_intermediate\_analyzeNewUpdate} = 1)$$
$$+ P(\#p32_{\_finish\_analyzeNewUpdate} = 1)$$
$$+ P(\#p33_{\_init\_reportUpdate} = 1) + P(\#p36_{\_faulty\_reportUpdate} = 1)$$
$$+ P(\#p37_{\_error\_DBFull} = 1)$$

According to the *and:or* subnet reduction rule, we will have two paths similar to the original subnet. Output transitions will be preserved as shown in Figure 9.15(b). The delay of the reduced subnet calculated from the first auxiliary model is presented in Section 9.4.

$$d_{agg} = \frac{\bar{n}_{agg}}{\sum_i \lambda_{aggi}}$$

For normal path, the mean number of weighted traversable tokens and throughput is obtained by:

$$\bar{n}_{agg\_normal} = \sum_{n=1}^{max} n \cdot p41_{\_request\_postUpdate} + \mathrm{P}\left(\#p30_{\_init\_idle} = 1\right) + 2\,\mathrm{P}(\#p31_{aggregated} = 1)$$

$$\lambda_{agg\_normal} = \mu_{Tagg\_do\_reportNewReading} \cdot \mathrm{P}(\#p31_{\_aggregated} > 0)$$

For the erroneous path the mean number of weighted traversable tokens and throughput is obtained by:

$$\bar{n}_{agg\_\mathrm{erroneous}} = \sum_{n=1}^{max} n \cdot p41_{\_request\_postUpdate} + \mathrm{P}\left(\#p30_{\_init\_idle} = 1\right) + \mathrm{P}(\#p31_{\_aggregated} = 1)$$

$$\lambda_{agg\_\mathrm{erroneous}} = \mu_{Tagg\_failure\_unableToSave} \cdot \mathrm{P}(\#p31_{\_aggregated} > 0)$$

Now we apply the delay equivalence between both subnets for each path in order to obtain the constant rate of the aggregated transitions.

For the normal path

$$\frac{\bar{n}_{agg\_normal}}{\lambda_{agg\_normal}} = \frac{\bar{N}_{normal}}{\lambda_{normal}}$$

After we substitute equations we get the following:

$$\mu_{Tagg\_do\_reportNewReading} = \frac{\mu_{T8\_entry\_save} \cdot \mathrm{P}(\#p33_{\_init\_reportUpdate} > 0)}{\mathrm{P}(\#p31_{\_aggregated} > 0)} \cdot \frac{\bar{n}_{agg\_normal}}{\bar{N}_{normal}}$$

Similarly in the erroneous path

$$\frac{\bar{n}_{agg\_\mathrm{erroneous}}}{\lambda_{agg\_\mathrm{erroneous}}} = \frac{\bar{N}_{\mathrm{erroneous}}}{\lambda_{\mathrm{erroneous}}}$$

After substituting equations we get the following:

$$\mu_{Tagg\_failure\_unableToSave}$$

$$= \frac{\mu_{T10\_propagate} \cdot \mathrm{P}(\#p37_{\_error\_DBFull} > 0)}{\mathrm{P}(\#p31_{\_aggregated} > 0)} \cdot \frac{\bar{n}_{agg\_\mathrm{erroneous}}}{\bar{N}_{\mathrm{erroneous}}}$$

These equations are implemented using the CSPL to be used by the SPNP solver.

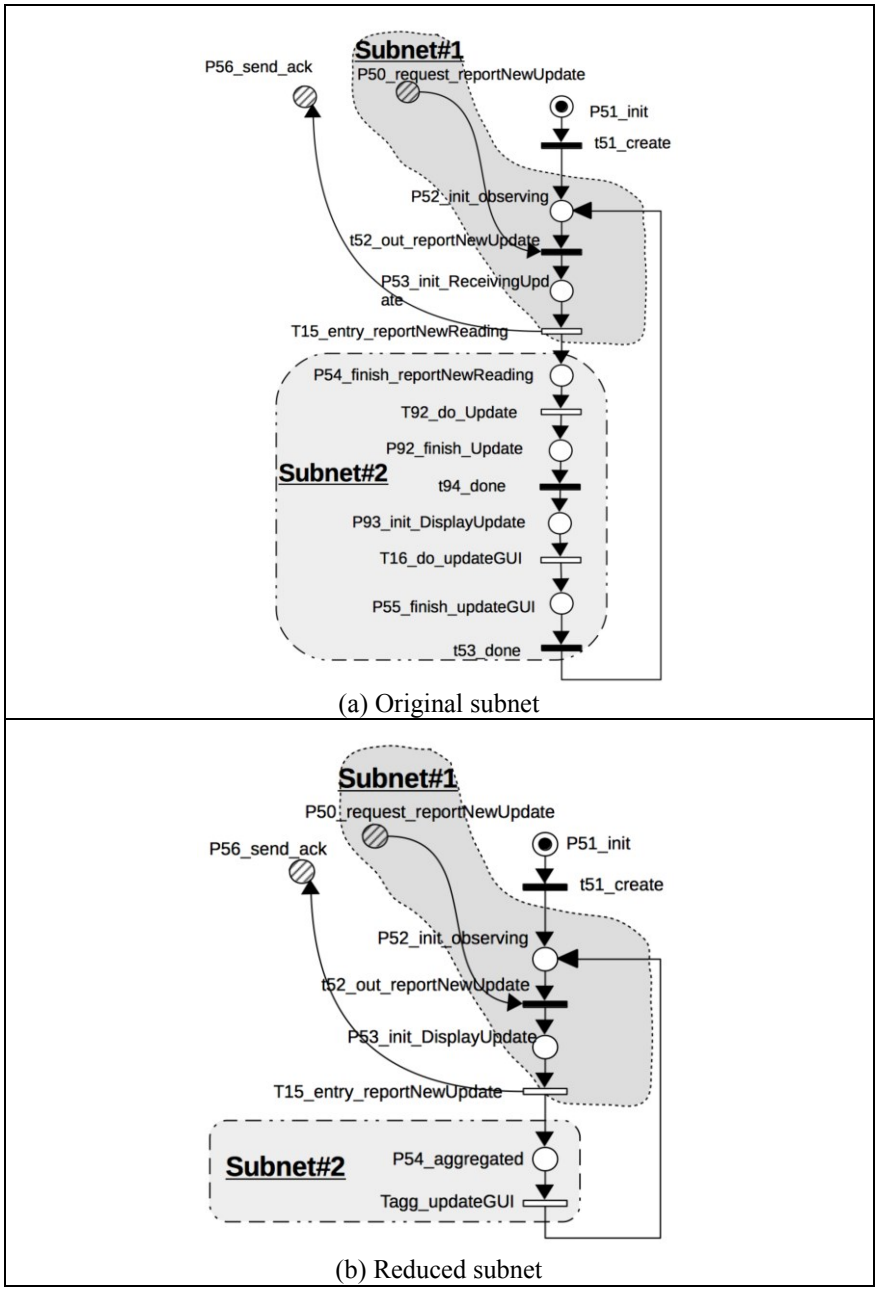


Figure 9.16: Internal Behavior SRN Subnet of Supervisory System Component

Figure 9.16(a) has the derived SRN model of a supervisory system component and its reduced subnet. After applying the subnet identification algorithm, we found two subnets. The subnet one cannot further be reduced, but the second subnet can be reduced using the SISO reduction rule.

The original subnet delay, mean number of weighted traversable tokens and throughput are computed from the first auxiliary model as follows:

$$D = \frac{\bar{N}}{\lambda}$$

$$\bar{N} = \mathrm{P}\left(\#p54_{\_finish\_reportNewReading} = 1\right) + \mathrm{P}\left(\#p92_{\_finish\_update} = 1\right) + \mathrm{P}(\#p93_{\_init\_displayUpdate} = 1) + \mathrm{P}(\#p55_{\_finish\_updateGUI} = 1)$$

$$\lambda = \mu_{T16\_do\_updateGUI} \cdot \mathrm{P}(\#p93_{\_init\_displayUpdate} > 0)$$

The reduced subnet delay, mean number of weighted traversable tokens and throughput are computed from the second auxiliary model as follows:

$$d_{agg} = \frac{\bar{n}_{agg}}{\lambda_{agg}}$$

$$\bar{n}_{agg} = \mathrm{P}(\#p54_{aggregated} = 1)$$

$$\lambda_{agg} = \mu_{Tagg\_updateGUI} \cdot \mathrm{P}(\#p54_{aggregated} > 1)$$

As explained earlier, to obtain the constant rate of the aggregated transition we will match the delay of reduced subnet in the second auxiliary model with its corresponding original subnet in the first auxiliary model

$$d_{agg} = D$$

$$\frac{\mathrm{P}(\#p54_{aggregated} = 1)}{\mu_{Tagg\_updateGUI} \cdot \mathrm{P}(\#p54_{aggregated} > 1)} = \frac{\bar{N}}{\lambda}$$

Since each place in the subnet has at max only one token, then

$$\mathrm{P}\left(\#p54_{aggregated} > 1\right) = \mathrm{P}(\#p54_{aggregated} = 1),$$

which is the probability of place being non-empty

Now, by substituting equations we got the equation to obtain the rate of the aggregate transition $\mu_{Tagg\_updateGUI}$ as follows:

$$\mu_{Tagg\_updateGUI} = \frac{\mu_{T16\_do\_updateGUI} \cdot \mathrm{P}(\#p93_{\_init\_displayUpdate} > 0)}{\overline{N}}$$

9.7 Experimental Results

Table 9.1 show the values for timed transitions of the original subnets and the deployment nodes respectively and Table 9.2 show the list of guard expressions to model failure propagation from deployment node to the hosted software component.

Table 9.1: Original Model Parameterization

Transition name	Assumed Rate
Sensor Controller	
T90_do_logUpdat	1/(3 sec.)
T1_entry_formatMessag	1/(4 sec.)
T2_propagate	1/(2 sec.)
Monitoring Data Service Component	
T7_do_analyzeNewUpdat	1/(10 sec.)
T8_entry_save	1/(3 sec.)
T9_localFaultOccurrenc	1/(9000 sec.)
T10_propagat	1/(2 sec.)
Supervisory System Component	
T15_entry_reportNewUpdat	1/(5 sec.)
T92_do_Update	1/(4 sec.)
T16_do_updateGUI	1/(1.5 sec.)
Hardware Nodes	
T17_node1Fail	1/(604800 sec.)
T18_node1Repair	1/(300 sec.)
T19_node2Fail	1/(259200 sec.)
T20_node2Repai	1/(120 sec.)
T21_node3Fail	(1/720000sec.)
T22_node3Repair	(1/180 sec.)

