UNIVERSITÀ
DEGLI STUDI
FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Dottorato di Ricerca in Informatica, Sistemi e Telecomunicazioni
Ciclo XXVI
(Anni 2011/2013)

Settore Scientifico Disciplinare: INF/01

A METHODOLOGY AND FRAMEWORK FOR MODEL-DRIVEN DEPENDABILITY ANALYSIS OF CRITICAL EMBEDDED SYSTEMS AND DIRECTIONS TOWARDS SYSTEMS OF SYSTEMS

LEONARDO MONTECCHI

Supervisor: *Prof. Andrea Bondavalli*

PhD Coordinator: *Prof. Luigi Chisci*

December 2013

Leonardo Montecchi: *A Methodology and Framework for Model-Driven Dependability Analysis of Critical Embedded Systems and Directions Towards Systems of Systems*, Dottorato di Ricerca in Informatica, Sistemi e Telecomunicazioni, Ciclo XXVI, Università degli Studi di Firenze.

ABSTRACT

In different domains, engineers have long used models to assess the feasibility of system designs; over other evaluation techniques modeling has the key advantage of not exercising a real instance of the system, which may be costly, dangerous, or simply unfeasible (e.g., if the system is still under design).

In the development of critical systems, modeling is most often employed as a fault forecasting technique [6], since it can be used to estimate the degree to which a given design provides the required dependability attributes, i.e., to perform quantitative dependability analysis. More in general, models are employed in the evaluation of the Quality of Service (QoS) provided by the system, under the form of dependability [6], performance, or performability [125] metrics. From an industrial perspective, modeling is also a valuable tool in the Verification & Validation (V&V) process, either as a support to the process itself (e.g., FTA [174]), or as a means to verify specific quantitative or qualitative requirements.

Modern computing systems have become very different from what they used to be in the past: their scale is growing, they are becoming massively distributed, interconnected, and evolving. Moreover, a shift towards the use of off-the-shelf components is becoming evident in several domains. Such increase in complexity makes model-based assessment a difficult and time-consuming task. In the last years, the development of system has increasingly adopted the Component-Based Development (CBD) and Model-Driven Engineering (MDE) philosophies as a way to reduce the complexity in system design and evaluation. CBD refers to the established practice of building a system out of reusable "black-box" components, while MDE refers to the systematic use of models as primary artefacts throughout the engineering lifecycle [168]. Engineering languages like UML, BPEL, AADL, etc., allow not only a reasonable unambiguous specification of designs, but also serve as the input for subsequent development steps like code generation, formal verification, and testing. One of the core technologies supporting model-driven engineering is model transformation [58].

Transformations can be used to refine models, apply design patterns, and project design models to various mathematical analysis domains in a precise and automated way. In recent years, model-driven engineering approaches have been also extensively used for the analysis of the extra-functional properties of the systems. To this purpose, language extensions were introduced and utilized to capture the required extra-functional concerns.

Despite several approaches propose model transformations for dependability analysis, still there is not a standard approach for performing dependability analysis in a MDE environment. Indeed, when targeting critical embedded

systems, the lack of support for dependability attributes, and extra-functional attributes in general, is one of the most recognized weaknesses of UML-based languages. Also, most of the approaches have been defined as extensions to a "general" system development process, often leaving the actual process unspecified. Similarly, supporting tools are typically detached from the design environment, and assume to receive as input a model satisfying certain constraints. While in principle such approach allows not to be bound to specific development methodologies, in practice it introduces a gap between the design of the functional system model, its enrichment with dependability information, and the subsequent analysis. Finally, the specification of properties our of components' context, which typically holds for functional properties, is much less understood for non-functional properties.

The work in this thesis elaborates on the combined application of the CBD and MDE philosophies and technologies, with the aim to automate dependability analysis of modern computing systems. A considerable part of the work described in this thesis has been carried out in the context of the ARTEMIS-JU "CHESS" project [35], which aimed at defining, developing and assessing a methodology for the component-based design and development of embedded systems, using model-driven engineering techniques.

The work in this thesis defines and realizes an extension to the CHESS framework for the automated evaluation of quantitative dependability properties. The extension constitutes of: i) a set of UML language extensions, collectively referred to as DEP-UML, for modeling dependability properties relevant for quantitative analysis; ii) a set of model-transformation rules for the automated generation of Stochastic Petri Nets (SPNs) models from system designs enriched with DEP-UML; and iii) a model-transformation tool, realized as a plugin for the Eclipse platform, concretely implementing the approach. After introducing the approach, we detail its application with two case studies.

While for embedded systems it is often possible, or even mandatory, to follow and control the whole design and development process, the same does not hold for other classes of systems and infrastructures. In particular, large-scale complex systems don't fit well in the paradigm proposed by the CHESS project, and alternative approaches are therefore needed. Following this observation, we then elaborate on a workflow for applying MDE approaches to support the modeling of large-scale complex systems. The workflow is based on a particular modeling technique, and a supporting domain-specific language, TMDL, which is defined in this thesis. After introducing a motivating example, the thesis details the workflow, introduces the TMDL language, describes a prototype realization of the approach, and describes the application of the approach to two examples. We then conclude with a discussion and a future view on how the contribution of this thesis can be extended to a comprehensive approach for dependability and performability evaluation in a "System of Systems" [181] context.

More in detail, this dissertation is organized as follows. Chapter 1 introduces the context of the work, describing the main concepts related to dependability, and dependability evaluation, with a focus on model-based assessment. The foundation of CBD and MDE approaches, the role of the UML language, and main related work are instead discussed in Chapter 2.

Chapter 3 describes the CHESS project, and introduces the language extensions that have been defined to support dependability analysis. Moreover, the chapter details the entire process that drove us to such extensions, including the elicitation of language requirements and the evaluation of existing languages in the literature. The model-transformation algorithms for the generation of Stochastic Petri Nets are described in Chapter 4, while the adopted architecture for the concrete realization of the analysis plugin is described in Chapter 5. Chapter 6 describes the application of our approach to two case studies: of a multimedia processing workstation and a fire detection system.

The need for a complementary approach for the evaluation of large-scale complex system is discussed in Chapter 7, with the aid of a motivating example of a distributed multimedia application. Chapter 8 describes our approach for the automated assembly of large dependability models through model-transformation. The thesis then concludes with an outlook on the relevance of the work presented in this thesis towards a System of Systems approach to the evaluation of large-scale complex systems.

RELATED PUBLICATIONS

This thesis is partially based on work included in the following publications:

I. L. Montecchi, P. Lollini, and A. Bondavalli. "A Reusable Modular Toolchain for Automated Dependability Evaluation." In: *7th International Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS'13 (Turin, Italy, Dec. 10–12, 2013). 2013;

II. L. Montecchi, A. Ceccarelli, P. Lollini, and A. Bondavalli. "Meeting the challenges in the design and evaluation of a trackside real-time safety-critical system." In: *Proceedings of 4th IEEE Workshop on Self-Organizing Real-Time Systems*. SORT'13 (Paderborn, Germany, June 20, 2013). 2013;

III. N. Veeraragavan, L. Montecchi, N. Nostro, A. Bondavalli, R. Vitenberg, and H. Meling. "Understanding the Quality of Experience in Modern Distributed Interactive Multimedia Applications in Presence of Failures: Metrics and Analysis." In: *Proceedings of the 28th ACM Symposium on Applied Computing*. SAC'13 (Coimbra, Portugal, Mar. 18–22, 2013). DADS Track. ACM, 2013;

IV. A. Bondavalli, P. Lollini, I. Majzik, and L. Montecchi. "Modelling and Model-Based Assessment." In: *Resilience Assessment and Evaluation of Computing Systems*. Ed. by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel. Springer, July 2012, pp. 153–165;

V. A. Bondavalli, P. Lollini, and L. Montecchi. "Graphical formalisms for modeling critical infrastructures." In: *Critical Infrastructure Security: Assessment, Prevention, Detection, Response*. Ed. by F. Flammini. WIT Press, Feb. 2012, pp. 57–73

VI. L. Montecchi, P. Lollini, and A. Bondavalli. "Towards a MDE Transformation Workflow for Dependability Analysis." In: *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems*. ICECCS'11 (Las Vegas, NV, USA, Apr. 27–29, 2011). IEEE, 2011, pp. 157–166;

VII. L. Montecchi, P. Lollini, and A. Bondavalli. "Dependability Concerns in Model-Driven Engineering." In: *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. WORNUS'11 (Newport Beach, CA, USA, Mar. 28–31, 2011). IEEE, 2011, pp. 254–263;

and the following currently submitted work:

VIII. L. Montecchi, P. Lollini, and A. Bondavalli. *A DSL-Supported Workflow for the Automated Assembly of Large Performability Models*. *Submitted to* the 10th European Dependable Computing Conference (EDCC 2014).

The following publications are related to the topic of the thesis as well, but where published before the beginning of the Ph.D. course:

IX. A. Ceccarelli, J. Grønbæk, L. Montecchi, H.-P. Schwefel, and A. Bondavalli. "Towards a Framework for Self-Adaptive Reliable Network Services in Highly-Uncertain Environments." In: *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. ISORCW'10 (Carmona, Spain, May 4–7, 2010). IEEE, 2010, pp. 184–193;

X. A. Bondavalli, P. Lollini, and L. Montecchi. "QoS Perceived by Users of Ubiquitous UMTS: Compositional Models and Thorough Analysis." In: *Journal of Software* 4.7 (Sept. 2009);

XI. P. Lollini, L. Montecchi, M. Magyar, I. Majzik, and A. Bondavalli. "Analysis of the impact of communication protocols on service quality in ERTMS automatic train control systems." In: *Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems*. (FORMS/FORMAT'08) (Budapest, Hungary, Oct. 9–10, 2008). 2008;

XII. A. Bondavalli, P. Lollini, and L. Montecchi. "Analysis of User Perceived QoS in Ubiquitous UMTS Environments Subject to Faults." In: *6th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*. SEUS'08 (Capri Island, Italy, Oct. 1–3, 2008). Springer, 2008, pp. 186–197.

Finally, the following work, published within the Ph.D. course, is only marginally related to the topics of this thesis:

XIII. A. Ceccarelli, L. Montecchi, F. Brancati, P. Lollini, A. Marguglio, and A. Bondavalli. "Continuous and Transparent User Identity Verification for Secure Internet Services." In: *IEEE Transactions on Dependable and Secure Computing* (*To Appear*);

XIV. V. Bonfiglio, L. Montecchi, F. Rossi, and A. Bondavalli. "On the Need of a Methodological Approach for the Assessment of Software Architectures within ISO26262." In: *Proceedings of Workshop CARS (2nd Workshop on Critical Automotive applications: Robustness & Safety) of the 32nd International Conference on Computer Safety, Reliability and Security*. SAFECOMP'13/CARS'13 (Toulouse, France, Sept. 24–27, 2013). 2013;

XV. L. Montecchi, P. Lollini, A. Bondavalli, and E. La Mattina. "Quantitative Security Evaluation of a Multi-biometric Authentication System." In: *Computer Safety, Reliability, and Security. Proceedings of SAFECOMP 2012 Workshops*. SAFECOMP'12/DESEC4LCCI'12 (Magdeburg, Germany, Sept. 25–28, 2012). Ed. by F. Ortmeier and P. Daniel. Vol. 7613. LNCS. Springer, 2012, pp. 209–221;

XVI. L. Montecchi, P. Lollini, B. Malinowsky, J. Grønbæk, and A. Bondavalli. "Model-based analysis of a protocol for reliable communication in railway worksites." In: *Proceedings of the 15th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*. (MSWiM'12) (Paphos, Cyprus, Oct. 21–25, 2012). ACM, 2012, pp. 23–32;

XVII. A. Bondavalli and L. Montecchi. "Metodi Combinatori." Italian. In: ed. by A. Bondavalli. L'Analisi Quantitativa dei Sistemi Critici. Esculapio, 2011.

ACKNOWLEDGMENTS

I would like to deeply thank my supervisor, Prof. Andrea Bondavalli, for his continuous support and motivation to my study and research activities. On many aspects, his useful insights really helped me to have a broader point of view on problems and their solutions.

I would also like to express my sincere gratitude to Paolo Lollini, for his incomparable availability and contribution with comments, ideas, and suggestions on many topics presented in this work. Sincere thanks also to the other present and former members of the Resilient Computing Lab (RCL) research group, Valentina Bonfiglio, Francesco Brancati, Andrea Ceccarelli, Nicola Nostro, for the enjoyable time spent together and the stimulating discussions and collaborations.

My great appreciation goes also to all the people who participated in the National and International research projects in which I have been involved, especially the ARTEMIS-JU "CHESS" project, which has been a tough but enriching experience. In particular, I would like to thank Stefano Puri for his precious support on technical UML aspects, and Silvia Mazzini, Barbara Gallina, Tullio Vardanega for the useful discussions.

I would also like to thank all the other collaborators of my research, in particular Jesper Grønbæk, for the discussions on online diagnosis and reconfiguration, and Narasimha Raghavan, for the discussions on the World Opera system; both topics have been inspiring to me when shifting my attention to large-scale systems.

I am also thankful to all the people who have been working in the PhD room in these years, each of them really contributed to a better personal and professional experience.

Special appreciation is deserved to my parents, to whom I owe a debt of gratitude. Finally, thanks to Aurora, my cousin Serena, my aunt Maria and my uncle Mario; all of them have been reference points during the last three years, and I am grateful for their very different ways of supporting me during this period.

— Leonardo Montecchi

CONTENTS

1

DEPENDABILITY AND PERFORMABILITY EVALUATION

Information Technology (IT) has become widespread in our everyday life; modern society is increasingly relying on computerized systems for fundamental services like communications, transportation, power distribution. Many people need to rely upon the services provided by these system, as their malfunctions can cause very serious consequences both in terms of loss of human's life or in terms of conspicuous economical losses. It is therefore necessary to ensure that such *critical systems* are designed to fulfill specific properties, in order for us to "sufficiently trust" the service they provide. The set of these properties is studied in the conceptual framework of dependability.

1.1 DEPENDABILITY AND PERFORMABILITY CONCEPTS

The concepts described in this thesis are based on the universally accepted definition and taxonomy of dependability described in [6]. According to the original definition, *dependability* is "the ability to deliver service that can justifiably be trusted". An alternate definition, which also provides a way to judge *if* a given service is dependable is "the ability of a system to avoid service failures that are more frequent or more severe than is acceptable".

Dependability is a composite concept that encompasses number of different *attributes*, including:

- *availability*, readiness of correct service, i.e., the ability to deliver correct service with respect to the alternation between correct and incorrect service;
- *reliability*, continuity of correct service, i.e., the ability to continuously deliver correct service;
- *maintainability*, the ability to undergo repairs and modifications;
- *safety*, the absence of catastrophic consequences on the user(s) and the environment;
- *integrity*, the absence of improper system state alterations.
- *confidentiality*, the absence of unauthorized disclosure of information.

Security is defined as a composite of the attributes of confidentiality, integrity, and availability (for authorized actions only).

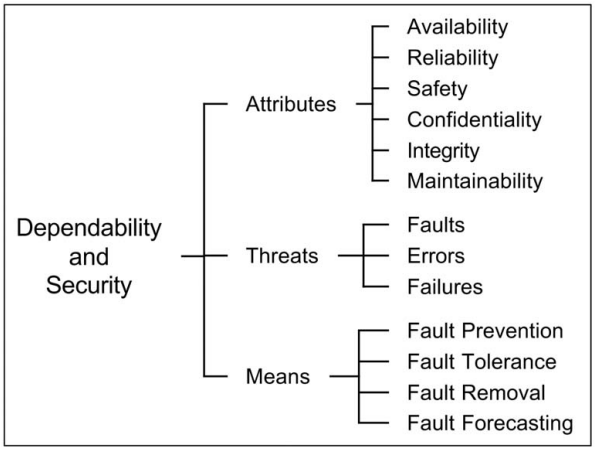


Figure 1.1: The dependability and security tree [6].