The Sensor Controller component is connected to the monitoring data service component via a WAN connection. For this connection, we assume the time to send the request is 2 seconds. The transient failure frequency of the WAN connection is assumed to be 550 seconds. For the LAN connection between the monitoring data service component and the supervisory system the time to send the request is 0.1 second and its transient failure is manifested every 5000 seconds. We use these values to set the rates of *T_send* and *T_lost* timed transitions.

Table 9.2: Deployment Nodes Failure Propagation SRN Guards Expressions

Guard Name	Function
Deployment Node 1	
isNode1Up	if (#(P60_node1_up) == 1) 1 else 0
isNode1Down	if (#(P61_node1_down) == 1) 1 else 0
Deployment Node 2	
isNode2Up	if (#(P62_node2_up) == 1) 1 else 0
isNode2Down	if (#(P63_node2_down) == 1) 1 else 0
Deployment Node 3	
isNode3Up	if (#(P64_node3_up) == 1) 1 else 0
isNode3Down	if (#(P65_node3_down) == 1) 1 else 0

Table 9.3: State Space of Approximate Models with and without Deployment.

Number of requests		Model without Deployment		Model with Deployment	
		Original Model	Approximate Model	Original Model	Approximate Model
1	**Tangible**	151	64	12772	4289
	Vanishing	216	84	54534	17803
2	**Tangible**	1278	445	160002	44710
	Vanishing	1301	380	745965	197155
3	**Tangible**	6090	1843	1299817	304344
	Vanishing	4800	1193	6461083	1434799
4	**Tangible**	20909	5721	not enough memory	1599153
	Vanishing	13418	3007	not enough memory	8016907
5	**Tangible**	58004	14722	not enough memory	not enough memory
	Vanishing	31371	6543	not enough memory	not enough memory

Table 9.3 shows the state space comparison between the original derived model and the approximate models. In the first case, we compare the state space between two models without adding the SRN subnet of the deployment node. As the number of the requests increases, the state space size increases with a larger difference between the approximate model and the original model. In fact, this shows the benefit of applying reduction rules. Adding hardware deployment and capturing the failure propagation to the hosted node increases the model complexity and the state space size. However, the approximate model also shows a great benefit in terms of reducing the state space size after including the deployment nodes.

Table 9.4: Results of the Aggregated Transitions Rate Iterative Tuning Process

Iteration / Aggregated transition	**Component Name**			
	Sensor Controller	Monitoring Data Service		Supervisory System
	$\mu_{Tagg\_do\_sendUpdate}$	$\mu_{Tagg\_do\_reportNewReading}$	$\mu_{Tagg\_failure\_unableToSave}$	$\mu_{Tagg\_updateGUI}$
1	0.319764565	0.241559479	6.29184E-05	0.19947465
2	0.343813679	0.184115041	4.46025E-05	0.199664811
3	0.365286604	0.164094538	3.77451E-05	0.199557826
4	0.384564252	0.157321939	3.53472E-05	0.199438728
5	0.401827724	0.155121083	3.45012E-05	0.199323283
6	0.417236716	0.15446076	3.41894E-05	0.199219806
7	0.430968647	0.154305472	3.40619E-05	0.199130048
8	0.443201379	0.15430826	3.39996E-05	0.199052984
9	0.454100604	0.154355278	3.39621E-05	0.198986866
10	0.463815491	0.154410318	3.39352E-05	0.198929979

Table 9.4 shows the results of the iterative tuning process. In the FMS case study, we stop after the tenth iteration since the difference between the last two iterations is fairly small. We use the tuned rates in the approximate model to compute the approximate unreliability of the system. Table 9.5, Figure 9.17 and Figure 9.18 show the system unreliability comparison with and without deployment nodes. In the first case, the

relative error percentage between exact and approximate models is between 4.34% and 2.46%. In the second case, after adding the deployment node to both models, we found the relative error percentage is between 1.87% and 0.66%. From these results we notice the approximations errors are lower in the case of including hardware nodes. In other words, the approximation errors are lower when the size of the model goes up since the model size is much bigger when we include hardware compared to the model without hardware nodes. An explanation of these results is the fact that for the lower size model with fewer components there is more dependency between components, while for the larger models, the dependency diminishes and so the approximation is getting better.

Table 9.5 : System Unreliability Comparison

Time (hours)	Original Model	Approximate Model	Rel. Error [%]	Original Model	Approximate Model	Rel. Error [%]
	Without Deployment Nodes			With Deployment Nodes		
1	0.039372348	0.041082348	4.34	0.028408387	0.028942435	1.88
2	0.077256947	0.080462351	4.15	0.054832797	0.055755449	1.68
3	0.113647477	0.118225128	4.03	0.080431762	0.081671997	1.54
4	0.148602859	0.154437095	3.93	0.105228274	0.106724222	1.42
5	0.182179693	0.189161938	3.83	0.129244775	0.130943085	1.31
6	0.214432344	0.222460729	3.74	0.152503176	0.154358415	1.22
7	0.245413035	0.254392033	3.66	0.175024862	0.176998948	1.13
8	0.275171929	0.285012009	3.58	0.196830704	0.198892369	1.05
9	0.30375721	0.314374509	3.50	0.217941062	0.22006535	0.97
10	0.331215163	0.342531174	3.42	0.238375804	0.240543587	0.91
11	0.357590244	0.369531522	3.34	0.258154307	0.260351837	0.85
12	0.382925162	0.395423043	3.26	0.277295467	0.279513949	0.80
13	0.407260938	0.420251273	3.19	0.295817715	0.298052905	0.76
14	0.430636974	0.444059877	3.12	0.313739018	0.315990846	0.72
15	0.453091121	0.466890729	3.05	0.33107689	0.3333491	0.69
16	0.474659735	0.488783982	2.98	0.347848405	0.350148222	0.66
17	0.495377742	0.509778144	2.91	0.364070199	0.366408013	0.64
18	0.51527868	0.52991013	2.84	0.379758487	0.382147553	0.63
19	0.534394781	0.549215355	2.77	0.394929064	0.397385226	0.62
20	0.55275699	0.567727767	2.71	0.409597314	0.412138744	0.62
21	0.570395045	0.585479929	2.64	0.423778228	0.426425175	0.62
22	0.5873375	0.602503059	2.58	0.437486399	0.440260963	0.63
23	0.603611789	0.618827097	2.52	0.450736042	0.453661957	0.65
24	0.619244261	0.634480751	2.46	0.463540992	0.466643425	0.67

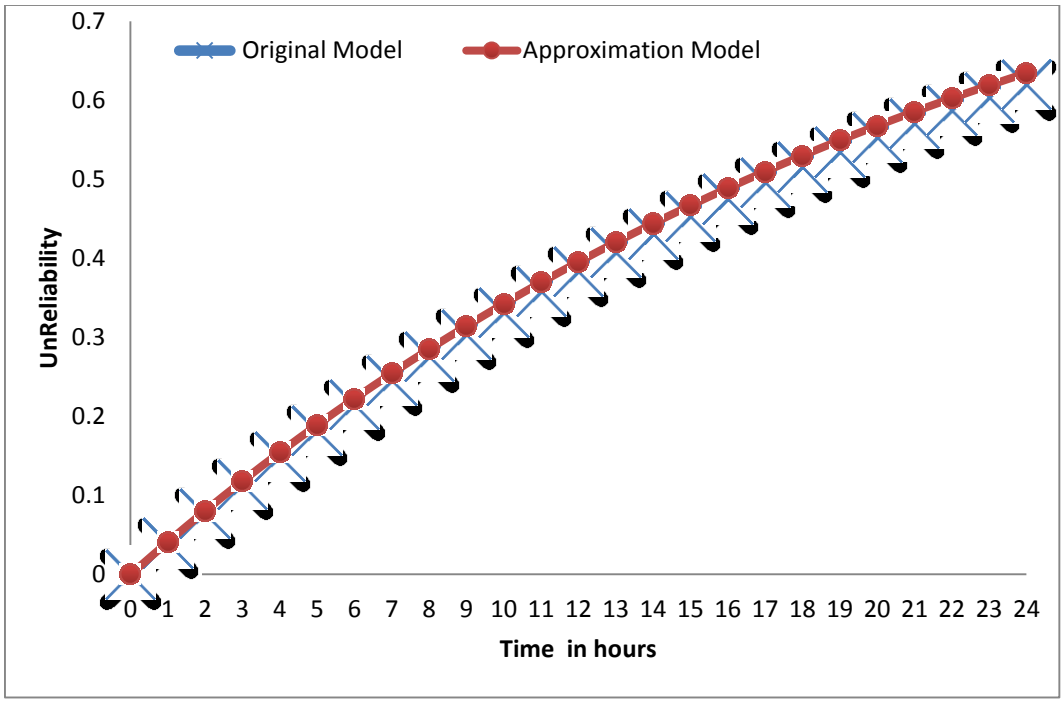


Figure 9.17: Unreliability of Original Model vs. Approximate Model without Deployment Nodes

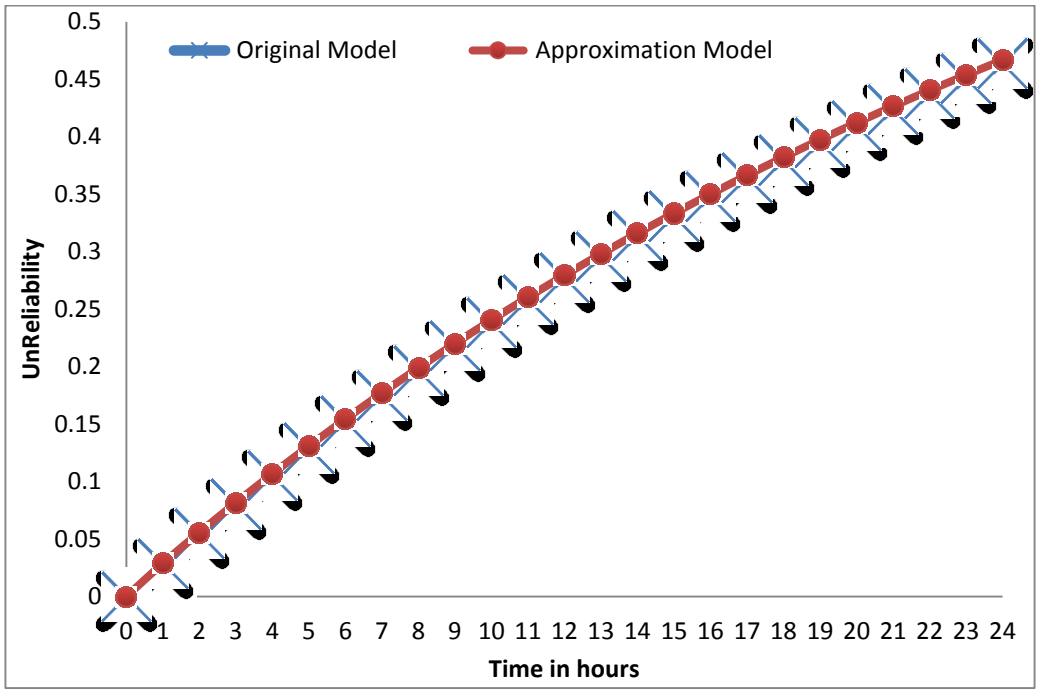


Figure 9.18: Unreliability of the original model vs. Approximate Model with Deployment Nodes

Chapter 10: Conclusion

This chapter summarizes the thesis contributions, discusses the limitations, and provides a list of possible future work.