Quantitative dependability evaluation involves the *quantification* of such attributes by means of *metrics* using different techniques, e.g., the Mean Time To Failure (MTTF) is a metric used to quantify the reliability attribute of a system. In certain classes of systems, referred to as degradable systems, dependability attributes are intertwined with performance aspects: the failure of system components can affect the overall system performance. This lead to the definition of *performability* [125], as a combination of performance and reliability concepts, thus describing the performance provided by the system in degraded service states. It should be noted that systems are typically not required to excel in *all* the attributes described above. Actually, some of them are in contrast to each other; for example, a way to avoid catastrophic consequences (safety) is to force the system into a "safe state", interrupting the service it provides (thus affecting its reliability).

A systematic description of dependability concepts spans its *attributes* (introduced above), *threats*, and *means* (Figure 1.1). It is practically impossible to guarantee that a system will be always working properly during its entire lifecycle: in the achievement of its dependability *attributes* a system has to face a certain number of *threats*, which should be contrasted with appropriate *means*, i.e., countermeasures. These concepts are detailed in the following.

1.1.1 *Basic definitions*

We first clarify the *system* concept: "a system is an entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena" [6]. These other entities with which the system interacts are its *environment*.

The *function* of a system is what the system is intended to do, and it is described by the functional specification. The *behavior* of a system is what the system does to implement its function, and it is described by a sequence of states. The *service* delivered by a system (called the *provider*) is its behavior as it is perceived by its user(s); a user is another system that receives such service. The part of the provider's system boundary where service delivery takes place, is provider's *service interface*. The part of the provider's total state that is perceivable at the service interface is its *external state*; the remaining part is its *internal state*. The delivered service is a sequence of the provider's external states.

The *structure* of a system is what enables it to generate the behavior. From a structural viewpoint, a system is composed of a set of components bound together in order to interact, where each component can be considered another system. The total state of a system is then the set of the external states of its atomic components. The recursion stops when a component is considered to be atomic: any further internal structure cannot be discerned, or is not of interest and can be ignored.

1.1.2 *Threats: faults, errors, failures*

Correct service is delivered when the service implements the system function. A *failure* is an event that occurs when the delivered service deviates from correct service. A system may not, and generally does not, always fail in the same way; the ways a system can fail are called *failure modes*, and they can be characterized by their domain (value or timing), by the perception of users (consistent and byzantine), by the capability to detect them (signalled or unsignaled) and by the consequences on the system environment (from benign to catastrophic). An *error* is a deviation from the correct system state. Failures are caused by errors: a failure occurs when an error reaches the service interface and alters the provided service.

A *fault* is an adjudged or hypothesized cause of an error; an error is then the manifestation of a fault within a program or data structure. With respect to its duration, it can be permanent, intermittent or transient: a *permanent fault* is continuous and stable, while a *transient fault* results from temporary environmental conditions. An *intermittent fault* is a fault that is only occasionally present due to instability in the system or the environment. Faults can also be classified with respect to their origin. Physical faults arise from physical phenomena either internal (such as shorts or opens), or external to the system (such as electromagnetic interference). Human faults may be either design faults, which are committed during system design or modification, they may be interaction faults, which are violations of operating or maintenance procedures. A complete classification of faults is provided in [6].

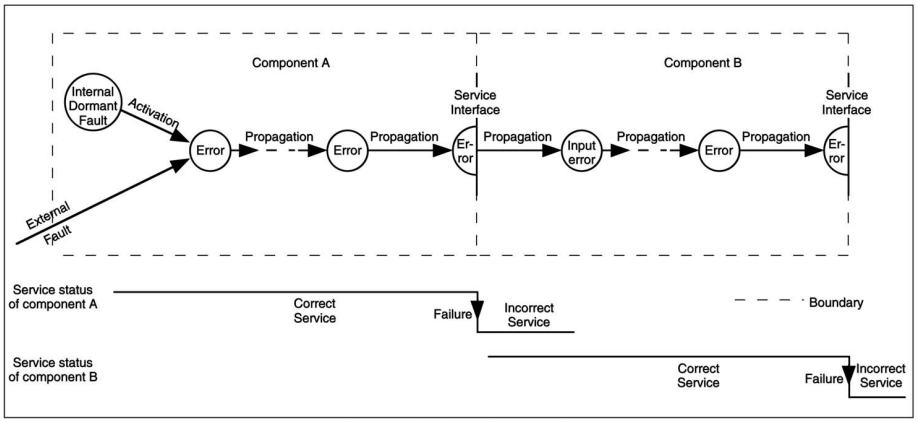


Figure 1.2: Error propagation [6].

A fault that has not yet produced an error is a *dormant* fault, otherwise it is an *active* fault. An existing dormant fault is said to become active when some change in conditions (e.g. an input applied to the component) causes the fault to affect the computation. An active fault is either i) an internal fault that was previously dormant and that has been activated, or ii) an external fault. The computation process can then produce other errors that can propagate within the same component (internal propagation), so that an error successively generates other errors. Error propagation from one component (A) to another component (B) that receives service from A (i.e., external propagation) occurs when an error reaches the service interface of component A. At this time, the service delivered by A to B becomes incorrect, and the ensuing failure of A appears to B as an external fault, thus propagating an error in the internal state of B.

Faults, errors and failures are collectively referred to as the "threats" to dependability, as they can induce a system to deliver an incorrect service (or to deliver no service), thus "threatening" the system in its objective of achieving the target dependability attributes. The causality relationships between faults, errors and failures is summarized in Figure 1.2.

1.1.3 *The means for achieving dependability*

To contrast the existing threats, the development of dependable systems requires the combined application of different techniques (means): *fault prevention*, to prevent faults to occur or be introduced in the system; *fault tolerance*, to deliver a correct service even in presence of faults; *fault removal*, to reduce the number or severity of faults; *fault forecasting*, to estimate the number of faults that are present in the system, their future impact, and their possible consequences.

FAULT PREVENTION. Fault prevention is performed thanks to specific development techniques and quality assessment processes during software design and the development of hardware components. Such techniques include for example structured programming and modularity for what concerns software, and rigorous productive processes for what concerns hardware. Physical faults can be prevented through specific protections, e.g., shielding from electromagnetic interference. Fault originating from human interactions can instead be prevented through training of personnel or strict procedures. Faults originating from external attacks can be prevented for example through firewalls and other security devices or policies.

FAULT TOLERANCE. Fault tolerance aims at preserving a correct service in presence of active faults. It is typically implemented through error detection and subsequent recovery of system state. The recovery of system state transforms a state containing one or more errors (and possibly faults) in a state that does not contains errors, or faults that can be activated again.

Recovery consists of error handling and fault handling. *Error handling* removes errors from system's state through either i) rollback, in which the system is reverted to a previous state; ii) rollforward, in which the system is moved to a completely new state; or iii) compensation, in which the system state contains enough redundancy to mask the error. *Fault handling* prevents identified faults to be activated again, through four phases: i) fault diagnosis, ii) fault isolation, iii) system reconfiguration, iv) system reinitialization. Fault handling is usually followed by corrective maintenance activities, which remove faults isolated by fault handling, e.g., replacing a component that has been marked as faulty.

FAULT REMOVAL. Fault removal can be performed both during the development phase, and during the operation life of the system. Fault removal is also one of the objectives of the V&V process. Verification is the process of checking whether the system adheres to given properties; if it does not, two steps should be undertaken: diagnosing the fault(s) that prevented the verification conditions from being fulfilled, and then performing the necessary corrections. Checking the specification is instead usually referred to as validation.

Verification techniques can be classified based on the need to exercise the system. Static verification verifies the system without its actual execution, e.g., via static analysis, theorem proving, or model-checking techniques. Verifying the system through exercising it constitutes dynamic verification; supplied inputs can be either symbolic (symbolic execution), or actual in the case of testing techniques. Corrective or preventive maintenance is a fault removal technique that is applied during the operational life of the system.

FAULT FORECASTING. Fault forecasting is performed by evaluating the system behavior with respect to the occurrence and activation of faults. Evaluation

can be either qualitative, aiming at identifying and classifying the combination of events that can lead the system to fail, or quantitative (i.e., probabilistic), aiming at evaluate in probabilistic terms the degree to which dependability attributes are satisfied by the system.

The two main complementary approaches to probabilistic fault forecasting, aimed to derive probabilistic estimates, are *modeling* and *testing*.

1.2 MODEL-BASED EVALUATION

In model-based evaluation [141], a model is an abstraction of a system that highlights the important features of the system and provides ways of quantifying its properties, neglecting all those details that are relevant for the actual implementation, but that are marginal for the objective of the study.

Models play a primary role in dependability and performability assessment of modern computing systems. They are employed, in different forms, to apply all the techniques described above. Models to support the generation of code implementing specific design patterns is a fault prevention technique; models as a support to diagnosis (e.g., see [59]) are a fault tolerance technique; the use of model-checking to verify protocols and algorithms is a fault removal technique. In this thesis we focus on modeling as a *fault forecasting* technique, i.e., as a method to derive probabilistic estimates of system dependability metrics.

As a fault-forecasting technique, model-based evaluation allow system architects to understand and learn about specific aspects of the system, to detect possible design weak points or bottlenecks, to perform early validation of dependability requirements, or to suggest solutions for future releases or modifications of the systems. Within the domain of critical systems, modeling is a valuable tool since it avoids to perform analysis, e.g., "what-if" analyses, on a real instance of the system, which may be costly, dangerous or simply unfeasible. Modeling is also of primary importance as a support to the design process, in which the real system is not yet available.

Assessing the resilience of composite systems, is a difficult task that may require the combination of several assessment methods and approaches. In this perspective, models can be profitably used as support for experimentation and vice-versa. On one side, modelling can help in selecting the features and measures of interest to be evaluated experimentally, as well as the right inputs to be provided for experimentation. On the other side, the measures assessed experimentally can be used as parameters in the models, and the features identified during the experimentation may impact the semantics of the dependability model.

1.2.1 *Modeling Formalisms*

Modeling is composed of two phases: i) the construction of a model of the system from the elementary stochastic processes that model the behavior of the components of the system and their interactions; and ii) processing the model to obtain the expressions and the values of the dependability measures of the system.

Research in dependability analysis has led to a variety of modeling formalisms. Each of these techniques has its own strengths and weaknesses in terms of accessibility, ease of construction, efficiency and accuracy of solution algorithms, and availability of supporting software tools. The choice of the most appropriate model depends upon the complexity of the system, the questions to be answered, the accuracy required, and the resources available to the study.

In the following, we provide an overview of modeling formalism that are most common in model-based evaluation of dependable systems.

Combinatorial models

Modeling formalisms can be broadly classified into combinatorial (non-state-space) models and state-space models. In contrast with state-space models, combinatorial models do not enumerate all possible system states to obtain a solution. Instead, simpler approaches are used to compute system dependability measures [141]. While being concise, easy to understand, and supported by efficient evaluation methods, such methods require strong assumptions to be made on the system. Typically, realistic features such as interrelated behaviour of components, imperfect coverage, non-zero reconfiguration delays, and combination with performance can not be captured by these models.

Despite some extensions to "classical" combinatorial models introduce primitives to specify some kinds of dependencies between components (e.g., see [67]), their modeling power is still limited with respect to that offered by state-space models.

The most widespread combinatorial model in dependability analysis is arguably the Fault Trees (FTs) formalism [174]. A fault tree is a connected acyclic graph (i.e., a tree), in which internal nodes are logic gates (e.g., AND, OR, k-out-of-n) and leaves represent "basic events", typically failures of system components. The root of the tree, also known as "top event", represents the failure of the system. A fault tree models the conditions that need to occur, in term of basic events, in order to cause the occurrence of the top event. Several variants of this basic definition exist; for example, in certain formulations internal nodes are allowed to share a common input for modeling convenience. When using such a notation, a fault tree, from a strictly formal point of view, is not a "tree" anymore.

The quantitative evaluation of a FT consists in the determination of top event probability, based on the probabilities of basic events; the probability of any intermediate event (i.e., other internal nodes of the tree) can also be determined. Probability values can represent different metrics (e.g., reliability, availability) for different applications.

It is important to highlight that a fault tree is not in itself a quantitative model. It is a qualitative model that can be evaluated quantitatively, as it is often done. The use of fault trees is of particular industrial relevance in the development of critical systems; their usage, non only for quantitative evaluation, is standardized as the Fault Tree Analysis (FTA) practice [95].

Reliability Block Diagrams (RBDs) are another popular combinatorial formalism in dependability analysis, due to its resemblance with classical block diagrams describing the physical structure of systems. However, an RBD is a graphical structure which maps the operational dependency of a system on its components, and not the actual physical structure of the system. An RBD is composed of two types of nodes: blocks representing system components, and "dummy" nodes for connections between the components. Edges and dummy nodes model the operational dependency of a system on its components. At any instant of time, if there exists a path in the system from the start dummy node to the end dummy node, then the system is considered operational; otherwise, the system is considered failed. A failed component blocks all the paths on which it appears.

Other graph-based models which can be classified as combinatorial models exist in literature, e.g., Reliability Graphs (RGs) [120], and Attack Trees [177], a variant of fault trees tailored to security analysis.

Markov Chains

When combinatorial models are not sufficient, modelers can employ different kinds of state-space models, which allow more complex relationships between system components to be represented. To this purpose, Markov Chains (MCs) are widely used in different domains, and are also the theoretical basis for the evaluation of more expressive state-based formalisms.

A Markov Chain (MC) [19] is a Markov process with a discrete (or countable) state space. A system can be modeled using a MC if its evolution in time is independent from the past, but only depends on the current state. The set of possible states of a Markov chain is called the state-space, denoted by S. A state change of a Markov chain is called a state transition. More formally, a MC is a stochastic process $\{X(t), t \geqslant 0\}$ with a discrete state space such that for any $n > 0$ and any sequence of increasing time instants $t_1, t_2, \ldots, t_n, t_{n+1}$, the following equation holds:

$$\begin{aligned} P\{X(t_{n+1}) = j \mid X(t_n) = i_n, X(t_{n-1}) = i_{n-1}, \ldots, X(t_1) = i_1\} = \\ = P\{X(t_{n+1}) = j \mid X(t_n) = i_n\}, \quad \forall j, i_n, i_{n-1}, \ldots, i_1 \in S. \end{aligned} \tag{1.1}$$

Equation 1.1 describes the *memoryless* (or Markov) property: the future behavior of the process is independent from its past. Moreover, if the exact characterization of the present state of the process is independent from the current time, then the Markov chain is said to be time-homogeneous, otherwise it is said to be a non-homogeneous Markov chain. The parameter t that indexes the Markov chain can be either discrete or continuous. In the first case we have a Discrete-Time Markov Chain (DTMC), $\{X_n \mid n \geqslant 0\}$, where state transitions only occur at discrete points in time, often called steps, whereas in the latter case we have a Continuous-Time Markov Chain (CTMC), $\{X(t) \mid t \geqslant 0\}$, in which state transitions may occur at any point in time.

The only continuous probability distribution that satisfies the memoryless property is the exponential distribution; therefore, each transition from state i to state j of a CTMC chain occurs in an exponentially distributed time; the rate of the transition is thus exactly the inverse of the mean of the corresponding exponential distribution.

Unfortunately, not all the existing systems and their features can be properly described using Markov processes, since these processes require the strong assumption that the holding time in any state of the system is exponentially distributed. In some cases this assumption may be very unrealistic, and to properly represent the system behaviour more general stochastic processes (e.g., semi-Markov, Markov Regenerative or even non-Markovian processes) must be used.

When dealing with such processes, complex and costly analytical solution techniques may have to be used. If analytic solution methods do not exist at all, discrete-event simulation must be used to solve the models thus providing only estimates of the measures of interest. Alternatively, one can approximate an underlying non-Markovian process with a Markov process; the price to pay following this approach is a significant increase in the number of states of the resulting Markov model, and errors introduced by the approximation.

Petri Nets

Petri Nets (PNs) theory was originally introduced by C. A. Petri in 1962 [157]. A Petri net is a particular kind of directed graph, together with an initial state called the *initial marking*, μ_0. The underlying graph of a Petri net is a directed weighted graph consisting of two kinds of nodes, *places* and *transitions*, where arcs connect either a place to a transition, or a transition to a place. Formally [139], a place-transition Petri net is 5-tuple $PN = (P, T, A, M, \mu_0)$, where:

- $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places.
- $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions.
- $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs connecting elements of P and elements of T, also called flow relation. Arcs going from a place to a transition are

called input arcs, while arcs directed from a transition to a place are called output arcs.

- $M: A \to \mathbb{N}^+$ is the multiplicity associated with arcs in A.
- $\mu_0 : P \to \mathbb{N}$ is the initial marking that denotes the initial number of tokens for each place in P.

The places that are linked to transition t by an input arc are called the *input places* of the transition. Similarly, the places linked to transition t by an output arc are called the *output places* of the transition.

In the graphical representation, places are drawn as circles and transitions are drawn as bars or boxes. Arcs are labeled with their multiplicity (i.e., a positive integer), where arcs with multiplicity k are interpreted as k parallel arcs. A *marking* (corresponding to a *state* of the model) assigns to each place a non-negative integer. If a marking assigns to place p an integer k we say that p is marked with (i.e., contains) k *tokens*, which are graphically represented as black dots inside the place. A given marking μ_i is denoted by a vector of of size n, where $n = |P|$ is the number of places; $\mu_i(p)$ denotes the number of tokens in place p in marking μ_i. The *reachability set* is defined to be the set of all markings reachable by firings of transitions from the initial marking; the *reachability graph* is a directed graph in which nodes are elements in the reachability set, and arcs connect marking μ_i to μ_j if marking μ_j can be reached in one step (i.e., the firing of a single transition) from μ_i.

Transitions model activities which can occur and change the state of the system. Transitions are only allowed to fire if they are enabled, and this happens when there are enough tokens available in the corresponding input places. When the transition fires, it removes from each of its input places a number of tokens equal to the multiplicity of the corresponding input arc, and adds to each of its output places a number of tokens equal to the multiplicity of the corresponding output arc. When two enabled transitions share an input place and the number of tokens therein is not sufficient for both of them to fire, the transitions are said to be in conflict, and a selection rule (usually a priority associated to transitions) must be employed to break the competition in favor of one of them. A system can be modelled by representing its states as markings of the PN: tokens can be used to represent entities of the system, such as tasks to be executed, messages to be sent, while transitions model activities or synchronization constraints of the system, and the firing rules define the pre-conditions to be satisfied for the activities to be executed or the synchronization to be completed, respectively.

The absence of the notion of time in the class of basic "Place/Transition" (P/T) Petri nets does not allow quantitative analysis of the modeled system to be performed. This formalism was mainly introduced to model qualitative aspects of systems (concurrency, parallelism) and to verify its structural prop-

erties (like, the absence of deadlocks, a given order in the firing of transition, or other invariants).

Stochastic Petri Nets

A very popular extension of the place-transition Petri nets formalism is the class of Stochastic Petri Nets (SPNs) [129]. In the classical definition of SPNs, each transition t is associated with a random firing delay following an exponential probability distribution. The enabling rules for transitions are the same as of the PN models. As soon as a transition t gets enabled, a random firing time is sampled from the exponential distribution associated to t, and a timer starts counting from that time down to zero. Transition t fires if and only if it remains continuously enabled until the timer reaches zero. When t fires, the tokens are removed from their input places and added to the output places in a single atomic and instantaneous operation (atomic firing rule).

It is interesting to observe that in the time interval between the enabling and the firing of t, other transitions sharing some input places with t can get enabled and fire without disabling it, provided that there is a sufficient number of tokens in the common input places. On the contrary, in the case of a conflict, the transition whose timer reaches zero first is the one that fires ("race model"). It is important to note that the use of exponential distribution relieves the user from the specification of the behavior of transitions that do not fire after having been enabled, i.e., the specification of an *execution policy* [44], which specifies how the time they spent while enabled should be taken into account in the computation of their *new* firing time. With the "resampling" policy a new firing time is always sampled; with the "enabling memory" policy the elapsed time is taken into account provided that the transition remains enabled; with the "age memory" the elapsed time is always taken into account (i.e., even if it is disabled and then becomes enabled again). Thanks to the memoryless property of the exponential distribution (Equation 1.1), whether the memory of the elapsed time is kept or not, the remaining time is still exponentially distributed with the same rate.

The evolution of a SPN model can be represented by a continuous-time homogeneous Markov chain, whose state space elements are in a one-to-one correspondence with the elements of the reachability set, and whose transitions rates among states are equal to the firing rates of the transitions that produce the corresponding marking change in the SPN. An SPN model can thus be solved in terms of the marking occupation probabilities by performing the analysis of the associated Markov chain.

SPNs Extensions and Other Formalisms

Due to their expressiveness, SPNs are commonly used as a method to describe a Markov process at a higher abstraction. Several extensions have been intro-

duced in literature, adding new primitives to support a more compact specification of the state-space, or allowing the specification of non-Markov processes.

Two of the most well known extensions to the SPNs formalism include Generalized Stochastic Petri Nets (GSPNs), which allow exponentially timed transitions as well as immediate (i.e., firing with zero delay) transitions, and Deterministic and Stochastic Petri Nets (DSPNs), which allow transitions with deterministic firing delays to be specified. Other extensions include Semi-Markovian Stochastic Petri Nets (SMSPN), Timed Petri Nets (TPN), Generalized Timed Petri Nets (GTPN), Markov Regenerative Stochastic Petri Nets (MRSPN). Some formulations define SPNs as the formalism in which timed transitions fire after a random firing delay, following a general probability distribution, and then define the class of SPNs where all distributions are exponential as a particular subclass. A classification of main SPNs variants and their underlying stochastic processess can be found in [44].

Stochastic Reward Nets (SRNs) [47] provide guards, marking-dependent arc multiplicities, priorities, and support the specification of metrics for the analysis directly in the model. Stochastic Activity Networks (SANs) [167] are an even more powerful extension of SPNs; they allow arbitrary probability distributions for the firing time of activities, support the specification of both discrete and continuous state, and introduce the additional *input gate* and *output gate* primitives allowing arbitrary modification on SAN marking to be performed upon firing of a transition (*activity* in SANs terminology). Furthermore, they allow most model parameters to be specified as marking-dependent expressions.

Other modelling formalisms exist that allow a high-level specification of Markov Chain models, e.g., Stochastic Automata Networks [158] or the family of formalisms collectively known as Stochastic Process Algebras [51]. Such formalisms are extensions of basic process algebras, which are augmented with the ability to associate probabilities and/or time delays to the execution of actions, thus allowing quantitative analysis to be performed on the model. Different stochastic process algebras have been introduced, with Performance Evaluation Process Algebra (PEPA) [91] being the most influencing one in dependability and performance analysis. Similarly to extensions to Petri nets, some of these formalisms are Markovian, e.g., PEPA, or MTIPP[1] [88], therefore having evaluation techniques based on the evaluation of the underlying Markov chain. Other formalisms allow the specification of more general probability distributions, e.g., SPADES [83], and thus require more sophisticated numerical techniques or discrete-event simulation.

The work in this thesis focuses on dependability evaluation techniques based on Stochastic Petri Nets and their extensions. More information on stochastic process algebra approaches can be found in [8, 51].

1 Markovian Timed Processes for Performance Evaluation

1.2.2 *Model construction and solution approaches*

The main problem in using state-based models to realistically represent the behavior of a complex system is the explosion in the number of states (often referred to as the "state-space explosion" problem). Significant progress has been made in addressing the challenges raised by the large size of models both in the model construction and model solution phase, using a combination of techniques that can be categorized with respect to their purpose: largeness avoidance and largeness tolerance, see [101, 141] for three comprehensive surveys.

Largeness avoidance techniques try to circumvent the generation of large models using, for example, state truncation methods [34], state lumping techniques [103], hierarchical model solution methods [164], fixed point iterations [116], hybrid models that combine different model types [140] and fluid flow approximation [91, 92]. However, these techniques may not be sufficient as the resulting model may still be large. Thus, largeness tolerance techniques are needed to facilitate the generation and the solution of large state space models.

Largeness tolerance techniques propose new algorithms and/or data structures to reduce the space and time requirements of the model. This is usually achieved through the use of structured model composition approaches, where the basic idea is to build the system model from the composition of sub-models describing system components and their interactions. Efficient processing rules are then defined for the elaboration of the sub-models and their interconnections. Following the approach proposed in [159], for example, the generator matrix of a CTMC is not entirely stored, but it is implicitly represented as Kronecker product of a number of smaller matrices. In [48] largeness is tolerated using Multivalued Decision Diagrams (MDDs) to efficiently explore large state spaces.

Other approaches try to tolerate model largeness using model decomposition and aggregation of the partial results. The basic idea is to decouple the model into simpler and more tractable sub-models, and the measures obtained from the solution of the sub-models are then aggregated to compute those concerning the overall model. A survey on decomposition/aggregation approaches can be found in [114]. In the same work, also extended in [112], the authors also propose a general modelling framework that adopts three different types of decomposition techniques to deal with model complexity: at functional, temporal, and model-level. Largeness tolerance techniques that are applied at implementation-level also exist, such as disk-based approaches [64], where the model structure is stored in the disk, thus allowing larger models to be solved, or "on-the-fly" approaches, [63] which completely avoid the storage of structures in memory, generating them iteratively while computing the solution.

Rather than focusing on model construction, some approaches concentrate on the definition of the measures of interest to be evaluated on the model. In fact, many sophisticated formalisms exist for specifying system behaviors, but methods for specifying performance and dependability metrics have remained quite primitive until recent years. To cope with this problem, modelers often must augment system models with extra state information and events to support particular variables. To address this problem the so-called "path-based reward variables" have been introduced in [142]. More recently, model-checking techniques have been extended to support the specification of quantitative properties, which are evaluated by stochastic model-checking approaches [8]. Since their introduction, stochastic temporal logics like the Continuous Stochastic Logic (CSL) and its extensions [68], have been applied for the purpose of dependability evaluation as well [2, 84].

Even if these techniques are used, solving large state-space models still remains a difficult task. Moreover, under certain conditions model solution may be a challenge even for models having only a few states. In particular, a large difference between the rates of occurrences of events leads to the *stiffness* problem (e.g., see [119]. Stiffness may be avoided using aggregation and decomposition techniques in which the resulting sub-problems are non-stiff (e.g., see [18]), or it may be tolerated using specific numerical solvers (e.g., see [136, 138]).

It is important to note that all the techniques discussed above are complementary and, when evaluating real-life systems, more than just a single technique may be needed at the model construction and model solution levels.

1.2.3 *Modelling and solution tools*

Several software tools have been developed over the years to address dependability and performability modelling and evaluation. Extensive surveys of the problems related to techniques and tools for dependability and performance evaluation can be found for example in [29, 85, 165]. Tools for the evaluation of dependability and performability models are often broadly grouped in two main categories.

Single-formalism/multi-solution tools are built around a single formalism and one or more solution techniques. They are very useful inside a specific domain, but their major limitation is that all parts of a model must be built in the single formalism supported by the tool. Within this category we can distinguish between two main sets of tools, based on the adopted modeling formalism. The first set of tools is based on the Stochastic Petri Nets formalism and its extensions; some examples are DSPNexpress [110], GreatSPN [7], SURF-2 [12], DEEM [20], TimeNET [191]. Other tools are instead based on stochastic process algebras; they provide numerical solutions and in some cases simulation-based results as well. This set includes for example the PEPA Eclipse Plugin

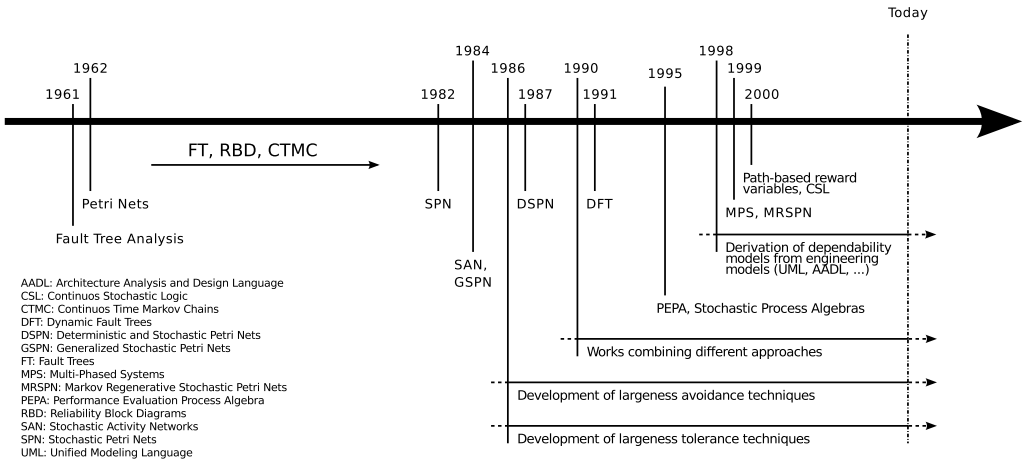


Figure 1.3: Timeline of advances in modelling and model-based assessment [25].

[178], CASPA [106], PEPS [11], and PRISM [107]. All the above tools provide analytic/numerical solvers for a generated state-level representation and, in some cases, support simulation-based solution as well. Other tools use other model specification approaches, sometimes tailored to a particular application domain, e.g., HIMAP [173] and TANGRAM-II [28].

Multi-formalism/multi-solution tools support multiple modelling formalisms, multiple model solution methods, and several ways to combine the models, also expressed in different formalisms. They can be distinguished with respect to the level of integration between formalisms and solution methods they provide. In particular, some tools provide the infrastructure to unify different single-formalism modelling tools into a unique software environment; examples in this category are IDEAS [74], FREUD [135], and DRAWNET++ [73].

Other tools actually implement new formalisms, composition operators and solvers within a unique comprehensive framework. Though more difficult than building a software environment out of existing tools, this approach has the potential to much more closely integrate models expressed in different modelling formalisms. To the best of our knowledge, SHARPE [179], SMART [45], DEDS [10], OsMoSys [184], POEMS [1] and Möbius [55] are the main tools falling in this category.

1.3 SUMMARY AND HISTORICAL VIEW

The previous section has provided an overview of stochastic model-based approaches for dependability and performance evaluation of computing systems, with a focus of SPNs-based approaches. An historical timeline of research in model-based evaluation, as well as current active directions, is sketched in Figure 1.3 and summarized in the following.