The thesis presents a framework for integrating model-based dependability assessment techniques in the early stages of software development. This framework is based on UML and profiles such as MARTE and DAM, and it takes into consideration component erroneous behavior and error propagation. The ultimate goal is to help developers and dependability experts to evaluate dependability attributes during the design phase. Moreover, such a framework will provide the modeler with quantitative data to support design decisions such as selecting a fault tolerance technique and comparing different design alternatives. The proposed framework encompasses a set of approaches summarized as follows:

1) CeBAM is an aspect-oriented approach that provides a practical solution for modeling component erroneous behavior and error propagation. In fact, ignoring the erroneous behavior and error propagation in models used for dependability analysis has a negative impact on the dependability assessment accuracy. However, modeling component erroneous behavior (internal and ports) along with normal behavior in one model, increase model complexity and make it hard to maintain. In addition, capturing error propagation between components is not straightforward using existing modeling languages, i.e., UML state machine. The CeBAM approach was developed to address these issues and provide modelers and dependability experts with a practical solution to model component erroneous behavior separately from its normal behavior. It helps to captures the origin of the fault and the chain of dependability

threats for the component internal behavior, as well as for its ports. This approach utilizes AOM and other UML profiles to model the erroneous behavior separately and then to automatically compose erroneous and normal behavior for each component. The outcome of the CeBAM approach is a composed state machine that has normal and erroneous behavior of component internal behavior and its ports. It is transformed to an SRN model for components conformance and compatibility validation as well as reliability analysis.

2) Computing dependability measures, such as reliability and availability of the critical scenario, are achieved by translating component behavioral models that conform to the CeBAM approach to an SRN model. We defined a set of transformation rules to derive an SRN model from component internal and ports behavior. These rules were classified into two groups. The first group is a set of mapping rules that illustrates the semantics of translating the state machine and the extended protocol state machine to the SRN subnet. The second group is compositional patterns that define the composition semantics of the SRN subnets derived. According to the software architecture model, the composition of the derived SRN subnets is performed in two steps. It starts by linking the derived SRN subnet from component internal behavior with the derived SRN subnet from component port protocol behavior. The result of this step is a set of SRN subnets in which each one represents the component derived analysis model. The next step is to compose component SRN subnets to build a scenario SRN model according to the provided and required service. This SRN model is used to perform dependability analysis after transforming it to CSPL code as input for the SPNP solver.

3) In order to study and analyze the reliability or availability, we need to verify the conformance and compatibility of the involved components. This verification helps a modeler to resolve all kind of incompatibility issues in the software model before solving the derived SRN model. We proposed conformance and compatibility verification that support a modeler in fixing modeling errors in the original software model. It supports the modeler to be certain that any manifested internal failure inside a component will be propagated to all connected components. Moreover, it validates that the incoming or outgoing messages in component ports are handled properly by the component internal behavior to avoid a deadlock or absorbing states in the analysis model. In this approach, we used the same formal SRN model for conformance and compatibility verification. In conformance validation, we verify whether the component's internal behavior state machine is conforming to its port protocol state machine or not. This verification is performed during the composition of the component internal SRN subnet with SRN subnets derived from its port protocol behavior. Compatibility verification checks the communication between components in the system. For two connected components, the services provided must be compatible with the requester. Compatibility verification is performed while composing components SRN subnets, according to the software architecture model.
4) A transformation chain was designed and implemented to automate the process of proposed dependability framework. This transformation chain has three model-to-model transformation engines, based on QVT-O, and one model-to-text transformation based on MTL (Acceleo). The first engine is *CebamWeaver* that implements the CeBAM approach. For each component it applies the refactor aspect

and then composes component erroneous behavior with component normal behavior. It verifies the conformance with CeBAM modeling guidelines before weaving the aspect model. It uses QVT-O black-box code to parse the *pointCut* OCLQuery string that is embedded in the aspect model. The output model of *CebamWeaver* is transformed to the SRN model using the second transformation engine (*sm2srm*). This engine has several tasks as follows: i) transforming component internal behavior to SRN subnets, ii) transforming component ports to SRN subnets, iii) composing component SRN subnets derived from component internal and port behavior models, iv) composing components SRN subnets according to the services provided and required, and v) verifying the conformance and compatibility and providing the user with a transformation progress report. A software architectural model is used to guide the transformation engine during the composition process. The QVT-O mapping rules are based on the defined SRN transformation semantics. The SRN model derived by the *sm2srn* transformation captures the SRN subnet structure and marking dependent expressions. In order to solve the derived SRN using the SPNP solver we have to generate the corresponding CSPL code. We fully automated the process of generating CSPL code from the derived SRN. The *srn2cspl* is an intermediate transformation that builds the CSPL model and defines solver environment settings. The *cspl2text* is a model-to-text transformation that generates CSPL code accepted as input by the SPNP solver.

5) SvFTAM is an aspect-based technique for modeling structural and behavioral fault tolerance tactics as reusable models. This approach aims to provide quantitative data for supporting an easy comparison of different fault tolerance tactics in order to select

the best solution for a given system. The following examples of fault tolerance tactics have been modeled using SvFTAM: spare with checkpoint, standby spare, retry, and restart. The SRN derivation rules were extended by introducing new mapping rules for the system with fault tolerance tactics. We solved the derived model to study the effect of applying fault tolerance on a case study and compared different SvFTAM tactics in terms of unreliability.

6) State space explosion is a well-known problem in state-based analysis methods. We proposed a decomposition and reduction technique for the SRN model derived. An identification candidate SRN subnet for reduction was performed systematically. In fact, we utilized composition transformation rules to identify SRN subnets in component internal behavior. The original behavior and subnet properties were preserved after applying PN reduction rules. We adopted the iterative method introduced in [15], [123] to compute the transition rates of the aggregated SRN transitions. To compute system reliability, we constructed an approximate model consisting of reduced subnets. SRN subnets of the deployment nodes were also added to the approximate model with the failure propagation to the hosted component. A comparison of the state space for both models (original and approximate with/without deployment) showed the benefit of the proposed approach. Moreover, the unreliability results of the approximate model were close to the original model with a small relative error percentage.

10.1 Limitations

Although the approach proposed in this thesis has achieved the initial planned goals, some limitations have been identified due to the modeling approach, modeling

tools integration, model transformation support, and selected analysis model. The following list summarizes the limitations found in the proposed approach:

- The proposed CeBAM approach is required to model component internal behavior as well as its port protocol behavior. It needs to build many models since each component should have a single internal behavioral model that represents its normal behavior and a separate erroneous aspect model that captures component fault manifestation and failure modes. A refactor aspect model may be required in some cases. Similarly, for component ports it should have one normal protocol behavior and one erroneous protocol behavior. These models were developed iteratively to capture failure modes and propagations between components involved in the critical scenario. In fact, these models can be developed separately by different teams, i.e., software developers and a dependability expert who is responsible for modeling component erroneous behavior as an aspect model. Despite the number of required models, the proposed approach has many advantages such as capturing complete component behavior with failure propagation in a maintainable way and ease in reading or modifying the models. Another advantage in including port protocol behavior is that it helps in verifying conformance and compatibly of the critical scenario and then deriving the correct SRN model. In addition, it helps in identifying a candidate SRN subnet to be reduced using PN reduction rules without losing the original behavior.
- The proposed approach only focuses on building one analysis model to compute system reliability and availability. However, it could be extended to cover other NFPs, such as performance.

- The proposed approach uses MagicDraw as UML editor and the Eclipse modeling framework for model transformations. However, exporting and importing files between these tools requires extra effort from the user to prepare the exported XMI file. An example is removing customized profiles by the MagicDraw tool which were included in the exported file.
- Eclipse QVT-O implementation was used to develop transformation engines. It is still under development and some operations defined in the OMG standard are still not implemented.
- The proposed automation focuses only on the implementation of core engines. To make this approach user friendly, an interface should be developed to manage all transformation chain processes.
- Traceability is not supported. However, the naming convention of the derived SRN model elements is helpful to manually trace back from the derived model to the software model.

10.2 Future work

The following is a list of future work and directions to extend the proposed approach:

- Extend the transformation rules to derive other analysis models that can be used to study other dependability attributes, such as safety.
- Define new transformation rules to derive other analysis models that can be used to study other NFP, such as performance or security.
- Build a user friendly interface that manages all processes of analysis model derivation by calling the developed transformation engines.

- Integrate existing solvers used for analysis (e.g., SPNP) with the modeling environment.
- Extend *sm2srn* transformation engine to complete the implementation of the SvFTAM approach.
- Automate the decomposition and reduction approach by implementing the proposed algorithms.
- Extend the transformation chain by adding a traceability feature.

Appendices

Appendix A: SRN and CSPL metamodels

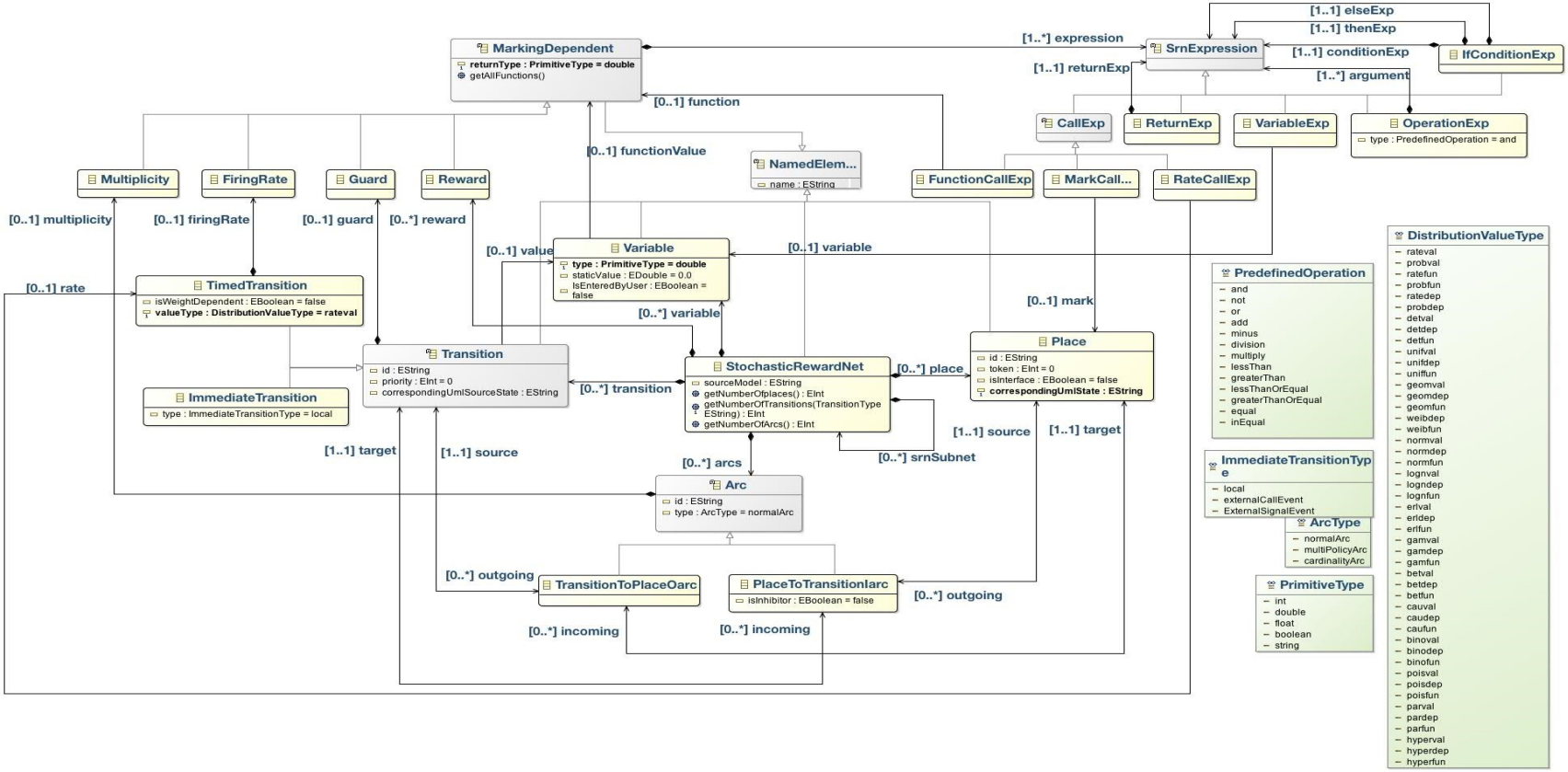


Figure A.1: SRN Metamodel

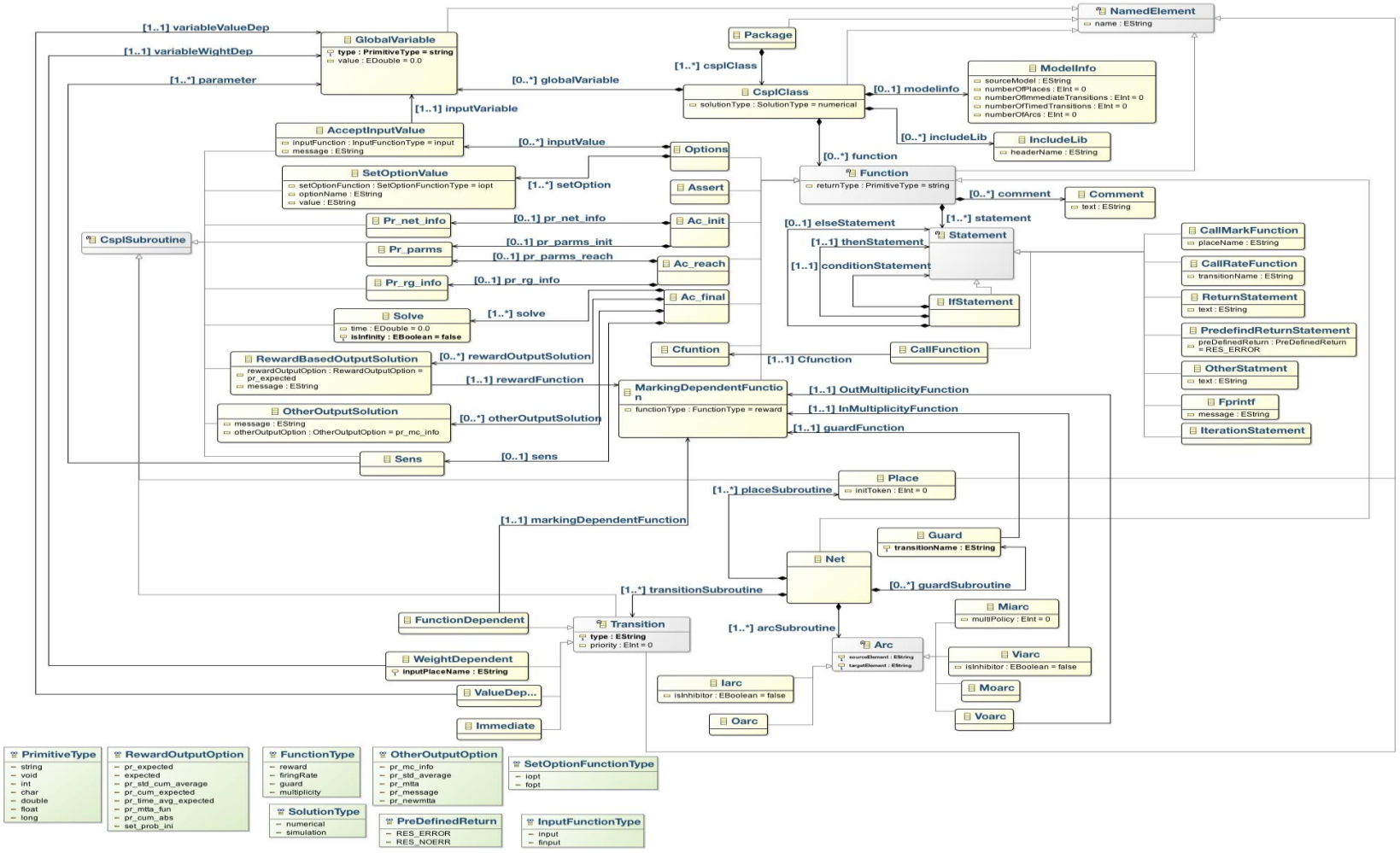


Figure A.2: CSPL Metamodel

Appendix B: Eclipse Implementation of Model Transformation Engines

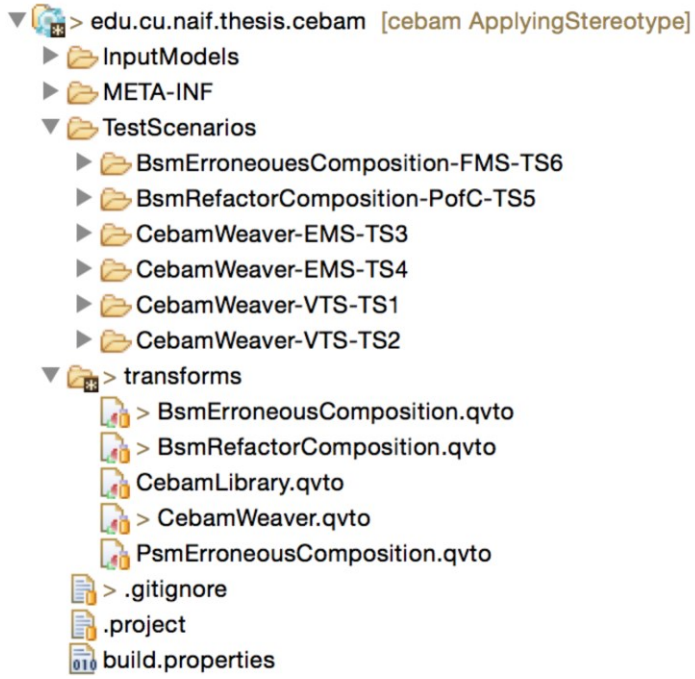


Figure B.1: QVT-O Eclipse Implementation of *CebamWeaver* Transformation Engine

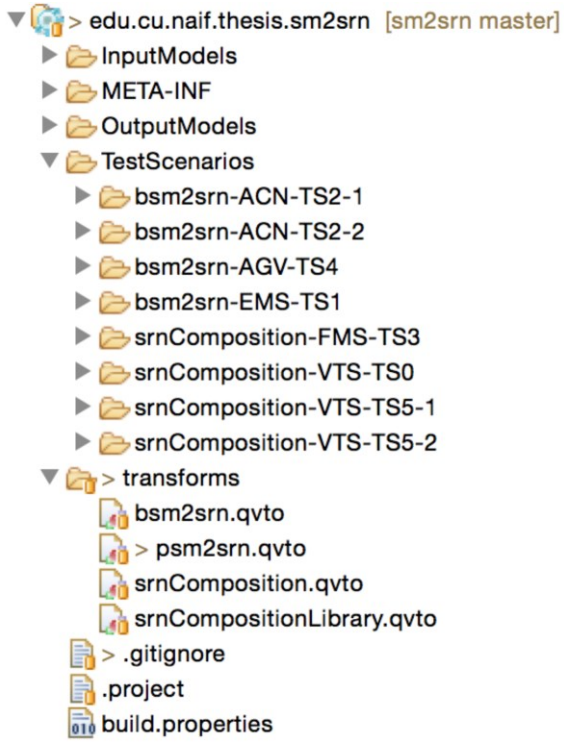


Figure B.2: QVT-O Eclipse Implementation of *sm2srn* Transformation Engine