The introduction of FTA andm PNs in the early 60's had a great impact on the formalization of model-based assessment practice. For several years, model-based assessment was based on these formalisms, as well as on the earlier theories of MCs and Queuing Networkss (QNs) [19]. Later, PNs and MCs influenced to many other higher-level formalisms (SPNs, GSPNs, SANs, etc.), which are currently widely used for model-based analysis. The largeness and complexity of the models rapidly became a challenging issue to be addressed, and in the beginning of the 80's researchers started focusing on the development of methodologies, techniques and tools to avoid or tolerate model complexity. From the early 90's a direction began to explore the combination of different evaluation approaches (mainly experimental approaches and modeling), with the aim to exploit their synergies and complementarities. In the same years, stochastic extensions to process algebras were being introduced in the literature, most notably with the introduction of PEPA in the mid 90's.

Aside from the continuing development in largeness avoidance and largeness tolerance techniques, two main fertile research lines in model-based evaluation currently focus on i) the integration of model-based evaluation with other techniques, e.g., experimental evaluation; and ii) the automated derivation of analysis models from higher-level architectural descriptions of the system. Actually, in the last 20 years, a major direction has focused on the use of engineering languages (UML, AADL, etc.) to facilitate the construction of the models by designers, and on the development of transformation techniques and tools to translate such high-level models to analysis models for dependability evaluation. The work presented in this thesis falls within this research direction, whose context is introduced in the next chapter.

2

MODERN APPROACHES TO SYSTEM DEVELOPMENT

In this chapter we describe the context in which dependability analysis is cast throughout this dissertation, discussing the problem(s) that we are going to address, and surveying the main related work present in literature.

We first introduce two popular system development methodologies that have gained popularity in recent years, namely Component-Based Development (Section 2.1) and Model-Driven Engineering (Section 2.2), which consitute the basis for our approach. Then, we discuss the role of UML in supporting such methodologies (Section 2.3), and present the main work related to model-driven dependability analysis (Section 2.4). A summary is then provided in Section 2.5, in which we highlight our motivations and we position the work in this thesis with respect to the presented state of the art.

2.1 COMPONENT-BASED DEVELOPMENT

The need for modularity and composition in tackling the complexity of modern systems has emerged in several engineering domains. The one in which this aspect is most evident is perhaps software engineering, where techniques for constructing a software system out of reusable elements had a great resonance [76, 124].

Actually, since the early days of programming, subroutines were invented to conserve memory [52]. Their function was to allow programmers to execute code segments more than once, without having to duplicate that code in each physical location where it was needed. This kind of reuse was aiming at conserving computational resources; later, it was realized that reuse could save "human" resources as well. The principles of separation of concerns [65] and information-hiding [156] contributed to set the basis for modern software reuse, advocating the need to divide the system into parts, such that the whole system was easily changed by replacing any module with one satisfying the same interface.

Still today, software engineering is exploring and growing this paradigm, shifting from objects to components. While there exist slightly different definitions of what a "component" is [27, 176], it is generally agreed that a component should have at least three fundamental characteristics. A component:

i. is an independent and replaceable part of a system that fulfills a clear function;

ii. works in the context of a well-defined architecture;

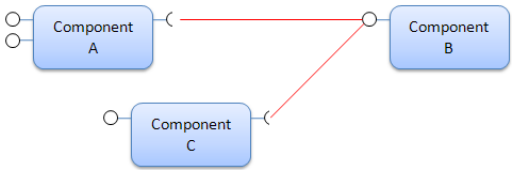


Figure 2.1: Components are assembled through their interfaces.

iii. communicates with other components through its interfaces.

The shift from *objects* to *components* led to a new approach to the definition of system architectures, in which the system is assembled — rather than developed — by connecting pre-existing components through their interfaces (Figure 2.1). Software engineering is therefore focusing to develop methodologies to structure system architectures so that they can be built out of reusable components. Moreover, ways in which components are interconnected are also being studied, so that the simple "subroutine call" approach can be substituted with higher level mechanisms such as event signalling. This includes "wrapping" stand-alone systems in software to make them behave as components, or wrap components to make them behave as stand-alone systems [52]. This approach to software development is known as either Component-Based Development (CBD) or Component-Based Software Engineering (CBSE), with the latter having a stronger emphasis on software.

It might be argued that the component concept is similar to the traditional object concept in the OOP[1] paradigm. However, while there are many analogies between the two, the *component* notion goes much further. Reuse in OOP usually refers to reuse of classes or libraries in a specific programming language, e.g., Java or C#. A component should be reusable without even knowing the programming language or platform that it uses in its internals: the same specification can be implemented in different ways. Also, while CBSE develops Object Oriented Programming (OOP) concepts at a further abstract level, it is acknowledged that OOP is neither sufficient nor necessary for the application of CBSE principles [27]. Finally, the CBSE approach has at its foundation the "buy, don't build" philosophy: assuming that there is sufficient commonality in many large software systems, it should be more convenient for companies to rely on specialized Commercial Off-The-Shelf (COTS) components, rather than building their own solution to the (sub)problem. Such COTS components may then be developed by different organizations, using different languages and platforms.

In order for components to be composed together, they must however conform to a given *component model*, which fixes the kind of information that is associated with components, how components are defined, their structure, as

1 Object Oriented Programming [124]

well as their possible interactions. More precisely, according to [33], a component model defines standards for i) properties that individual components must satisfy, and ii) methods, and possibly mechanisms, for composing components. A more general definition of *component* is then "a software building block that conforms to a component model".

Actually, the CBSE approach has been quickly adopted by the software engineering community, and several commercial component models have emerged; Microsoft's COM/COM+ [126], Oracle's JavaBeans [175], and the OMG's CORBA standard [145] are well-established technologies for the component-based development of software systems.

However, while CBD is a reality in some domains (e.g., enterprise applications), in the development of embedded systems it is not yet a completely accepted practice, and industrial software-developers are still, to a large extent, using monolithic and platform-dependent software technologies [128]. This aspect is even more prominent in safety-critical or mission-critical embedded systems.

Among the factors which are preventing a complete adoption of the CBD in such domains, the unavoidable impact on the development process [56], and resulting costs and risks associated with the adoption of a new development methodology play a key role, but they are not the main reason. In particular, the vast majority of tools for supporting component-based development allow component and services to be specified only from a *functional* point of view, while *non-functional* attributes are not addressed with a comparable maturity. Such non-functional — or extra-functional — attributes, which are mildly important in other domains, are of utmost important in embedded and real-time systems, especially if employed for critical applications. They need therefore to be addressed in the component model, so to be taken into account during all the phases of system design.

Even though there is much more emphasis on software in literature, it should be noted that the CBD notion can be applied to hardware components as well. Actually, another challenge in CBD development of embedded systems concerns with coping with both hardware and software components: their integration is often cumbersome due to their incompatibilities, different specifications and different approaches in their development [109].

2.2 MODEL-DRIVEN ENGINEERING

CBD is not the only initiative that emerged in the last years to facilitate the development of software-intensive systems and improve productivity. Model-Driven Engineering (MDE) [168] refers to the systematic use of models as primary artefacts throughout the engineering lifecycle, focusing on the aspects

of the particular problem to solve, and abstracting away from implementation details.

Software developers have always tried to develop abstractions to help them to focus on concepts related to the problem to be solved, rather than to the underlying computing environment.

Modern programming languages, like Java or C#, and the associated frameworks have successfully raised the abstraction level, and the adoption of domain-specific libraries have minimized the need to re-invent common solutions for problems related to specific application domains. Nevertheless, the complexity problem has not vanished: libraries and development platforms are increasing in complexity, similarly to applications that rely on them; moreover, they should be maintained and upgraded, which requires considerable time and resources. Most importantly, even though such technologies have improved the abstraction level, they are still oriented to computational aspects of the problem: they provide abstractions in the *solution* domain (computing technologies) rather than in the *problem* domain, i.e., in concepts related to the particular application domain of interest [168].

An approach to cope with such complexity is to develop MDE techniques, which combine:

i. Domain Specific Languages (DSLs), which formalize the application structure, behavior and other information required in a particular domain; and

ii. Model-transformations and generators, which analyze specific aspects of a model, and synthesize different kind of *artifacts*, such as source code, simulators, documentation, etc.

The ability to automatically synthetize such artifacts from system models helps ensuring the consistency between system specification, implementation, and analysis models. DSLs are defined by means of *metamodels*; a metamodel defines in an unambigous way the concepts that belong to a certain domain, their structure, and their relationships. A metamodel defines the elements of the modeling language and defines the constraints that a model must fulfill in order to be consdered valid, i.e., it is a "model of models" in that particular language. MDE tools (should) then verify that models respect the structure imposed by their metamodels and any other constraints that may have been specified. By performing such constraints-checking they are able to i) guarantee the correctness of artifacts produced by transformations, or ii) identify many of the problems arising in the design process already from its early phases.

2.3 THE UNIFIED MODELING LANGUAGE

The Unified Modeling Language (UML) is, in general, a central resource in the development of modern software systems; this becomes especially true when

a MDE process is employed. Historically, UML was born as the result of the joint effort of important personalities in object-oriented software development, Grady Booch, Ivar Jacobson and James Rumbaugh, which were working together in the 90's with the aim to fuse their leading design methods (Booch, OOSE, OMT) in a single standard language; the initial version of UML (1.0) was proposed to the Object Management Group (OMG) in 1996, and was officially adopted as a standard in November 1997 [104]. The most recent version currently released by the OMG is UML 2.4.1 [149, 150], which was released in August 2011.

The objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes. UML is a language with a very broad scope that covers a large and diverse set of application domains. Not all of its modeling capabilities are necessarily useful in all domains or applications; for the same reasons it may be difficult to specify precise concepts belonging to the domain of interest in a convenient way. To a certain extent, this limitation can be in part overcome through the *profiling* mechanism (see Section 2.3.2), the extension mechanism provided by the UML standard.

The set of modeling concepts of UML is partitioned into four horizontal layers of increasing capability, called compliance levels, organized in two specifications, "Infrastructure" [149] and "Superstructure" [150]. The UML Infrastructure provides an entry-level modeling capability (layer L0), and represents a common denominator that can serve as a basis for interoperability between different categories of modeling tools. Further modeling capabilities are defined by the UML Superstructure, which includes the specification for layers L1, L2, and L3.

2.3.1 *UML Diagrams*

UML consists of thirteen diagram types, each addressing a different aspect of the system or providing a different way for organizing system concepts. UML diagram types can be categorized into *structural diagrams*, which are used to model the structure of the system, and *behavioral diagram*, which focus on modeling the behavior of the system and its components (Figure 2.2).

Concerning structural diagrams, the most widespread is certainly the *Class Diagram*, which shows the structural entities of the designed system as related classes and interfaces, with their features, constraints and relationships; the *Object Diagram* is an instance-level class diagram, i.e., it shows instance specifications of classes, interfaces, and associations. The *Package Diagram* shows packages and relationships between packages. The *Composite Structure Diagram* is typically used to show the internal structure of a classifier as composed of

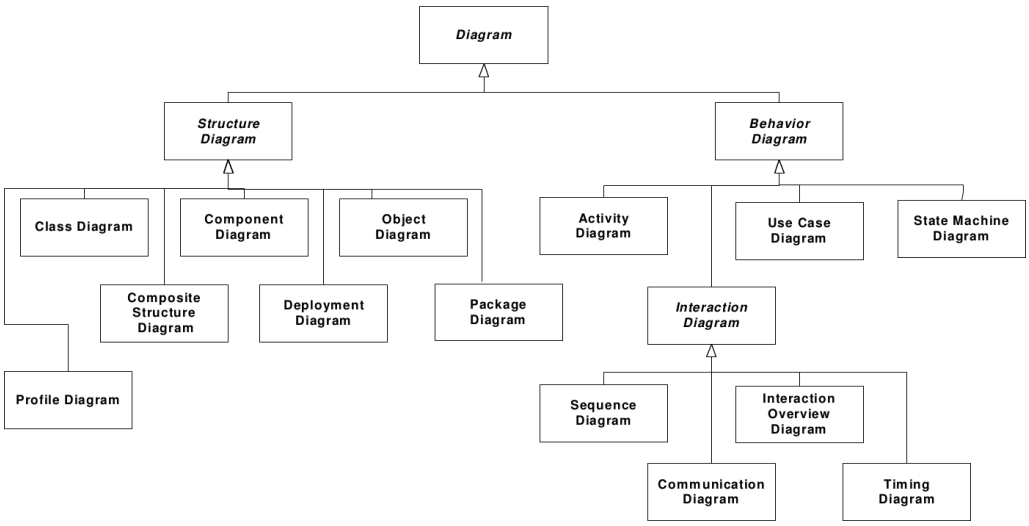


Figure 2.2: UML 2.4.1 diagram types [150].

its properties, parts, and relationships; in some contexts it is also employed to describe — from a structural point of view — the elements that collaborate to produce a certain behavior. The *Component Diagram* shows components and dependencies between them, in a CBD fashion. The *Deployment Diagram* addresses the architecture of the system as deployment of software artifacts to deployment targets; in general, deployment can involve specification-level deployment, which does not references specific instances of artifacts or nodes, or instance-level deployment. The *Profile Diagram* is a particular kind of diagram, similar to the Class Diagram, which is used to define UML *profiles* (see Section 2.3.2).

Concerning behavioral diagrams, the *Use Case Diagram* describes a set of actions that some "subject" (i.e., the system) should or can perform in response to external users, called "actors". The *Activity Diagram* describes a parameterized behavior, represented as a flow of actions (i.e., lower-level behaviors); the diagram is used to describe the control flow, as well as the data flow, between actions of the system. The *State Machine Diagram* is used for modeling discrete behavior through finite state transitions; state machines can express the behavior or usage protocol of a classifier or of parts of the system.

A particular subset of behavioral diagrams are identified as *interaction* diagrams, since they are used to model interactions between system elements. Among them, the *Sequence Diagram* is the most widely used, and describes the interactions that are needed between objects in order to to accomplish a given task; the focus of this diagram is on message interchange and ordering across a number of "lifelines". The *Communication Diagram* is related to the sequence diagram, in that it shows messages exchanged between objects; however, the focus here is not on temporal ordering, but rather on the structure of inter-

actions between objects. The *Timing Diagram* is used with a primary purpose to reason about time; is shows the timeline of changes occurring in individual classifiers or interactions of classifiers along a linear time axis. Finally, the *Interaction Overview Diagram* is a variant of the Activity Diagram, in which nodes are interactions or interactions users; it is therefore used to describe the coordination of different interactions. The lifelines and messages do not appear at this overview level.

2.3.2 *The Profiling Mechanism*

The UML2 specification defines a lightweight mechanism for extending the UML metamodel, called *profiling*. A UML *profile* is an extension of the UML metamodel containing specializations for a specific domain, platform, or purpose; a UML profile is defined through the *Profile Diagram*. Extensions are defined by means of *streotypes* and *meta-attributes*. A stereotype specializes an existing metaclass, by adding meta-attributes and/or constraints to it. Since UML2 a stereotype introduced by the profiling mechanism is considered to all purposes a new metaclass. From a practical point of view, a UML profile is a particular kind of *package*, grouping all the newly introduced extenisons and constraints.

The profiles mechanism however is not a first-class extension mechanism, since it does not allow a complete freedom in modifying the metamodel, but rather it imposes constraints on how metamodel extensions should be performed. Actually, the profiling mechanism does not allow to modify the existing UML metamodel, or to create a new metamodel, and it is not possible to take away any of the existing constraints: the source metamodel is considered as "read-only", and profiles can only extend it.