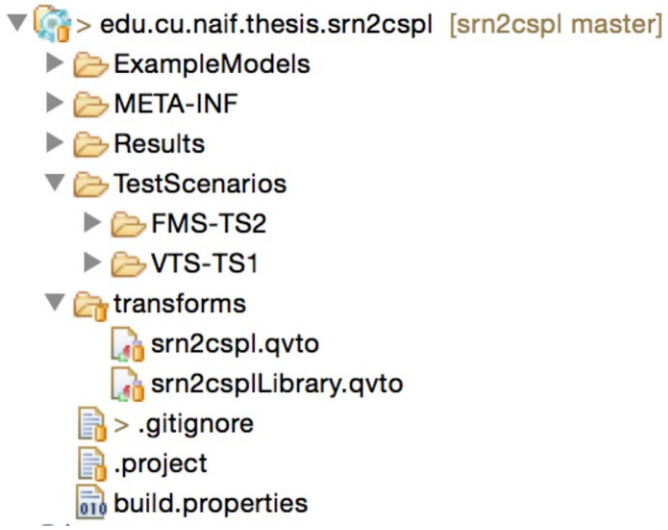


Figure B.3 : QVT-O Eclipse Implementation of *srn2cspl* Transformation Engine

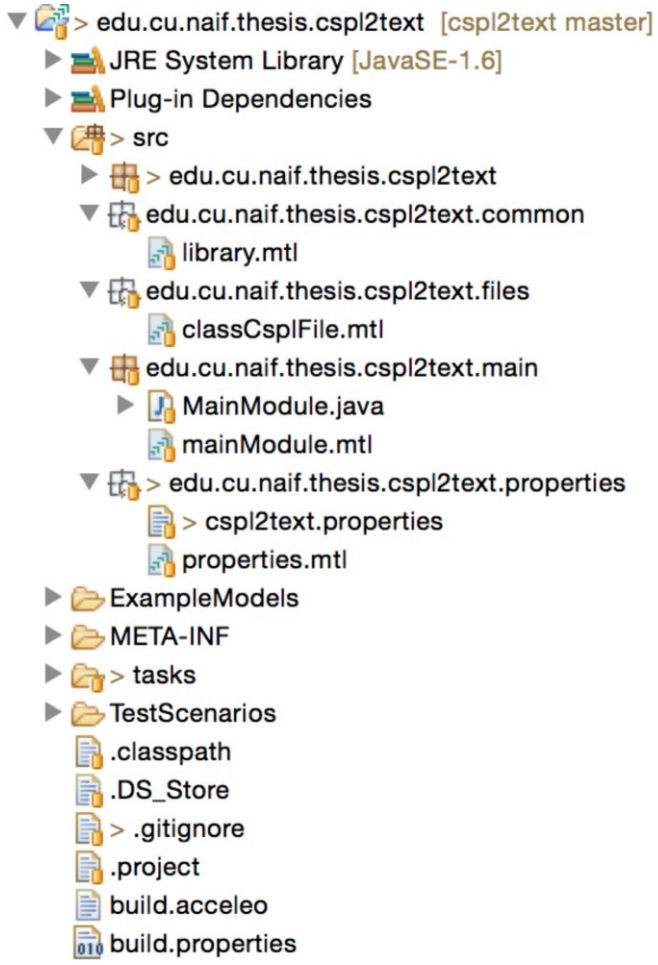
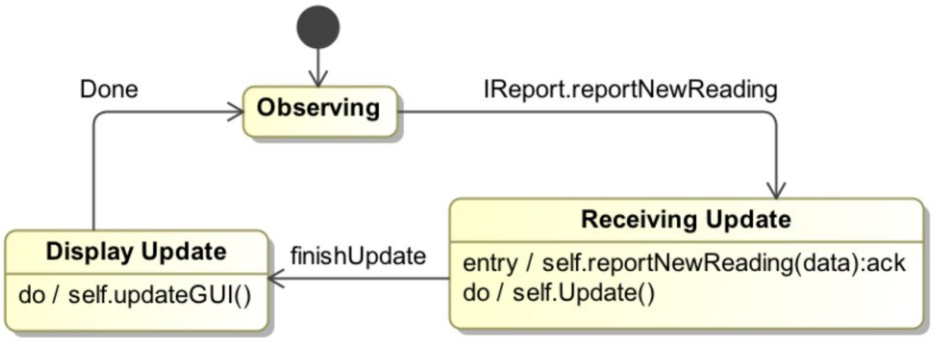
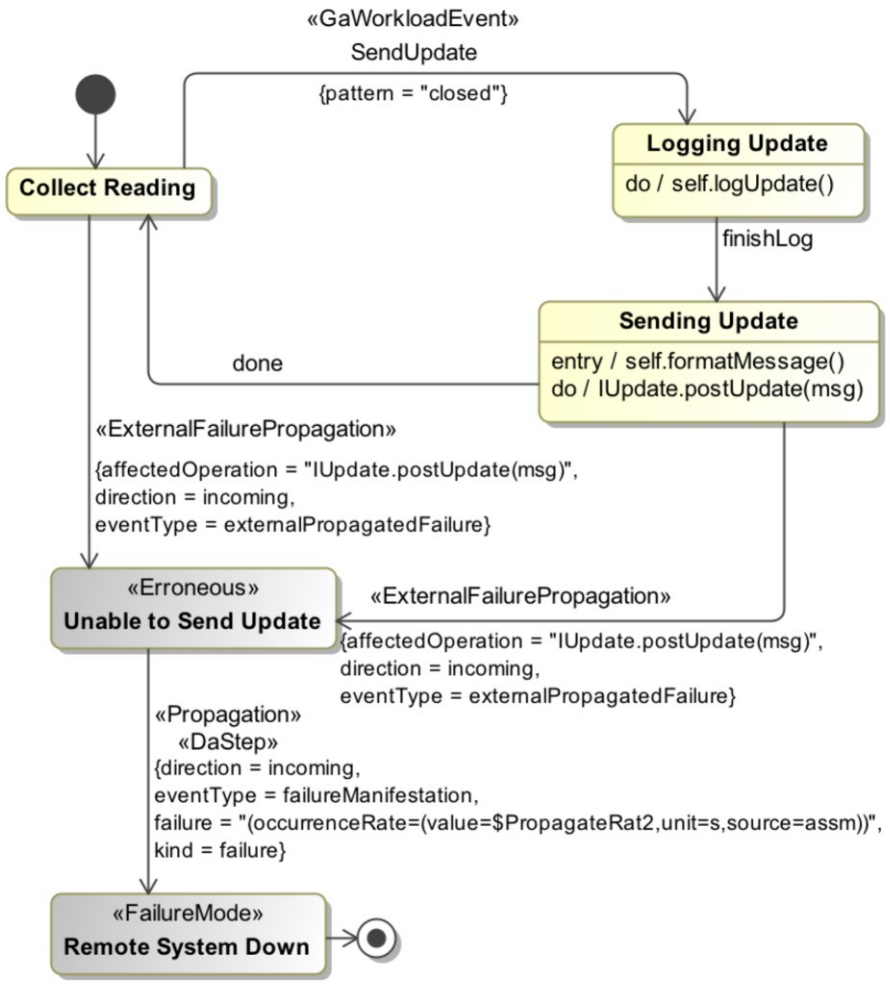


Figure B.4 : Acceleo Eclipse Implementation of *cspl2text* Transformation Engine

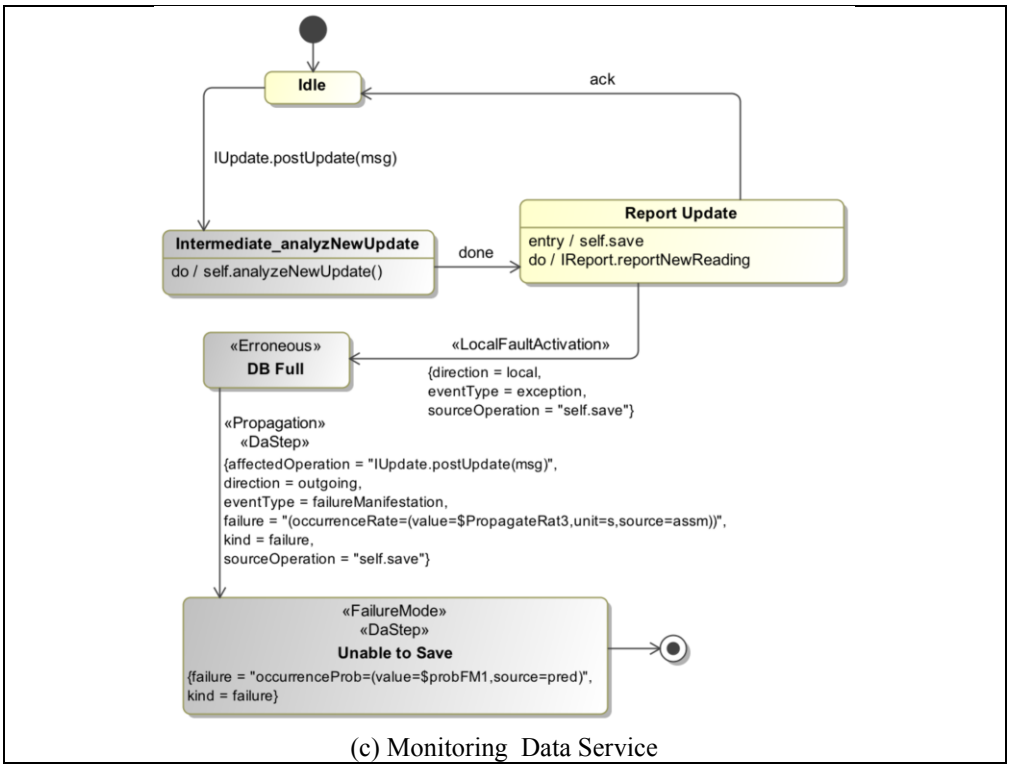
Appendix C: Application of CeBAM to Field Monitoring System



(a) Supervisory System Component

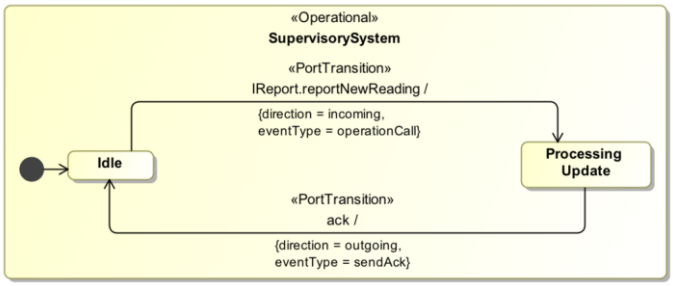


(b) Sensors Controller Component

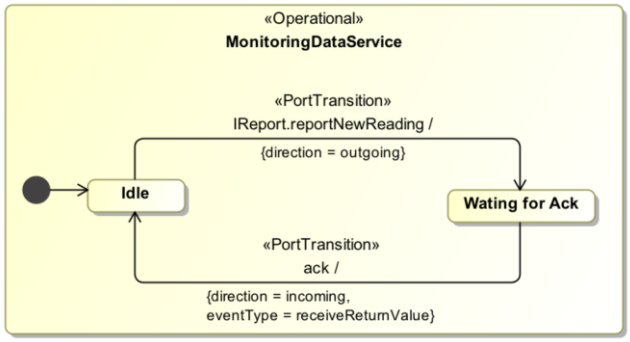


(c) Monitoring Data Service

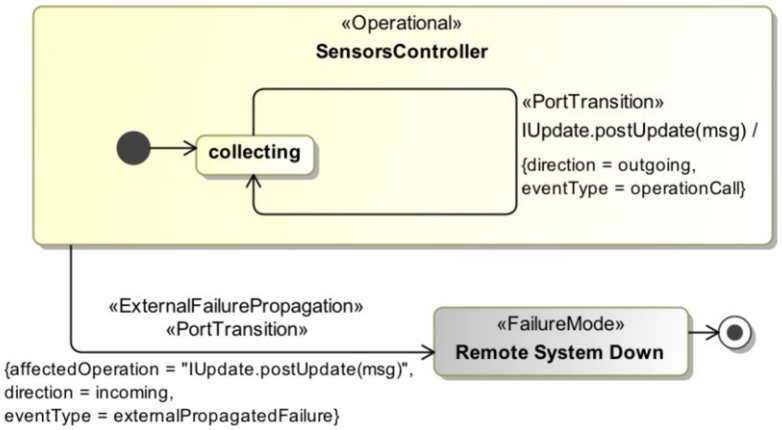
Figure C.1: FMS Component Internal Behavior after applying CeBAM Approach



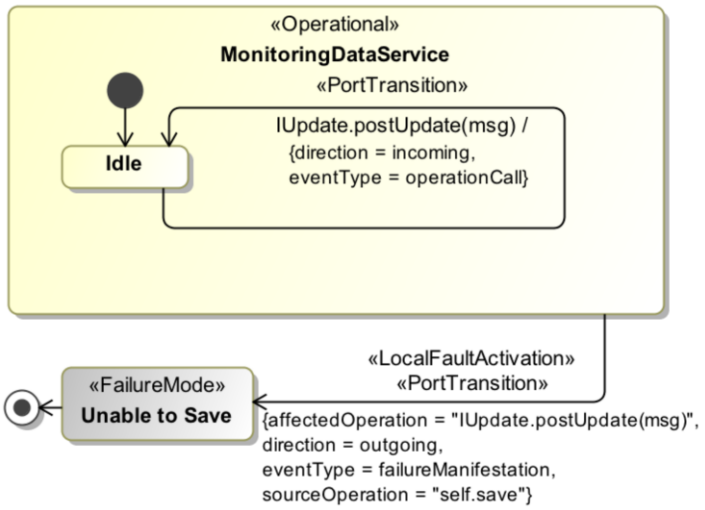
(a) Supervisory System – PReport



(b) Monitoring Data Service – RReport



(c) Sensor Controller _ RUpdate



(d) Monitoring Data Service - PUpdate

Figure C.2: FMS Component Ports Protocol Behavior after applying CeBAM

References

[1] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[2] F. Brosch, B. Buhnova, H. Koziolek, and R. Reussner, "Reliability prediction for fault-tolerant software architectures," presented at the Proceedings of the joint ACM SIGSOFT conference -- QoSA and ACM SIGSOFT symposium, New York, USA, 2011, pp. 75–84.

[3] OMG, "OMG Unified Modeling Language - Superstructure 2.3," pp. 1–758, May 2010.

[4] S. Bernardi, J. Merseguer, and D. C. Petriu, "A dependability profile within MARTE," *Softw Syst Model*, 2011.

[5] I. Crnkovic and M. Larsson, *Building Reliable Component-Based Software Systems*, 1st ed. Artech House Publishers, 2002.

[6] G. N. Rodrigues, G. Roberts, and W. Emmerich, "Reliability Support for the Model Driven Architecture," *Architecting Dependable Systems II*, pp. 79–98, 2004.

[7] G. Fairbanks, *Just Enough Software Architecture: A Risk-Driven Approach*, 1st ed. Marshall & Brainerd, 2010.

[8] S. S. Gokhale, "Architecture-Based Software Reliability Analysis: Overview and Limitations," *IEEE Transaction on Dependable and Secure Computing*, vol. 4, pp. 32–40, 2007.

[9] P. Popic, D. Desovski, W. Abdelmoez, and B. Cukic, "Error propagation in the reliability analysis of component based systems," *Proceedings 16th IEEE International Symposium on Software Reliability Engineering*, 2005.

[10] V. Cortellessa and V. Grassi, *A Modeling Approach to Analyze the Impact of Error Propagation on Reliability of Component-Based Systems*, vol. 4608, no. 10. Berlin, Heidelberg: Lecture Notes in Computer Science, 2007, pp. 140–156.

[11] J. Knight, *Fundamentals of Dependable Computing for Software Engineers*. Chapman and Hall CRC Press, 2012.

[12] K. S. Trivedi, R. Mansharamani, D. S. Kim, M. Grottke, and M. Nambiar, "Recovery from Failures Due to Mandelbugs in IT Systems," presented at the PRDC '11: Proceedings of the 2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing, 2011, pp. 224–233.

[13] M. Grottke and K. S. Trivedi, "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate," *Computer*, vol. 40, no. 2, pp. 107–109, Feb. 2007.

[14] S. M. Koriem, "Fast and simple decomposition techniques for the reliability analysis of interconnection networks," *Journal of Systems and Software*, vol. 45, no. 2, pp. 155–171, Mar. 1999.

[15] Y. Li and C. Woodside, "Complete decomposition of stochastic Petri nets representing generalized service networks," *IEEE Transactions on Computers*, vol. 44, no. 4, pp. 577–592, 1995.

[16] T. Murata, "Petri nets: Properties, analysis and applications," presented at the Proceedings of the IEEE, 1989, vol. 77, no. 4, pp. 541–580.

[17] G. Ciardo and K. S. Trivedi, "A decomposition approach for stochastic reward

net models," *Performance Evaluation*, 1993.
[18] J. Muppala, G. Ciardo, and K. S. TRIVEDI, "Stochastic reward nets for reliability prediction," *Communications in reliability, maintainability and serviceability*, vol. 1, no. 2, pp. 9–20, 1994.
[19] M. R. Lyu, *Software Fault Tolerance*. John Wiley & Sons Inc, 1994.
[20] SPNP Tool Manual, "SPNP User's Manual Version 6.0," *Duke University*, pp. 1–219, Sep. 1999.
[21] W. Hasselbring and S. Giesecke, *Dependability Engineering*, 1st ed. GITO Verlag, 2006.
[22] A. Avizienis, J.-C. Laprie, and B. Randell, "Fundamental concepts of dependability," *Technical Report No. CS-TR-739, Newcastle University*, 2001.
[23] B. Meyers and A. Schiper, "Dependable software," *to appear in Dependable Systems: Software, Computing, Networks,Lecture Notes in Computer Science, Springer-Verlag, 2006.*, pp. 1–42, Nov. 2006.
[24] I. Eusgeld, F. Fraikin, M. Rohr, F. Salfner, and U. Wappler, "Software Reliability," *Dependability Metrics Lecture Notes in Computer Science*, vol. 4909, no. 4909, pp. 1–22, Mar. 2008.
[25] R. Hanmer, *Patterns for Fault Tolerant Software*. John Wiley & Sons, 2007.
[26] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*, 1st ed. Morgan Kaufmann, 2007.
[27] H. Gomaa, "Software Modeling and Design," Cambridge University Press, Feb. 2011.
[28] L. Xing and S. V. Amari, "Fault Tree Analysis," *Handbook of performability engineering*, pp. 595–620, 2008.
[29] A. Zimmermann, "Dependability evaluation of complex systems with TimeNET," *Proceedings of the First Workshop on DYnamic Aspects in DEpendability Models for Fault-Tolerant Systems*, pp. 33–34, 2010.
[30] M. Malhotra and K. S. Trivedi, "Dependability modeling using Petri-nets," *IEEE Transactions on Reliability*, vol. 44, no. 3, pp. 428–440, 1995.
[31] J. I. Aizpurua and E. Muxika, "Design of Dependable Systems: An Overview of Analysis and Verification Approaches," *DEPEND 2012 : The Fifth International Conference on Dependability*, pp. 4–12, Jul. 2012.
[32] FMECA Technical Manual, "Failure Modes, Effects and Criticality Analysis (FMECA) for Command, Control, Communication, Computer, Intelligence, Surveillance, and Reconnaissance (C4ISR) Facilities," *Department of the Army - Headquarters*, pp. 1–75, Sep. 2006.
[33] Dong Nguyen, "Failure modes and effects analysis for software reliability," presented at the Proceedings IEEE International Conference on Cluster Computing CLUSTR-03, 2001, pp. 219–222.
[34] M. Čepin, "Reliability Block Diagram," in *Assessment of Power System Reliability*, no. 9, London: Assessment of Power System Reliability, 2011, pp. 119–123.
[35] R. A. Sahner and K. S. Trivedi, "Reliability Modeling Using SHARPE," *IEEE Transactions on Reliability*, no. 2, pp. 186–193, 1987.
[36] J. B. Dugan, S. J. Bavuso, and M. A. Boyd, "Dynamic fault-tree models for fault-tolerant computer systems," *Transactions on Reliability*, vol. 41, no. 3,