There are several reasons to extend the existing UML metamodel, e.g., providing a specific terminology for a certain domain; providing a different notation for existing elements; add constraints on the usage of the metamodel; *add information that can be used for model-transformation* or code generation purposes.

While UML is the leading modeling language in software engineering, some of its limitations impair its application for the development of critical and real-time systems, most importantly the limited support for the specification of non-functional properties. To address this problem, the UML profiling mechanism has been widely adopted, leading to a wide range of profiles for different purposes. Among them, the Object Management Group itself has published as OMG standards several UML profiles related to non-functional system properties, e.g., the SPT [152], QoS&FT [151], MARTE [143], and SysML [148] profiles. To a certain extent, the aforementioned profiles address some of the aspects concerning the specification of non-functional properties; further details will be provided in Section 3.4.

Concerning dependability properties however, there are still no standardized solutions, although the awareness of this problem is growing, to the point that OMG has recently published a Request For Information (RFI), and later a Request For Proposal (RFP) on "assuring dependability of consumer devices" [144, 146], i.e., its intent to produce a new standard for such domain.

2.4 MODEL-DRIVEN DEPENDABILITY ANALYSIS

Deriving dependability analysis models from the engineering models that are created during the development process has the advantage that — besides the required model extensions — there is no need to learn and use specific dependability analysis formalisms, and modelling efforts can therefore be saved. This is definitely a benefit, since formalisms employed for model-based evaluation are highly specialized, far from the typical engineering languages that are used to design system architectures, and thus require specific expertise for their application, as well as a higher learning and modeling effort.

Although the interest in the definition of a standard UML profile for dependability properties is quite recent, the idea of automated derivation of dependability models from higher-level engineering models has appeared in literature in different ways, even before the formalization of the MDE methodology. At the same time, the emergence of MDE, and the elaboration of automated model transformation techniques [58] have opened up new ways to integrate model-based evaluation into the development process.

Different approaches for the automated derivation of dependability models have appeared in literature, often using ad-hoc language extensions. Two surveys on this topic can be found in [9] and [15].

Some approaches directly model the detailed dependability-related behavior in the engineering model that describes the system architecture. A good example of this approach is the AADL Error Model Annex [172]: the dependability behavior is described at a very detailed level, which facilitates the mapping to a state-based analysis formalism like GSPNs [163]. As another example, in [153] UML is used to specify information on error propagation and module substitution, which is then mapped to dynamic fault trees.

The opposite approach is to limit the amount of information that is specified at UML level, and to synthetize dependability models by combining information obtained from different diagrams, typically including structural diagrams which describe the system architecture. The work in [61] defines an approach for deriving a fault tree by analyzing the information contained in sequence diagrams, a deployment diagram, and the system's operational profile. Similarly, the authors of [54] define an approach for evaluating system reliability by means of Bayesian formulas, synthetizing information from a set of diagrams annotated with probability values. Such approach has been later extended in

the context of SPT and QoS&FT profiles in [53]. The authors of [79] propose a methodology for risk assessment of UML models. The approach aims at identifying the high-risk components and connectors of the product architecture: first, for each component and connector in software architecture, a dynamic heuristic risk factor is obtained and severity is assessed based on hazard analysis. Then, a Markov model is constructed from he UML model to obtain scenarios risk factors. The work in [21, 117] defined a model-transformation algorithm that generates a GSPN model from a set of UML diagrams in which components and connectors are enriched with dependability attributes. The resulting dependability model is obtained by combining dependability information applied on system components with structural information present in the functional model of the system.

Other approaches focus instead on behavioral UML diagrams, and derive stochastic analysis models by fixing a precise semantics for them and attaching timing information to actions. As an example, the work in [60] relies on guarded statecharts (i.e., state machines) to generate SRN models, which can then be analyzed to obtain quantitative dependabiltiy and performability metrics. The approach presented in [14] and [123] defines a semantics for UML statecharts and sequence diagrams aimed at performance evaluation, the approach focuses on delays and synchronization in the executions of actions, and defines a transformation to derive a performance model based on such semantics. The approach has been later extended [16].

Another set of work put more emphasis on the definition of a language to support dependability or performance properties. The work in [81] define the KLAPER language, an intermediate model focused on supporting performance analysis in a model-driven fashion. The work in [16] focuses on dependability properties and defines the DAM profile, by aggregating information used by different approaches in literature; the DAM profile is discussed in more details in Section 3.4, together with the other main relevant languages Another major contribution to a dependability-oriented language is UMLsec [100], which defines extensions for specifying security-related information in UML software specifications. The UMLsec focus is however in i) evaluating UML specifications for the presence of vulnerabilities using formal semantics, and ii) supporting the generation of code enforcing the security properties specified at UML level. In other terms, UMLsec focuses on *fault prevention* and *fault removal*, rather than *fault forecasting*.

Finally, some approaches use model-driven engineering techniques to integrate various aspects from different models into a global system dependability model. In complex, dynamic distributed systems the dependability model shall be constructed from several engineering models that capture various aspects of the system at different hierarchy levels. Typically the user, application, architecture, and network levels are distinguished. For example, the work in [22, 105] defined an evaluation workflow for large mobile systems, in which the gener-

ation of the dependability model is based on i) the UML workflow model of user activities, ii) the topology model of network connections, constructed automatically from user mobility traces; and iii) the application-service-resource dependency models, also specified with UML.

Transformations from engineering languages other than UML have also been defined, using similar approaches. For example, in the web services domain, the BPEL[2] language is typically used as a source for generating a dependability analysis model, e.g. see [78, 190]. Some of the approaches presented above have also been implemented in supporting tools, e.g., see [94, 115, 162, 186].

2.5 SUMMARY

Although a lot of work has been (and is being) developed on MDE techniques for dependability analysis, most of the approaches have been defined as extensions to a "general" system development process, often leaving the actual process unspecified. Similarly, supporting tools are typically detached from the design environment, and assume to receive as input a model satisfying certain constraints.

While in principle such approach allows not to be bound to specific development methodologies, in practice it introduces a gap between the design of the *functional* system model, its enrichment with dependability information, and the subsequent analysis. This is one of the effects of the complexity and generality of UML, which typically leads system designers to adopt only a subset of the language. Even worse, different teams or companies are likely to adopt different subsets, possibly using different approaches and diagrams to accomplish the same task. This is particularly relevant in CBD, where the language primitives selected for system design depend on the adopted component model, and not vice-versa [155].

As a result, once the functional model of the system has been designed, there is no guarantee that a given UML-based analysis technique can be applied, e.g., the required functional elements might not have been used in the description of the system, and thus cannot be extended with dependability information, or they may have been used for a different purpose with respect to the intended one.

Following this observation, the work in this thesis defines a MDE approach for dependability analysis by taking into account a concrete system development process, and the associated component model. Proper language extensions and model transformations to support automated dependability analysis are defined, and the approach is then realized as an analysis plugin to be integrated in the design environment. From a practical point of view, having the analysis tool capable of direct interactions with the system design environment:

2 Business Process Execution Language [187].

i) ensures and enforces the correctness of the model provided as input to the transformation algorithms, and ii) enables back-annotation of obtained results directly in the design model.

3

SUPPORTING DEPENDABILITY ANALYSIS IN A COMPONENT-BASED FRAMEWORK

In this chapter we describe the process that we followed for enriching a component-based system design framework with support for automated dependability analysis. The first necessary step is the definition of the supporting language. We explicitly target the CHESS component-based framework [35], and we describe the process that has been followed for the definition of dependability extensions to the core CHESS methodology. Such extensions led to the definition of a UML profile, DEP-UML.

The process we adopted consists of the following steps:

i) identification of modeling requirements (Section 3.2);

ii) definition of domain concepts of interest (Section 3.3);

iii) analysis of relevant existing languages (Section 3.4);

iv) definition of extensions for supporting dependability analysis in the target framework (Section 3.5).

These steps are detailed in the following sections. Additionaly, Section 3.1 introduces the peculiarities of system design methodology advocated by CHESS.

3.1 THE CHESS METHODOLOGY

This section introduces the CHESS project and methodology, detailing its objectives and highlighting its peculiarities with respect to the system design process; a proper introduction to such context is necessary for understanding some of the choices in the definition of our approach.

3.1.1 *Project Overview*

The ARTEMIS-JU "CHESS" project [35] aimed at developing, applying and assessing an industrial-quality MDE infrastructure that permits high-integrity embedded systems to be assembled in a component-based fashion, while retaining guarantees in terms of functional and non-functional properties. One of the objectives of the project was to define a cross-domain methodology, suitable for (but not limited to) the automotive, railway, and aerospace application domains.

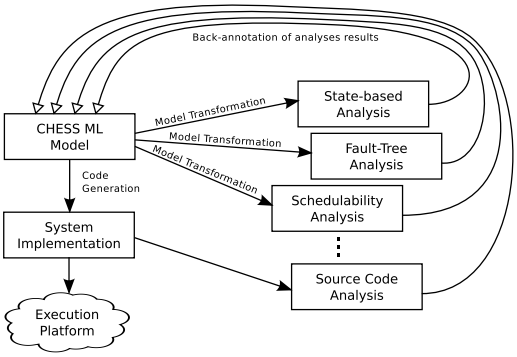


Figure 3.1: The CHESS workflow for system development and analysis.

In the CHESS approach, the development process is supported by different kinds of analyses which allow the feasibility of the system's design to be assessed from different points of view. Such analyses include static code analyses, schedulability analyses, and dependability analyses. In accordance with MDE principles, the analysis models should be automatically derived from the high-level model that describes the system's architecture (Figure 3.1).

The CHESS approach promotes an iterative and incremental development process, in which the system's model is constantly updated and refined based on the results obtained by the different analysis techniques. Analysis results are used to enrich the initial CHESS model from which the analysis has been triggered, in a process usually called "back-annotation" [50, 86], and can be used as input for subsequent analyses. The source code, possibly including legacy code, can be analyzed as well using code analysis techniques (e.g., call-graph analysis).

Concretely, the project aimed to: i) define a distinctive design and development methodology, ii) define an ADL capable of supporting it, and iii) concretely implement the methodology in an industrial-quality framework.

3.1.2 *Methodology Overview*

The CHESS philosophy refers to a particular MDE initiative, the Model-Driven Architecture (MDA) proposed by the OMG [147]. In the MDA workflow, the system designer creates a Platform Independent Model (PIM), which is independent from the execution platform that will actually implement the system. From the By coupling the PIM model with deployment information, a Platform Specific Model (PSM) is then generated by automated transformations. From the PSM, code generation may be triggered to obtain an implementation of the designed system for a given execution platform.

Accordingly, the CHESS user constructs a PIM, specifying a component-based solution to his problem that is not dependent on a specific implemen-

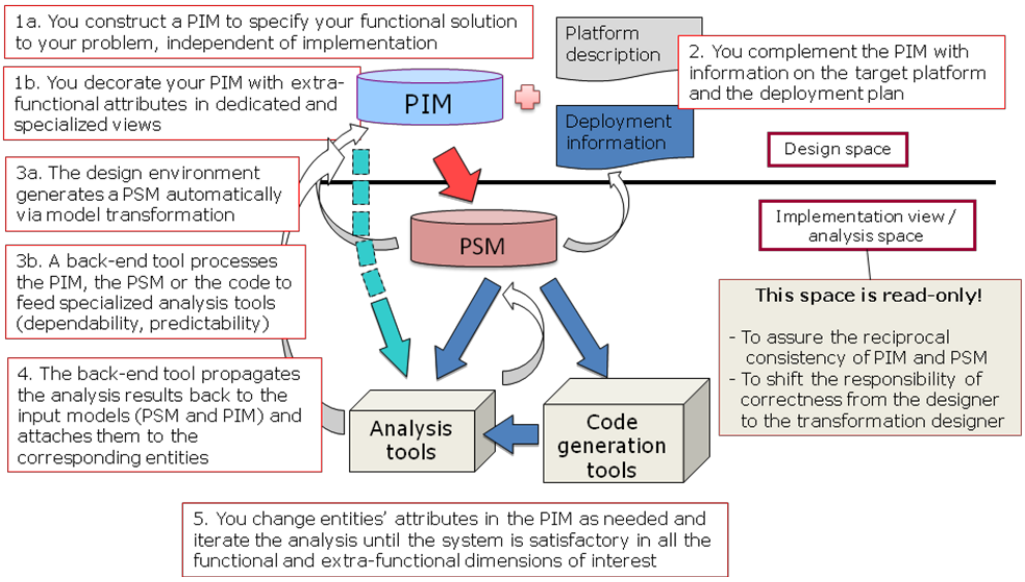


Figure 3.2: CHESS high-level design process [40].

tation. This platform independent model is then decorated with all the needed extra-functional attributes related to dependability and predictability. When the user has provided a description of the target execution platform and allocated the PIM components on it, the PIM is then automatically transformed to a Platform Specific Model (PSM). In the CHESS workflow the PSM is read only: users are not allowed to directly modify it [40].

The CHESS development environment allows the user, through a range of analysis and code generation tools, to validate the feasibility of the solution. These analyses can be performed either on the PIM, the PSM, or the generated code (Figure 3.2). Analysis tools propagate their specific results back to the PIM; such results are then used by the user to refine his PIM and its attached extra-functional attributes in order to meet his purposes. Once the user is satisfied with the application design (i.e., when it satisfies all the desired requirements), code generation tools allow him to generate source code for the desired target platform.

In adopting such workflow, CHESS defines its own methodology: a component-oriented design process, centered around the "Correctness-by-Construction" and "Separation of Concerns" concepts [36].

Correctness-by-Construction

Correctness by Construction (C-by-C) [32] is a software production method that fosters the early detection and removal of development error to build safer, cheaper and more reliable software. The practices engaged in the realization of these goals include: i) the use of formal and precise tools and notations for

any product of the development cycle, whether document or code; ii) the use of tools to verify the product of each development step; iii) the conscious effort to say things only once so as to avoid contradictions and repetitions; iv) the conscious effort to design software that is easy to verify, by e.g., using safer language subsets or explicit coding and design patterns.

In a model-driven workflow, two alternate solutions are possible in the design space for ensuring model correctness [36]:

i. To allow the user the largest freedom of expressive power in the software specification and modeling and then verifying a-posteriori the correctness of the model against some correctness criteria.

ii. To restrict the expressive power of the user by propagating to the design space the applicable domain-specific constraints, and enforcing them actively so that the resulting model is correct by construction against the domain criteria.

In the former approach, the design space is intentionally void of any domain-specific semantics or constraints and the user's meaning can be expressed with full freedom: the domain-specific aspects are introduced at a later point. A model specified in this manner may of course fail some a-posteriori checks and thus incur the need for possibly escalating modifications, whose repercussions are difficult to judge in effort, time and cost.

In the latter approach instead, the design infrastructure supports the desired semantics and constraints directly in the user's design space. In that manner we have a-priori guarantees that the user model is correct against those constraints; this also means that it can safely be used as input for the subsequent MDE activities. This second alternative is the one selected (and enforced) by the CHESS design methodology.

Separation of Concerns

Separation of concerns is a concept first advocated by Dijkstra in [65], and involves the clean separation of different aspects of software design. As a major advantage, it enables separate reasoning and focused specification. The CHESS methodology especially seeks rigid separation between functional and extra-functional concerns.

IEEE Std-1471 [99], also known as ISO/IEC 42010:2007, describe recommended practices for the architectural description of software-intensive systems. The document prescribes that the "architectural description of the system is organized into one or more constituents called views", where a view is a partial representation of a system from a particular viewpoint, which is the expression of some stakeholders' concerns.

Currently, only the SysML language [148] provides direct support for user-defined viewpoints and views [36]. The CHESS methodology moves a step

further, by defining a modeling environment that allocates distinct concerns to distinct views. Modeling concerns, and therefore views, are first separated out in the two main categories: *functional* and *extra-functional*. The extra-functional view is then further subdivided in: *deployment*, where the system configuration is specified in terms of physical (i.e., hardware) modeling as well as of software apportionment and allocation to it; *predictability*, where real-time attributes and requirements are specified; and *dependability*, where attributes and requirements related to safety and dependability are specified.

In order to make a view-based development effective, it is important that all the supported PIM views be fully consistent with one another for all model elements that may appear in multiple views. In general, two alternate strategies exist to meet this goal [36]:

i. A synthetic approach, in which each view is modeled separately with one or more models, and is later composed with the other views;

ii. A projective approach, in which the information that pertains to individual views is extracted from a single underlying model that describes the entire system.

CHESS adopts the second option. In keeping with the principle of separation of concerns, CHESS views do not incur overlaps of responsibility for modification: while multiple views can have read rights over cross-cutting aspects, only a single view can have create/write rights on them.

Moreover, views in CHESS are not limited to being a way to master complexity, but they are also a way to enforce correctness by construction: the user is prevented from creating or modifying elements in a given diagram depending on the current view; editor-level restrictions include for example palette features and property editors.

3.1.3 *System Design in CHESS*

The CHESS methodology organizes the system design (and development) process in a series of precise steps, which guide and constrain the user in a proper component-based workflow. The main steps are outlined in the following [36, 38, 154].

1: *Interfaces.* An interface is a set of operations and interface attributes. Operations are defined with a signature, determined by an operation name and an ordered set of parameters, each one with a direction (in, in out, out) and a parameter type An interface can also contain the declaration of one or more interface attributes. An interface shall include at least one operation or one interface attribute; multiple operations can be grouped in the same interface.

2: *Component types.* The component type is the design entity that forms the basis for a *reusable software element*. A designer specifies component types, in isolation, with no relationship with other components. The component type specifies a list of *provided* interfaces and *required* interfaces, (i.e., the services which have to be provided to/by other components), by referencing already-defined interfaces. The component type may additionally define a set of component type attributes, which are typed parameters with a private visibility (visible only by the operations of the component).

3: *Component implementations.* The designer proceeds in the design by creating a component implementation from a component type. A component implementation is a concrete realization of a component type, and its realization can be delegated to a software supplier. A component type may have several implementations, which may differ for example for resource usage, accuracy in the solution, etc., or simply for the adopted programming language.

   A component implementation *must* implement all the functional services of its component type. The code included in a component implementation is purely sequential code and shall be void of any tasking or timing constructs. Despite the sequential nature of the code, an implementation may set specific non-functional constraints to preserve the functional correctness of its behaviour. For example, a control law may work correctly only if executed within a range of frequencies. Technical budgets on the execution time or memory footprint can be placed either on operations or on the whole component.

4: *Component instances and component bindings.* A component instance is instantiated from a component implementation. A component instance is the design entity that is subject to composition with other components to fullfill their functional needs, and it is the deployment unit of the approach. From the functional point of view, there are no differences between a component implementation and an instance derived from it. However, instances of the same component implementation may be decorated with different non-functional attributes. Components connections are defined by the user at design time, by matching the required and provided interfaces of component instances.

5: *Decoration with non-functional attributes.* After the functional model of software entities has been completely specified, the designer can initiate its decoration with non-functional attributes. At this stage, the designer can specify dependability attributes, timing and synchronization attributes, as well other non-functional attributes.

6: *Hardware topology and target platforms.*

In parallel with the software architecture definition, the CHESS development process requires the user to provide a description of the target execution platform; the hardware model should be limited to the elements that are relevant for code generation or analysis purposes. The execution platform is modeled in CHESS through components, following a process similar to the one adopted for the definition of the software system: the user first define the hardware components, whose instances are then connected together to build the hardware architecture.

7: *Component instance deployment.* Once the hardware topology has been defined, the last step to perform in the design space is instance deployment, which includes the allocation of component instances to processing units, and possibly the allocation of component bindings to physical interconnections. This additional information is used for code-generation purposes, but can be used also to refine analysis results, e.g., the computation of WCETs[1].

8: *Model-based analysis.* Once the model has been fully annotated with the non-functional properties of interest, static model-based analysis can be performed to assess the feasibility of current designs with respect to different dimensions, including dependability properties.

The extraction of information from the user model, the generation of the input for the analysis tools, and the application of the analysis techniques shall be automatic. Moreover, the results of the analysis shall be propagated back to the design model, so that the designer can use them as input to implement the required design modifications. The analyses can be iterated at will, with different iterations resulting from a simple change of attributes, until the properties of interest are judged to be satisfactory by the designer.

3.1.4 *CHESS ML and the CHESS Editor*

Practical support to the CHESS methodology is provided by the "CHESS Modeling Language" (CHESS ML). The core CHESS ML language has been defined as a *collection-extension of subsets* of standard OMG languages: UML, MARTE, and SysML, and it is implemented as an UML2 profile [37].

CHESS ML contains specific features to support the CHESS methodology. As an example, it defines the «`ComponentType`» and «`ComponentImplementation`» stereotypes (both extending the UML `Component` metaclass), which are used to represent the respective entities defined in the methodology. To further enforce the defined design workflow, elements identified as «`ComponentType`» cannot own any behavior, since they should not address implementation concerns.

1 Worst-Case Execution Times

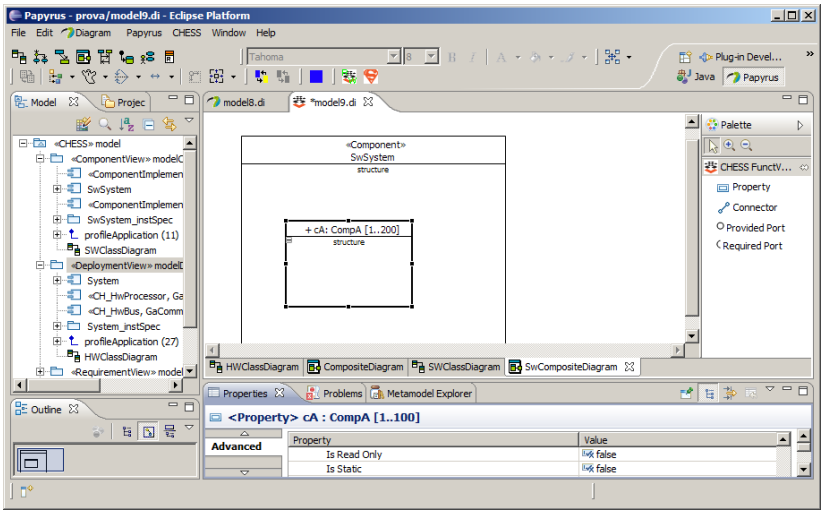


Figure 3.3: Support for views in the CHESS editor [38].

The allocation of software component instances to the hardware platform is performed by importing the `Assign` stereotype from `MARTE`, which applies to `UML::Comment` elements. The "view" concept is also addressed by the language, and it is implemented by stereotyping `UML::Package` elements; for example, the «`DeploymentView`» stereotype is used to define the *Deployment* view.

The CHESS ML language provides support for modeling properties at instance level: in a CHESS ML model, component instances can be explicitly represented using `UML::InstanceSpecification` elements, and can be decorated with non-functional properties. A specific stereotype, «`CHGaResourcesPlatform`» extends the «`MARTE::GaResourcesPlatform`», allowing its application to `InstanceSpecification` elements. In this way, non-functional analyses can be performed at instance level as well.

The overall CHESS methodology is implemented as a set of plugins for the Eclipse platform [80], which are publicly available for download [35]. The CHESS toolset includes a diagram editor based on a customized version of Papyrus [82], an EMF-based graphical model editor focused on UML. The CHESS editor supports different design views (Figure 3.3), enforcing a strict the separation between them: *RequirementsView*, for modeling system requirements; *FunctionalView*, for modeling functional properties; *ExtraFunctionalView*, for modeling extra-functional software properties; *DeploymentView*, for modeling the hardware platform and allocation; *DependabilityView*, for modeling dependability properties of the hardware platform; *AnalysisView*, for modeling analysis-specific concerns.

The CHESS methodology aims at properly supporting the representation of non-functional properties at instance level in the same model, i.e., the ability to attach non-functional properties to `InstanceSpecification` elements. How-

ever, since a hierarchical description of instances is not properly supported at diagram level, the hierarchical structure of composite components is modeled in CHESS using the *parts* concept, i.e., defining subcomponents using `Property` elements. Taking `Property` entities as representation of instances is particularly useful for the modeler, since it allows the usage of the *Composite Structure Diagram* (CSD), which is a very common diagram in UML, and it is therefore well supported by current modeling tools. On the other hand, the CSD has strong limitations with respect to the modeling of instances, since parts are instances modeled in the context of a given component "A" [150]: when creating an instance out of A we can't provide different values for properties of A internals parts (i.e., properties of subcomponents).

Due to such UML limitation about the entities available in the CSD, in order to properly apply different non-functional properties at instance level, the information modeled in CSDs needs to be mapped to UML `InstanceSpecification` elements. The CHESS toolset implements this feature (through the "Build Instances" command), allowing to automatically derive a set of `InstanceSpecification` elements from a Composite Structure Diagram (CSD). Each `Property` and `Connector` element is mapped into a dedicate `InstanceSpecification`, while each `Port` is mapped into a `Slot`. The collection of generated instances is what should be taken into account by model-transformation algorithms. In this way, different non-functional properties can be applied to different component instances.

3.2 DEPENDABILITY MODELING REQUIREMENTS

The goal of this section is to define the dependability aspects that the CHESS ML language should be able to express. Multiple information sources have been analyzed, including available languages and extensions, explicit requirements from industrial partners of the CHESS project, and experiences from past projects starting from the HIDE project [21, 89]. Such analysis helped us to identify a set of nine requirements, grouped in four categories (Table 3.1). Detailed requirements from CHESS industrial partners have shown to be a subset of these abstract requirements and have been successfully mapped to them [39].

Requirement `SA01` states that the language should allow the system to be represented as a collection of interconnected atomic and/or composite components allowing for hierarchical structures. Hierarchical modeling is a mandatory feature for dependability analysis of complex systems: it allows designers to adopt the right level of abstraction for different purposes and promotes an incremental design methodology.

Components can be classified in various ways: based on their nature (software or hardware); based on their structure (atomic or composite); based on their behavior with respect to error propagation (stateful or stateless). This

Table 3.1: Identified requirements to support dependability analysis.

	Definition of the System Architecture
SA01	Need to model the structure of the system as a composition of subsystems/components.
SA02	Need to define different kind of components.
SA03	Need to define the common types of fault tolerant structures and the role that each component plays in such structures.
SA04	Need to model dependency relations between components of the system.
	Definition of the Fault/Error/Failure Characteristics
FEF01	Need to describe the different type of faults, errors, and failures that may affect system components.
FEF02	Need to define specific qualitative and quantitative dependability properties to qualify components.
	Definition of Maintenance Characteristics
M01	Need to represent different maintenance policies and activities.
	Definition of Requirements and Measures
A01	Need to define the metrics of interest to perform dependability and safety analysis.
A02	Need to define system dependability and safety requirements.

classification is fundamental to correctly represent system components from a dependability perspective (requirement SA02). Stateful components (i.e., components that exhibit an internal state), for instance, may be subject to latent errors; the distinction between hardware and software components determines the kind of threats that may affect the component [6].

Fault-tolerant structures are commonly used to improve the dependability of safety-critical or high-available systems and must be carefully represented in the system model. A language for the specification of dependability properties should allow the designer to describe fault-tolerant structures and the role that each component plays within the structure (requirement SA03). One typical task in the design of high-integrity systems is to analyze the effectiveness of redundancy mechanisms in the system design, possibly comparing different alternatives. In this perspective, such modeling capabilities are of fundamental importance: a simple way to represent fault-tolerant structures and a seamless integration with the functional modeling facilitates the comparison of different designs.

Requirement SA04 states that the language should be able to represent dependency relations that may exist between components of the system. From the perspective of dependability analysis, a dependency relation between two components induces an error propagation path between them. For instance, the failure of a computing hardware resource will propagate to the software components allocated on it.

The language should allow designers to describe the different threats that may affect the system, both from a qualitative and quantitative point of view (requirement FEF01). Such kind of description is usually required by certification authorities in the form of a Failure Mode, Effects, and Criticality Analysis (FMECA) [96] and can be also used for other kind of quantitative and qualitative analyses. In component-based system development the detailed information on the internal behavior of individual components however might not be available ("black-box" components); therefore, in addition to detailed threats specification, it should also be possible to represent component-level dependability properties, e.g., the mean time required to repair a given component or its fault occurrence rate (requirement FEF02).

Requirement M01 addresses maintenance: high-integrity systems are subject to precise maintenance policies and activities during their lifespan, in order to guarantee an adequate dependability level. Maintenance activities are typically classified as preventive or corrective, based on whether they are executed based on some predetermined schedule, or only when specific errors are detected. The language should allow the modeler to describe the different maintenance activities that need to be applied to system components.

Requirements A01 and A02 take into account the specification of dependability-related requirements and constraints, as well as the definition of dependability metrics to be analyzed by automated analysis techniques. These two requirements reflect the two possible uses of a dependability profile in the design of high-integrity systems: on one hand it supports tracking dependability requirements and constraints during system design; on the other hand it supports the assessment of specific dependability properties with different analysis techniques. The two aspects are of course closely related.

3.3 CONCEPTUAL MODEL

Based on the requirements identified and described in the previous section, we defined a *conceptual model* that collects the domain concepts of interest. A preliminary version of the conceptual model was first presented in [134]. The conceptual model described in this section guided the definition of dependability extensions to the CHESS ML language.

The goal in the definition of a conceptual model is to identify at abstract level the main concepts that belong to the domain of interest, together with their

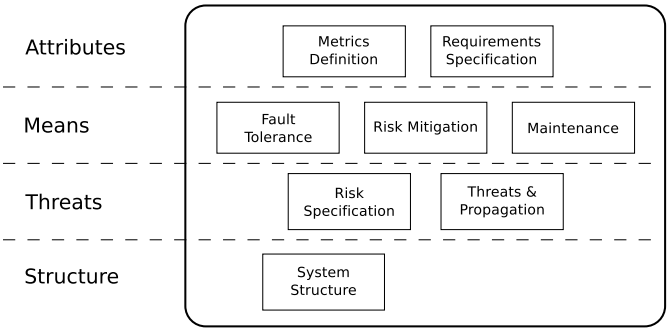


Figure 3.4: Packages in the defined conceptual model for dependability properties.

relationships. The conceptual model is explicitly chosen to be independent of design or implementation concerns; at this stage, how the elements that have been identified will be represented at UML level has not been addressed yet.

The conceptual model is organized in four abstraction layers:

STRUCTURE This level identifies the basic components of the system and their dependability and safety properties.

THREATS This level concerns with the identification of the threats that may affect the system and its dependability and safety properties.

MEANS At this level the means to attain dependability are identified and described. From a general perspective, these are solutions and countermeasures which are developed to deal with the threats identified in the above level and to reach the dependability and/or safety requirements.