pp. 363–377, 1992.
[37] J. B. Dugan and T. S. Assaf, “Dynamic fault tree analysis of a reconfigurable software system,” *The 19th International System Safety Conference*, 2001.
[38] B. Kaiser, P. Liggesmeyer, and O. Mäckel, “A new component concept for fault trees,” *Proceedings of the 8th Australian workshop on Safety critical systems and software-Volume 33*, pp. 37–46, 2003.
[39] B. Kaiser, C. Gramlich, and M. Förster, “State/event fault trees—A safety analysis model for software-controlled systems,” *Reliability Engineering & System Safety*, vol. 92, no. 11, pp. 1521–1537, Nov. 2007.
[40] S. Bernardi, A. Bobbio, and S. Donatelli, “Petri nets and dependability,” *Lectures on Concurrency and Petri Nets*, pp. 125–179, 2004.
[41] M. Kuperberg, “Markov Models,” *Architecting Dependable Systems VI*, vol. 4909, no. 4909, pp. 48–55, Mar. 2008.
[42] R. Zurawski and M. Zhou, “Petri nets and industrial applications: A tutorial,” *IEEE Transactions on Industrial Electronics*, vol. 41, no. 6, pp. 567–583, 1994.
[43] G. J. Pai and J. B. Dugan, “Automatic synthesis of dynamic fault trees from UML system models,” *Proceedings 13th International Symposium on Software Reliability Engineering, ISSRE 2003*, pp. 243–254, 2002.
[44] J. Xiang, K. Yanoo, Y. Maeno, and K. Tadano, “Automatic Synthesis of Static Fault Trees from System Models,” presented at the 2011 International Conference on Secure Software Integration and Reliability Improvement (SSIRI), 2011, pp. 127–136.
[45] A. Joshi, S. Vestal, and P. Binns, “Automatic Generation of Fault Trees from AADL Models,” *Workshop on Architecting Dependable Systems*, 2007.
[46] C. Lauer, R. German, and J. Pollmer, “Fault tree synthesis from UML models for reliability analysis at early design stages,” *SIGSOFT Software Engineering Notes*, vol. 36, no. 1, Jan. 2011.
[47] A. Bondavalli, I. Majzik, and I. Mura, “Automated dependability analysis of UML designs,” presented at the Proceedings of 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 1999, pp. 139–144.
[48] L. Berardinelli, S. Bernardi, and V. Cortellessa, “UML Profiles for Non-functional Properties at Work: Analyzing Reliability, Availability and Performance.,” presented at the 2nd International Workshop on Non-Functional System Properties in Domain Specific Modeling Languages, 2009, p. 15.
[49] L. Montecchi, P. Lollini, and A. Bondavalli, “Towards a MDE Transformation Workflow for Dependability Analysis,” presented at the 16th IEEE International Conference on Engineering of Complex Computer Systems, 2011, pp. 157–166.
[50] J. Merseguer and S. Bernardi, “Dependability analysis of DES based on MARTE and UML state machines models,” *Discrete Event Dynamic Systems*, vol. 22, no. 2, pp. 163–178, 2012.
[51] S. Distefano, M. Scarpa, and A. Puliafito, “From UML to Petri Nets: The PCM-Based Methodology,” *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 65–79, 2011.
[52] K. S. Trivedi, *Probability and Statistics with Reliability, Queueing, and*

Computer Science Applications, 2nd ed. Wiley-Interscience, 2001.
[53] P. Clements, R. Kazman, and M. Klein, "Evaluating Software Architectures: Methods and Case Studies," *Addison-Wesley Professional*, 22-Oct-2001.
[54] B. Roy and T. C. N. Graham, "Methods for Evaluating Software Architecture: A Survey," *Technical Report No. 2008-545, School of Computing Queen's University*, pp. 1–82, Apr. 2008.
[55] P. Shanmugapriya and R. M. Suresh, "Software Architecture Evaluation Methods - A survey," *International Journal of Computer Applications*, vol. 49, pp. 19–26, Jul. 2012.
[56] K. Goseva-Popstojanova and K. S. Trivedi, "Architecture-based approach to reliability assessment of software systems," *Performance Evaluation*, pp. 179–204, 2001.
[57] K. Tyagi and A. Sharma, "Reliability of component based systems: a critical survey," *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 6, pp. 1–6, 2011.
[58] A. Immonen and E. Niemelä, "Survey of reliability and availability prediction methods from the viewpoint of software architecture," *Softw Syst Model*, vol. 7, no. 1, pp. 49–65, Jan. 2007.
[59] S. Bernardi, J. Merseguer, and D. C. Petriu, "Dependability modeling and analysis of software systems specified with UML," *ACM Computing Surveys*, vol. 45, no. 1, Nov. 2012.
[60] A. Mohamed and M. Zulkernine, "A Comparative Study on the Reliability Efforts in Component-Based Software Systems," *Technical Report No. 2009-559, School of Computing, Queen's University, Kingston, Ontario, Canada*, pp. 1–33, Jun. 2009.
[61] H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj, "A Bayesian approach to reliability prediction and assessment of component based systems," presented at the Proceedings 12th International Symposium on Software Reliability Engineering, ISSRE 2001, 2001, pp. 12–21.
[62] V. Cortellessa, H. Singh, and B. Cukic, "Early reliability assessment of UML based software models," presented at the WOSP '02: Proceedings of the 3rd international workshop on Software and performance, 2002.
[63] S. Yacoub, B. Cukic, and H. H. Ammar, "A Scenario-Based Reliability Analysis Approach for Component-Based Software," *IEEE Transactions On Reliability*, vol. 53, no. 4, pp. 465–480, Dec. 2004.
[64] Y. Si, X. Yang, X. Wang, C. Huang, and A. J. Kavs, "An architecture-based reliability estimation framework through component composition mechanisms," *2nd International Conference on Computer Engineering and Technology (ICCET)*, vol. 2, pp. V2–165–V2–170, 2010.
[65] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo, "Reliability prediction for component-based software architectures," *Journal of Systems and Software*, vol. 66, no. 3, pp. 241–252, Jun. 2003.
[66] D. Hong, T. Gu, and J. Baik, "A UML Model Based White Box Reliability Prediction to Identify Unreliable Components," presented at the Secure Software Integration & Reliability Improvement Companion (SSIRI-C), 2011 5th International Conference on, 2011, pp. 152–159.
[67] I. Majzik, A. Pataricza, and A. Bondavalli, "Stochastic Dependability Analysis

of System Architecture Based on UML Designs," *Architecting dependable systems*, pp. 219–244, 2003.

[68] R. C. Cheung, "A User-Oriented Software Reliability Model," *IIEEE Transaction Software Engineering Software Engineering*, vol. 6, no. 2, pp. 118–125, 1980.

[69] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik, "Early prediction of software component reliability," presented at the the 13th international conference, New York, New York, USA, 2008, pp. 111–120.

[70] I. Krka, L. Cheung, G. Edwards, L. Golubchik, and N. Medvidovic, "Architecture-based software reliability estimation: Problem space, challenges, and strategies," presented at the Proceedings of the …, 2008.

[71] I. Krka, G. Edwards, L. Cheung, L. Golubchik, and N. Medvidovic, "A comprehensive exploration of challenges in architecture-based reliability estimation," *Architecting Dependable Systems VI*, pp. 202–227, 2009.

[72] L. Montecchi, P. Lollini, and A. Bondavalli, "Dependability Concerns in Model-Driven Engineering," presented at the 2011 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, 2011, pp. 254–263.

[73] A.-E. Rugina, K. Kanoun, and M. Kaâniche, "A system dependability modeling framework using AADL and GSPNs," *Architecting Dependable Systems IV*, pp. 14–38, 2007.

[74] J. Musa, *Software Reliability Engineering: More Reliable Software Faster and Cheaper*, 2nd ed. Authorhouse, 2004.

[75] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Architecture-Based Reliability Prediction with the Palladio Component Model," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1319–1339, 2012.

[76] W. Abdelmoez, D. M. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H. H. Ammar, B. Yu, and A. Mili, "Error propagation in software architectures," presented at the Software Metrics, 2004. Proceedings. 10th International Symposium on, 2004, pp. 384–393.

[77] M. Hiller, A. Jhumka, and N. Suri, "PROPANE: An Environment for Examining the Propagation of Errors in Software," *SIGSOFT Softw. Eng. Notes*, pp. 81–85, 2002.

[78] R. Fredriksen and R. Winther, "Challenges related to error propagation in software systems," *Risk, Reliability and Societal Safety,aylor & Francis Group, London*, pp. 83–90, Jul. 2007.

[79] J. Voas, "Error propagation analysis for COTS systems," *Computing & Control Engineering Journal*, 1997.

[80] A. Mohamed and M. Zulkernine, "On Failure Propagation in Component-Based Software Systems," presented at the Quality Software, 2008. QSIC '08. The Eighth International Conference on, 2008, pp. 402–411.

[81] S. Shukla and K. Seth, "Failure Dependent Reliability Analysis for Component Based Software Systems," *Third International Conference on Computer and Communication Technology (ICCCT)*, pp. 149–153, 2012.

[82] P. Popic, "The Impact of Error Propagation on Software Reliability Analysis of Component-based Systems," *Master Thesis, West Virginia University*, pp. 1–

127, Sep. 2005.
[83] D. M. Nassar, W. A. Rabie, M. Shereshevsky, N. Gradetsky, H. H. Ammar, B. Yu, S. Bogazzi, and A. Mili, "Estimating Error Propagation Probabilities in Software Architectures," *International Symposium on Software Metrics No10, Chicago IL, ETATS-UNIS*, pp. 384–393, 2002.
[84] X. Ge, R. F. Paige, and J. A. McDermid, "Probabilistic Failure Propagation and Transformation Analysis," *Computer Safety*, pp. 215–228, 2009.
[85] M. A. Javed and F. U. Muram, "A framework for the analysis of failure behaviors in component-based model-driven development of dependable systems," Master Thesis,Mälardalen University, IDT Department, 2011.
[86] L. Grunske and J. Han, "A Comparative Study into Architecture-Based Safety Evaluation Methodologies Using AADL's Error Annex and Failure Propagation Models," *11th IEEE High Assurance Systems Engineering Symposium*, pp. 283–292, 2008.
[87] V. Cortellessa and P. Potena, "Path-based error propagation analysis in composition of software services," *Software Composition*, pp. 97–112, 2007.
[88] H. Aysan, S. Punnekkat, and R. Dobrin, "Error modeling in dependable component-based systems," *Annual IEEE International Computer Software and Application Conference*, pp. 1309–1314, 2008.
[89] L. Montecchi, P. Lollini, and A. Bondavalli, "An Intermediate Dependability Model for state-based dependability analysis," *Technical Report RCL101115,University of Firenze, DIP. Sistemi E Informatica*, 2011.
[90] OMG, "Object Management Group: Query View Transformation (QVT) v1.1 formal/2008," pp. 1–240, Apr. 2008.
[91] T.-T. Pham and X. Defago, "Reliability Prediction for Component-Based Systems: Incorporating Error Propagation Analysis and Different Execution Models," presented at the 2012 12th International Conference on Quality Software (QSIC 2012), pp. 106–115.
[92] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Parameterized reliability prediction for component-based software architectures," presented at the QoSA'10: Proceedings of the 6th international conference on Quality of Software Architectures: research into Practice - Reality and Gaps, 2010.
[93] H. Koziolek and F. Brosch, "Parameter Dependencies for Component Reliability Specifications," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 1, pp. 23–38, Oct. 2009.
[94] R. Adler, D. Domis, K. Höfig, S. Kemmann, T. Kuhn, J.-P. Schwinn, and M. Trapp, "Integration of component fault trees into the UML," *Models in Software Engineering*, pp. 312–327, 2011.
[95] L. Grunske and B. Kaiser, "An automated dependability analysis method for COTS-based systems," *4th International Conference, ICCBSS on COTS-Based Software Systems*, pp. 178–190, 2005.
[96] S. Bernardi, J. Merseguer, and D. C. Petriu, *Model-Driven Dependability Assessment of Software Systems*. Springer Science & Business Media, 2013.
[97] S.-W. Leu, E. B. Fernandez, and T. Khoshgoftaar, "Fault-tolerant software reliability modeling using Petri nets," *Microelectronics Reliability*, vol. 31, no. 4, pp. 645–667, Jan. 1991.

[98] G. Ciardo and J. K. Muppala, “Analyzing concurrent and fault-tolerant software using stochastic reward nets,” *Journal of Parallel and Distributed Computing*, vol. 15, no. 3, pp. 255–269, Jul. 1992.
[99] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*. Artech House Publishers, 2001.
[100] J. Kienzle, “Software fault tolerance: An overview,” *Reliable Software Technologies—Ada-Europe 2003*, pp. 45–67, 2003.
[101] G. K. Saha, “Approaches to Software Based Fault Tolerance – A Review,” *Computer Science Journal of Moldova*, vol. 13, no. 2, pp. 1–39, Dec. 2005.
[102] H. Muccini and A. Romanovsky, *Architecting Fault Tolerant Systems*. University of Newcastle, Technical Report - No. CS-TR-1051, 2007, pp. 1–70.
[103] Z. Xie, H. Sun, and K. Saluja, “A Survey of Software Fault Tolerance Techniques,” *University of Wisconsin-Madison/Department of Electrical and Computer Engineering*, 2006.
[104] W. Torres-Pomales, *Software Fault Tolerance: A Tutorial, NASA*. 2000.
[105] L. Michotte, R. B. France, and F. Fleurey, “Modeling and Integrating Aspects into Component Architectures,” presented at the 11th IEEE International Enterprise Distributed Object Computing Conference, 2007, pp. 181–181.
[106] P. Domokos and I. Majzik, “Aspect-oriented modelling and analysis of information systems,” *periodica polytechnica Electrical Engineering*, vol. 51, no. 1, p. 21, 2007.
[107] P. Domokos and I. Majzik, “Design and analysis of fault tolerant architectures by model weaving,” *Ninth IEEE International Symposium on High-Assurance Systems Engineering*, pp. 15–24, 2005.
[108] L. Michotte, T. Vergnaud, P. H. Feiler, and R. B. France, “Aspect Oriented Modeling of Component Architectures Using AADL,” *New Technologies, Mobility and Security, 2008. NTMS '08*, pp. 1–6, 2008.
[109] J. Li, X. Chen, G. Huang, H. Mei, and F. Chauvel, “Selecting fault tolerant styles for third-party components with model checking support,” *12th International Symposium On Component-Based Software Engineering,CBSE 2009*, pp. 69–86, 2009.
[110] N. B. Harrison and P. Avgeriou, “Incorporating fault tolerance tactics in software architecture patterns,” presented at the SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, New York, New York, USA, 2008, p. 9.
[111] C. A. Lewis, R. W. Smith, and A. Beaulieu, “A model driven framework for N-version programming,” presented at the Systems Conference (SysCon), 2011 IEEE International, 2011, pp. 59–65.
[112] G. Denaro and M. Pezzè, “Petri nets and software engineering,” *Lectures on Concurrency and Petri Nets*, pp. 439–466, 2004.
[113] G. Ciardo, J. Muppala, and T. Trivedi, “SPNP: stochastic Petri net package,” presented at the Petri Nets and Performance Models, 1989. PNPM89., Proceedings of the Third International Workshop on, 1989, pp. 142–151.
[114] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley, 2012.
[115] G. N. Rodrigues, D. S. Rosenblum, and S. Uchitel, “Reliability prediction in

model-driven development," *Model Driven Engineering Languages and Systems*, pp. 339–354, 2005.
[116] N. Alzahrani and D. C. Petriu, "Modeling Component Erroneous Behavior and Error Propagation for Dependability Analysis," presented at the SDL 2013: Model-Driven Dependability Engineering, Berlin, Heidelberg, 2013, vol. 7916, no. 8, pp. 20–143.
[117] OMG, "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems," pp. 1–754, Jun. 2011.
[118] R. Yedduladoddi, *Aspect Oriented Software Development: An Approach to Composing UML Design Models*. VDM Publishing, 2009.
[119] V. Mencl, "Enhancing Component Behavior Specifications with Port State Machines," *Technical Report Charles University*, p. 13, 2003.
[120] Y. Moffett, A. Beaulieu, and J. Dingel, "Verifying UML-RT protocol conformance using model checking," presented at the MODELS'11: Proceedings of the 14th international conference on Model driven engineering languages and systems, 2011, pp. 410–424.
[121] D. C. Craig and W. M. Zuberek, "Compatibility of Software Components - Modeling and Verification," presented at the Proceedings of the International Conference on Dependability of Computer Systems, 2006.
[122] B. Tuffin, P. K. Choudhary, C. Hirel, and K. S. Trivedi, "Simulation versus analytic-numeric methods: illustrative examples," presented at the ValueTools '07: Proceedings of the 2nd international conference on Performance evaluation methodologies and tools, 2007.
[123] Y. Li, "Solutions Techniques for Stochastic Petri Nets," curve.carleton.ca, Phd Thesis, Carleton, Canada, Ottawa.
[124] B. Selic, "A systematic approach to domain-specific language design using UML," *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pp. 2–9, 2007.
[125] S. Ali, L. C. Briand, and H. Hemmati, "Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems," *Softw Syst Model*, vol. 11, no. 4, pp. 633–670, Jun. 2011.
[126] N. A. M. Alzahrani and D. C. Petriu, "Modeling Fault Tolerance Tactics with Reusable Aspects," presented at the QoSA '15: Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, 2015.
[127] N. S. Teixeira and R. P. E. Silva, "Compatibility Evaluation of Components Specified in UML," presented at the Computer Science Society (SCCC), 2011 30th International Conference of the Chilean, 2011, pp. 90–99.
[128] Y. Moffett, "UML-RT Protocol Conformance Verification Through Exhaustive Exploration From theory to implementation,", PhD Thesis, Royal Military Collage, Canada, 2010.
[129] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek, "Conformance checking of service behavior," *Transactions on Internet Technology (TOIT*, vol. 8, no. 3, pp. 1–30, May 2008.
[130] OMG, "MOF Model to Text Transformation Language, v1.0," pp. 1–48, Feb. 2008.

[131] "MagicDraw," *nomagic.com*. [Online]. Available: http://www.nomagic.com/products/magicdraw.html. [Accessed: 21-Sep-2015].
[132] "Bug 410511," *bugs.eclipse.org*. [Online]. Available: https://bugs.eclipse.org/bugs/show_bug.cgi?id=410511. [Accessed: 21-Sep-2015].
[133] "Bug 425116," *bugs.eclipse.org*. [Online]. Available: https://bugs.eclipse.org/bugs/show_bug.cgi?id=425116. [Accessed: 21-Sep-2015].
[134] "Bug 428309," *bugs.eclipse.org*. [Online]. Available: https://bugs.eclipse.org/bugs/show_bug.cgi?id=428309. [Accessed: 21-Sep-2015].
[135] "Eclipse Modeling Project," *eclipse.org*. [Online]. Available: http://www.eclipse.org/modeling/. [Accessed: 21-Sep-2015].
[136] M. Mussa, S. Ouchani, W. Al Sammane, and A. Hamou-Lhadj, "A Survey of Model-Driven Testing Techniques," *Quality Software, 2009. QSIC '09. 9th International Conference on*, pp. 167–172, 2009.
[137] A. Z. Javed, P. A. Strooper, and G. N. Watson, "Automated Generation of Test Cases Using Model-Driven Architecture," presented at the Second International Workshop on Automation of Software Test, 2007.
[138] A. Ciancone, A. Filieri, and R. Mirandola, "Testing operational transformations in model-driven engineering," *Innovations Syst Softw Eng*, vol. 10, no. 1, pp. 19–32, 2014.
[139] OMG, Ed., "Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification 1.1," pp. 1–246, Jan. 2011.
[140] N. A. M. Alzahrani and D. C. Petriu, "Derivation of Stochastic Reward Net for Compatibility and Conformance Verification of Component Erroneous Behavior Model," presented at the IEEE 17th Pacific Rim International Symposium on Dependable Computing (PRDC), 2013, pp. 142–151.
[141] G. Ciardo and K. S. Trivedi, "A decomposition approach for stochastic reward net models," *Performance Evaluation*, vol. 18, no. 1, pp. 37–59, 1993.
[142] C. M. Woodside and Y. Li, "Performance Petri net analysis of communications protocol software by delay-equivalent aggregation," presented at the Petri Nets and Performance Models, 1991. PNPM91., Proceedings of the Fourth International Workshop on, 1991, pp. 64–73.