ATTRIBUTES This level of abstraction addresses attributes of dependability which are of interest. This includes both the identification of dependability requirements, as well as the specification of metrics that should be evaluated by dependability analysis techniques

Such organization reflects the steps that are undertaken in the construction of the analysis model: i) the definition of the system *Structure*; ii) the identification of *Threats* affecting system components; iii) the definition of *Means* that are adopted to avoid threats to affect the provided services, and finally, iv) the definition of dependability *Attributes* that are of interest. Each of these four levels is populated by different packages, which address the concerns related to that specific level in different ways (Figure 3.4).

It should be noted that the identified packages cover different aspects of dependability analysis, therefore it is expected that only a subset of them are used in different contexts. The actual set of packages to be used in the modeling process is determined by several factors, including the characteristics of the system to be analyzed, the objectives of analyses, as well as the analysis technique. The

role of each package, and the main concepts that it addresses are described in the following.

3.3.1 *Layer 1: Structure*

System Structure

This package addresses the description of the system structure with respect to dependability analysis (requirement SA01); this involves the definition of system components and their classification (requirements SA02 and FEF02), and the basic relations between them (requirement SA04).

The main concepts that are addressed by this package are the following:

- *Component*. Components are the basic blocks of the system with respect to dependability analysis. From the dependability point of view, a component may be affected by faults, errors, and failures.
- *Components classification*. Different kinds of components may exist within the system, having different properties and/or behavior with respect to dependability. Classifying them across different dimensions add useful information with respect to dependability analysis. For example, *hardware* and *software* components are subject to different kind of faults [6], and *stateful* and *stateless* components exhibit a different behavior with respect to error propagation.
- *Use relations*. Components are connected through their interfaces, and they interact in order to implement the system's function. Components may require services to perform their function (i.e., they are "clients"), or they may provide services to the other components (i.e., they are "servers"). When a component uses the functionality provided by another component, a potential error propagation path is established.
- *Composition relations*. Components may be composed of subcomponents, with multiple levels of depth. With respect to dependability analysis composition relations affect how failures of subcomponents propagate to the higher-level component.

3.3.2 *Layer 2: Threats*

Threats & Propagation

This package addresses the definition of different types of fault, errors and failures (FEF01), and the details of error propagation between components (SA04). The conceptual elements that are addressed by this package are the following:

- *Fault*. A fault that may affect a component of the system, possibly generating errors and/or failures. We distinguish two kinds of faults. *Internal faults* develop inside components with a certain occurrence rate or delay, *external faults* are generated by external causes and they may originate from failures of other components.

- *Error*. An error is a deviation from correct system's state [6]. Different errors may develop inside a component as a result of propagation or activation of latent faults. If compensation occurs, after a certain amount of time errors may disappear from the internal state of the component. For example, an erroneous value stored in a memory cell gets compensated if it is overwritten by a correct value, before the wrong one is actually used by other components.

- *Failure mode*. When an error reaches the service interface of the component, a failure occurs. The failure of the component may take different forms, called failure modes. Different failure modes may affect a component, as result of different error propagation paths. Failure modes are characterized by their domain, detectability, consistency, and consequences [6]. Failures may propagate in different ways to other components depending on their failure mode.

- *Propagation path*. This concept addresses the identification of propagation paths that exists in the system. Propagation paths can be characterized by additional properties (e.g., delay and/or probability of occurrence).

Risk Specification

This packages addresses the identification and documentation of unwanted accidents (FEF01): it is concerned with describing the vulnerabilities and risks associated with threats to the system. Specific concepts taken into account by this packaged are:

- *Asset*. An asset is a resource of value requiring protection. An asset can be tangible (e.g., a component, or a specific feature) or intangible (e.g., reputation, knowledge). Identifying and prioritizing a system's critical assets is a vital first step in the process to of identifying mitigation measures to improve its level of protection.

- *Threat*. A potential cause of an undesired event which may result in harm to system assets. A threat may refer to a fault, error, or failure of the "Threats & Propagation" package, but may also be an exceptional event, not specifically related to any system component (e.g., natural disasters).

- *Hazard*. A state of the system that, if other specific conditions occur in the system or the environment, will inevitably lead to an accident. An

instance of the hazard concept may refer to an instance of the failure concept in the "Threats & Propagation" package.

- *Consequence*. The consequence on assets of the occurrence of the occurrence of an accident. Classification of consequence depends on different factors, including the application domain.
- *Risk*. The risk associated the occurrence of an accident. This is typically obtained as a function of its expected frequency and its consequences (e.g., see [31]).

3.3.3 *Layer 3: Means*

Fault Tolerance

This package concerns with the specification of how faults and errors are handled by the system. This aspect includes the description of redundancy structures (requirement SA03), and the description of error detection mechanisms (requirement FEF01). The conceptual elements that are addressed by this package are the following:

- *Fault tolerance structure*. This concept identifies components which are implemented as a fault tolerant structure
- *Redundancy manager*. This conceptual element is used to specify that a component in the system has the role of redundancy manager of a fault tolerant structure. The redundancy manager is the interface of the structure to the other components and it is characterized by the redundancy scheme that it implements.
- *Adjudicator*. This element is used to specify that a component in the system has the role of adjudicator in a fault tolerant structure. The adjudicator checks the correctness of variants operation.
- *Variant*. This element is used to specify that a component in the system has the role of a variant in a fault tolerant structure.
- *Detection activity*. Error detection activities are used to detect the presence of errors in stateful components. Such activities are characterized by their coverage and false alarm ratio. Coverage is the probability to detect an error, given that it is present; false alarm ratio is the probability to detect an error, given that it is not present. Additional attributes should specify when the activity should be executed, its duration, which components are tested and the kind of errors that the activity can detect.

- *Performer of activity*. Maintenance and error detection activities may be performed by other components of the system (e.g., online tests). In such cases, relations must exist in the dependability model between the detection activity and the component which is in charge of executing it. With respect to dependability analysis, such relations allows taking into account the effect of not being able to perform the activity because of the failure the component that is in charge of executing it.

Maintenance

This package addresses maintenance-related properties and policies (requirement M01). Maintenance can be preventive or corrective; the need for corrective maintenance is determined by monitoring activities which are performed on system components. The conceptual elements that are addressed by this package are the following:

- *Repair activity*. Components may be repaired during the lifetime of the system. Repair activities may be planned or they may be performed when some event occurs in the system (e.g., the failure of a component). It is possible that repair activities do not always complete successfully, but instead that they are successful with a given probability.

- *Replace activity*. Components may be replaced during the lifetime of the system. Often replace can be seen as a special case of repair. However, it is possible that a component is replaced with a different (functionally interchangeable) component.

- *Overhaul activity*. Overhaul is a maintenance activity that has as its primary objective to prevent components to reach an age in which their dependability attributes and/or performance are degraded by frequent failures. In components having an increasing failure rate, overhaul tries to keep the failure rate below a certain threshold.

- *Performer of maintenance activity*. Maintenance activities may be performed by other components of the system. This may be the case, for example, of automated recovery activities (e.g., restart of a software application). In such cases, for the same reason of detection activities, relations must exist in the dependability model between the maintenance activity and the component which is in charge of executing it.

Risk Mitigation

This package concerns with the specification of means to mitigate the risks identified in the previous layer. A mitigation of failure is any system means that allows the detection, propagation control, avoidance and/or mitigation of

the effects of undesired events. Risk mitigation spans very different techniques, which are often related to the domain of interest or industrial practices of a specific company; they are therefore not listed here.

Some techniques include the definition of fault-tolerant structures and/or maintenance policies, and therefore a mitigation may refer to concepts defined in the "Fault Tolerance" and "Maintenance" packages. However, other techniques adopt different approaches; thus the need for a a separate package. As an example, some events cannot be avoided because the root cause occurs outside the scope of the system (e.g., earthquakes). The only possible mitigation is therefore a limitation of the resulting consequences, for example by operational protocols or specific prescriptions.

This kind of information is typically used for traceability of requirements (requirement A02) or qualitative dependability analysis such as FMECA.

3.3.4 *Layer 4: Attributes*

Metrics Definition

This package concerns with the definition of analysis objectives for dependability analysis (requirement A01). The main concepts that are addressed by this package are the following:

- *Dependability metric*. This concept identifies a dependability metric. A dependability metric is characterized by its name, which identifies the kind of measure that should be evaluated, and by a reference to the type of evaluation that should be performed for that measure.

- *Evaluation type*. This concept identifies the type of evaluation that should be performed for a dependability metric (e.g., instant of time, steady-state). Additional parameters concerning the kind of evaluation (e.g., the time point(s) in case of transient analysis) are addressed by this element.

- *Target component*. This is a relation connecting a dependability metric to the target component, i.e., the component whose dependability properties have to be evaluated.

- *Target failure*. For a fine-grained specification of dependability metrics, it should be possible to connect a dependability measure to some specific failure mode of a system component defined at the "Threats" level. This relation allows the modeler to define metrics that consider only a subset of the failure modes of a system component. This facility can be useful for example when components have multiple failure modes with different consequences.

Requirements Specification

This package concerns with the specification of dependability requirements (requirement A02). The concepts included in this package are the following:

- *Requirement*. A dependability-related requirement on the system or its components. A requirement contains a textual description, an may refer to a dependability metric defined in the "Metrics Specification" package.
- *Parent requirement*. This relation addresses traceability of requirements decomposition, by connecting a lower-level requirements with a higher-level requirement.
- *Requirement affects*. This relation can be used to connect a requirement with the system component that is involved. It should be noted that is not always possible to identify a single component that is involved in satisfying a requirement: higher-level requirements may involve the entire system or a set of components.

3.4 INVESTIGATION OF EXISTING LANGUAGES

As introduced in Section 2.2, different languages proposed in literature and/or established by industrial practice provide some means to address non-functional aspects of system architectures.

After having identified the proper language requirements, and defined our conceptual model, we analyzed the main relevant languages in literature, and evaluated to which extent they fulfill the identified requirements. The degree to which such requirements are satisfied is evaluated considering not only the offered modeling features, but also: i) the offered modeling convenience, and ii) the extent to which they can be adapted to the target CHESS methodology. Concerning i), in order to be effective in the development of real systems, the resulting language should implement the support for requirements in a convenient way for the modeler; concerning ii), the language should not contrast with the "correctness-by-construction" and "separation of concerns" approaches as intended by the CHESS methodology.

It should be noted that requirement SA01 is the main purpose of every ADL, and it is therefore addressed to some extent by all the languages that have been considered.

3.4.1 *QoS&FT*

The OMG QoS&FT profile [151] is an extension of the UML language that takes into account Quality of Service (QoS) contracts and fault tolerance for software architectures. The main contribution of this profile is the definition of a general

framework for QoS specification. It addresses requirements A01 and FEF02, since it allows users to specify QoS contracts on component interfaces and constraints on the execution of operations.

Relations between system components (SA04) are defined by means of end-to-end constraints affecting interactions between components. A "QoS Catalog" includes specific dependability concepts such as "fault" and "failure"; however some important properties like their occurrence rate are missing. Part of the profile is devoted to the definition of fault-tolerant software solutions; however the profile uses a descriptive approach, making it difficult to use the resulting information for the automated derivation of analysis models. Also, hardware solutions are not taken into account. Requirement SA03 is therefore only partially satisfied.

Finally, the profile uses a complex and heavy-weight annotation process, which has so far limited its practical usage. Indeed, it has been shown that adopting the QoS&FT profile for modeling real systems requires a lot of effort for the final user [17].

3.4.2 *MARTE*

The OMG MARTE profile [143] is one of the most widely adopted languages for modeling non-functional properties of system architectures, and includes also most of the concepts that were introduced in the earlier Schedulability, Performance, and Time (SPT) profile [152] Although it does not take into account dependability properties, some of its features allow it to partially fulfill the identified requirements and they are therefore discussed in the following.

Its "Generic Component Model" (GCM) refines the UML2 structured class modeling concept, while the "Hardware Resource Modeling" (HRM) package supports the description of the hardware platform, thus addressing requirement SA01. Furthermore, the "Alloc" package can be used to detail the allocation of application elements onto the available computing resources. The Generic Resource Modeling (GRM) package introduces the concept of resource, and provides several stereotypes that allow different resource types (e.g., storage, communication, computing and device resources) to be distinguished, thus addressing SA02.

A rich library of non functional quantities is included in the profile, allowing the specification of a wide range of quantitative properties (e.g., frequencies, delays). Such library does not explicitly define dependability properties; however it can be used to provide support for the definition of dependability attributes, e.g. using the `NFP_Frequency` elements to specify fault occurrence rates.

MARTE also defines the Value Specification Language (VSL), a textual language inspired by UML's Object Constraint Language, which can be used to specify parameters, variables, and relations between them, also allowing com-

posite values (such as collection, intervals, and tuple values) to be specified. VSL is a valuable support to the definition of dependability attributes (requirement FEF02) and analysis objectives (requirement A01).

Despite these useful features the profile is mainly focused on modeling real-time properties; for this reason, maintenance (M01), fault tolerance structures (SA03), and the specification of threats affecting system components (FEF01) are not considered at all.

3.4.3 *SysML*

The Systems Modeling Language (SysML [148]) reuses a subset of UML2 and provides additional extensions focused on systems engineering. SysML tries to reduce the size and software-centric structure of UML, adopting a more comprehensive view of systems' development processes. Considerable emphasis is put on the specification of system requirements and their traceability through the introduction of "Requirements Diagrams", thus fulfilling A02. Similarly as in MARTE, a comprehensive "Model Library for Dimension and Units" provides the support for describing quantitative attributes, thus addressing FEF02; however, as in MARTE, dependability properties are not explicitly defined.

SysML provides some enhancements for SA01 with respect to plain UML2. It comprises a new first class "Block" entity that can be used to model any structured information related to the system (i.e., both software and hardware for example); the concept of "Flow Port" extends the basic UML2 "Port" concept to model continuous flows of data or other materials between system components. SysML "Parametric Diagrams" allow users to represent relations between value properties of different blocks of the diagram, through mathematical expressions that specify constraints on their properties. Nevertheless, SysML has not been conceived for the specification of dependability properties, and also in this case specific concerns like maintenance, fault tolerance, and error propagation are not addressed.

3.4.4 *EAST-ADL2*

EAST-ADL2 [57] is a modeling language for electronics system engineering within the automotive domain, which reuses subsets of UML2 and SysML and provides additional extensions to satisfy specific automotive domain requirements. Safety requirements and properties are taken into account by the "extensions for SafetyRequirements" and "extensions for SafetyCase" packages.

Among the introduced features, specific subsets of the language allow to describe the erroneous behavior of components through an error model ("extensions for ErrorBehavior" package). EAST-ADL2 features thus provide extensive support for FEF01, FEF02, and A01. Due to its nature, EAST-ADL2 is however

very tied to the automotive domain: as a simple example, safety is defined by means of ASILs (*Automotive* Safety Integrity Levels).

Another issue with respect to the CHESS methodology resides in its error modeling capabilities. In EAST-ADL2, error modeling is treated as a completely separated view, orthogonal to the nominal architectural model: relationships between error behaviors are captured by means of explicit error propagation ports and connections. This approach provides the greatest degree of flexibility, but also allows the modeler to define propagation paths where no functional dependencies exist; this may result to incorrect modeling of error propagation, thus violating the "correctness-by-construction" principles.

3.4.5 *AADL*

The "SAE Architecture Analysis and Design Language" standard describes the AADL language [171], which originated from the avionic domain and has later been adopted in the automotive industry; it is currently maintained by the Society of Automotive Engineers (SAE). The AADL language allows components to be distinguished by category, including for example the "memory", "bus", and "processor" categories, thus fulfilling requirement SA02.

For the purpose of dependability annotations and analysis the AADL Error Model Annex [172] is of particular relevance, since it allows users to add dependability-related information to AADL architecture models. This information may include fault and repair assumptions, fault-tolerance mechanisms, stochastic parameters of the system, and properties of phases in a phased-mission system [71]. Separate error models are defined for system components, and error propagation takes place through components' interfaces.

The AADL Error Model Annex provides a textual language capable to address, from a modeling power point of view, most of the requirements in Table 3.1. However, no predefined attributes or properties are provided: the definition of the possible states of components, as well as the definition of propagation relations, is completely left to the user. While this approach allows for a detailed specification of dependability properties, a great effort is required to the end users even for relatively simple models.

3.4.6 *DAM*

The work in [16] defined the Dependability Analysis Modeling (DAM) profile, starting from a set of requirements similar to those in Table 3.1. DAM aims at defining a MARTE-based UML profile for dependability modeling; for this purpose, it defines some stereotypes as specializations of the stereotypes included in MARTE.

DAM introduces some features especially tailored to dependability analysis, representing one of the most complete proposals in literature. Requirement SA03 is addressed by using concepts from the approach defined in [21, 24], in which the elements of fault-tolerant structures are marked with specific stereotypes to define their role (e.g., voter, variant, etc.) within the fault-tolerant structure. The «ErrorPropagation» stereotype allows to model complex propagation relations between components, thus satisfying SA04. Requirements FEF01 and FEF02 are addressed by the "Core" and "Threats" packages, allowing to describe faults, errors, failures, and hazards.

Requirement M01 is only partially addressed by the "Maintenance" package: preventive maintenance is not fully addressed, and certain class of systems cannot be properly modeled (e.g., Scheduled Maintenance Systems, SMS [72]). Requirement A01 is addressed by defining a "Dependability Analysis Context", based on MARTE's "AnalysisContext" model element.

DAM is an important step forward in the introduction of dependability attributes at UML level, and the first attempt to create an "universal" dependability profile. In our opinion, it has however some limitations with respect to our objectives, which led us to the definition of a new language, using a different approach. On one hand, DAM is very coupled with MARTE, since most of the stereotypes are defined as extensions of MARTE concepts. For this reason, it suffers of an approach originally tailored to the analysis of real-time software properties, which makes it unbiased towards the modeling of software behavior, rather than hardware and structural properties of software. Actually, most of the stereotypes in the profile extend stereotypes of the MARTE "Generic Quantitative Analyis Modeling" (GQAM) package, which is intended to support analysies based on the software behavior [143].

While DAM supports the modeling of threats, they are specified as extensions of the MARTE::GQAM "Step" concept, which represents a primitive step of a software behavior and can be applied to very different modeling elements across the functional model. As a result, threat information may result dispersed across several diagrams, and thus difficult to aggregate by model-transformation algorithms without introducing further constraints. Moreover, as previously discussed, providing the user with such a high degree of freedom in specifying non-functional attributes does not provide guarantees on the correctness and consistency of the resulting system model.

An excessive diffusion of information is also noticeable in other parts of the language, leading to semantic overlaps. For example, the attribute *unreliabilty* can be specified for both "Component" elements, and "Service" elements, with a component providing one or more services. While it can be useful in supporting different modeling approaches, specifying both the attributes may result in an inconsistent model. To a certain extent, this problem is however not completely avoidable and derives from the intent to address priorities of different stakeholders in the same model [118].

Table 3.2: Requirements addressed by existing languages.

	QoS&FT	MARTE	SysML	EAST-ADL2	AADL	DAM
SA01	+	+	+	+	+	+
SA02		+			+	+
SA03	~			+	+	~
SA04	+		+	+	+	+
FEF01	~			~	+	~
FEF02	~			~	~	~
M01					~	~
A01	~	~	~			+
A02			+	+	+	~

While the DAM profile addresses — to some extent — all the requirements identified in Section 3.2, we found its approach not adequate for our purposes: the excessive freedom left to the users, and the possible inconsistencies resulting from the specification of non-functional properties are in contrast with the system design methodology we advocate.

3.4.7 *Summary*

Our evaluation of existing modeling languages with respect to the identified requirements is summarized in Table 3.2. Symbol "+" indicates that the requirement is properly fulfilled by the language; symbol "~" states that the requirement has been taken into account by the language, but it has not been addressed in a satisfactory way with respect to our objectives, as discussed in the previous sections. Finally, an empty cell indicates that the requirement is not addressed, or only marginally addressed, by the language.

As highlighted in the table, while some languages addressing most of the requirements exist (mainly AADL and DAM), none of them has been deemed completely satisfactory with respect to our objectives and the target system design methodology.

3.5 DEP-UML

The definition of the conceptual model first, and then the analysis of existing languages, put the basis for the definition of the "CHESS Dependability Profile", i.e., the language extensions to be applied to the core CHESS ML language in order to support the specification and analysis of dependability properties. In this thesis we focus on a specific portion of that we call "DEP-UML". This portion, which can also be seen as a standalone language, is the subset which supports the application of state-based stochastic dependability analysis. The

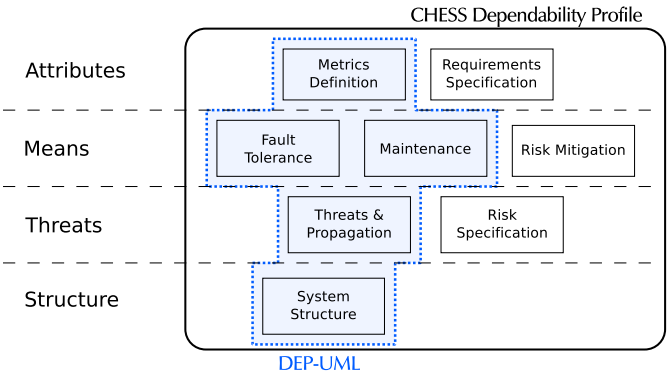


Figure 3.5: Relationship between DEP-UML and the CHESS Dependability Profile.

full specification of the CHESS Dependability Profile is given in [36], and contains model elements to support also other analysis techniques, e.g., FMECA[2] and FPTC[3]-based techniques [49, 75].

The relationships between DEP-UML, the CHESS Dependability Profile, and the conceptual model are depicted in Figure 3.5. The CHESS Dependability Profile, as a whole, is an implementation of the conceptual model described in Section 3.3. DEP-UML contributed to its definition by providing stereotypes for the *System Structure*, *Threats & Propagation*, *Fault Tolerance*, *Maintenance*, and *Metrics Specification* packages.

The main elements of the language are listed in Table 3.3. In the following, their description is provided through a practical view of the language, which describes how such stereotypes are used for modeling dependability information in a component-based system design process. The discussion will highlight the most distinctive features of the DEP-UML language, which are closely related to requirements listed in Section 3.2:

- support for the CHESS methodology for the specification of component-based system architectures (SA01);
- a set of "dependability template" stereotypes for adding dependability information to system components in a compact way (SA02, FEF02);
- simple modeling of error propagation (SA04);
- the ability to define custom error models for detailed threats specification (FEF01);
- support for hierarchical dependability modeling (SA01, SA04);
- simple modeling of redundancy structures (SA03);

2 Failure Mode, Effects and Criticality Analysis
3 Failure Propagation and Transformation Calculus

DepTemplate (*abstract*): `extensions`: *UML::Component, UML::InstanceSpecification*; `attributes`: *faultOcc*; `constraints`: *"Component must be a ComponentImplementation, or be included in the DeploymentView"*.

StatefulHardware: `extensions`: *DepTemplate*; `attributes`: *faultOcc, probPermFault, errorLatency, repairDelay*.

StatelessHardware: `extensions`: *DepTemplate*; `attributes`: *faultOcc, probPermFault, repairDelay*; `constraints`: *"May not have owned properties"*.

StatefulSoftware: `extensions`: *DepTemplate*; `attributes`: *faultOcc, errorLatency, repairDelay*.

StatelessSoftware: `extensions`: *DepTemplate*; `attributes`: *faultOcc*; `constraints`: *"May not have owned properties"*.

DependableComponent (*abstract*): `extensions`: *UML::Component, UML::InstanceSpecification*; `attributes`: *errorModel*; `constraints`: *"Component must be a ComponentImplementation, or be included in the DeploymentView"*.

Propagation: `extensions`: *UML::Connector, UML::Comment, UML::InstanceSpecification*; `attributes`: *prob, propDelay*; `constraints`: *"Comment has to be stereotyped as MARTE::Alloc::Assign"*.

ErrorModel: `extensions`: *StateMachine (from UML::StateMachines)*; `attributes`: *prob, propDelay*; `constraints`: *"ErrorModel state machine can have only states and transitions stereotyped with InternalFault, ExternalFault, Error, FailureMode, Propagation. It has to be owned by a ComponentImplementation"*.

Propagation: `extensions`: *Transition (from UML::StateMachines)*; `attributes`: *prob, propDelay*; `constraints`: *"From Error to Error, or Error to FailureMode only"*.

InternalFault: `extensions`: *Transition (from UML::StateMachines)*; `attributes`: *Occurrence, permanentProb, transientDuration*.

ExternalFault: `extensions`: *Transition (from UML::StateMachines)*; `attributes`: *fromPort, propagationCondition*.

ThreatState (*abstract*): `extensions`: *State (from UML::StateMachines)*.

Error: `extensions`: *ThreatState*; `attributes`: *vanishingTime*.

FailureMode: `extensions`: *ThreatState*; `attributes`: *affectedPorts*.

MMActivity (*abstract*): `extensions`: *UML::Activity, UML::Action*; `attributes`: *when, duration, probSuccess*.

Repair: `extensions`: *MMActivity*; `attributes`: *targets*.

ErrorDetection: `extensions`: *MMActivity*; `attributes`: *target, correctionProbability, controlledFailure*; `constraints`: *"controlledFailure must be a FailureMode defined for the target"*.

StateBasedAnalysis: `extensions`: *MARTE::GQAM:GaAnalysisContext*; `attributes`: *measure, measureEvaluationResult, targetFailureMode, targetDepComponent*; `constraints`: *"The platform attribute (from GaAnalysisContext) has to refer the system to be analyzed, i.e., a root InstanceSpecification owning hardware instances and the deployment information (MARTE::Assign)"*.

Table 3.3: Elements of the DEP-UML language.

- support for the specification of preventive and corrective maintenance activities (M01);
- detailed definition of metrics for dependability evaluation (A01).

All the attributes of DEP-UML stereotypes are based on MARTE `NFP_CommonType` elements, therefore inheriting the possibility to define their values as stochastic values, following a certain probability distribution [143].

3.5.1 *Component-based approach*

One of the peculiarities of the DEP-UML profile is the proper support for the CHESS component-based modeling methodology. In the CHESS design methodology (see Section 3.1), components are defined at type level first, possibly imported and reused from libraries, and then instantiated and connected together in a proper collaboration scenario. All the connections between components instances are assumed to be made through their (compatible) ports.

During the profile implementation specific emphasis has been put to assure that the dependability information related to a given component can be specified "out of its context", i.e., in isolation with respect to the other components that may populate the system architecture. This approach allows dependability information to be specified as a *reusable concept* for components, as already typically holds in component-based design regarding the functional specification, thus promoting reuse also in the specification of dependability properties.

However, while it should be possible to specify the dependability properties of system components in isolation, the context in which the component instance actually operates may influence its properties. A straightforward example is given by two instances of identical hard disks: while in principle they would have the same dependability properties (e.g., fault occurrence rate), the context in which the two instances operate may be of great influence. An environment subject to heavier vibrations could for example increase the fault occurrence rate of the involved disk instance.

Providing the right support for the modeling of dependability properties at instance level is therefore fundamental. As described in Section 3.1.2, CHESS ML relies on the `Property` and `InstanceSpecification` UML constructs for the modeling of component instances. The support for specializing dependability information at instance level is then provided by defining the main stereotypes as extension of the «`InstanceSpecification`» UML entity. Thanks to this feature, dependability attribute values defined at component level can be overridden (if needed) at instance level.

3.5.2 *Dependability templates*

DEP-UML provides two means to attach dependability information to system components. The most convenient way is to use a set of pre-defined "template" stereotypes that are provided by the language. Such stereotypes describe common classes of components, and allow their dependability properties to be

specified by means of a simple set of attributes. The definition of such elements has been inspired by the work developed within the past HIDE project [21].

When such stereotypes are applied, it is assumed that the involved component: i) is atomic, i.e., its internal structure is not considered, ii) is affected by a single failure mode only, and iii) when it fails, all the services it provides are affected. The introduced dependability templates distinguish between hardware and software components, and between stateful and stateless components. For software components it is assumed that they are only affected by transient faults, since permanent faults should have been removed by previous testing and debugging activities.

The combination of the two above dimensions leads to four different stereotypes (all of them extending the abstract «DepTempemplate» abstract class):

- «StatefulHardware», which models hardware component having internal state (e.g., CPUs, memories). Such components may be affected by latent errors (e.g., a CPU may hold a wrong value in a register), and they may experience transient as well as permanent faults.
- «StatelessHardware», which models hardware component without internal state (e.g., buses, cables). In such components fault activation (either of transient or permanent faults) immediately leads to failure.
- «StatefulSoftware», which models software components having an internal state. Such kind of components are only subject to transient faults, and may hold an erroneous value in one of their variables.
- «StatelessSoftware», which models software component that do not have an internal state. In such components fault activation immediately leads to failure; moreover the repair is immediate: as soon as the fault causing the failure is removed, the component is immediately working properly again, since i) there is no internal state, and ii) there failure has no "physical" effect on the component.

Based on the kind of component, a subset of the following four attributes can be specified:

- *faultOcc*, which specifies the fault occurrence rate of the component. If it is not specified otherwise, fault occurrence is assumed to follow an exponential probability distribution.
- *probPermFault*, which specifies the probability that the fault is a permanent one, as opposed to a transient fault. This attribute is applicable to hardware components only.
- *errorLatency*, which specifies the delay after which an error generates a failure of the component. If it is not specified otherwise, it is assumed to

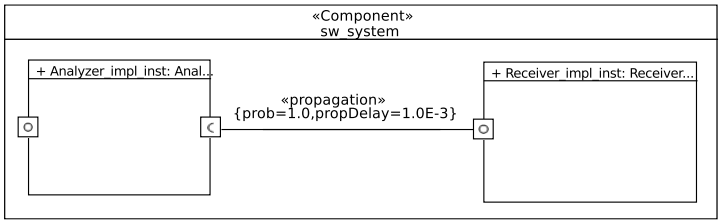


Figure 3.6: Applying dependability information to component instances connections.

follow an exponential distribution. This attribute is applicable to stateful components only.

- *repairDelay*, which specifies mean time required to repair the component. If it is not specified otherwise, it is assumed to follow an exponential distribution.

Such stereotypes can be used from the initial phases of system development, when detailed information on components dependability properties is not yet available.

Following the CHESS methodology, such stereotypes can be applied to `UML::Component` elements and `UML::InstanceSpecification` elements; the latter enables the specification of dependability properties at component instance level. An additional constraint imposes that the stereotype is applied only on «`ComponentImplementation`» elements, or «`Component`» elements in the *DeploymentView*. This ensures that dependability information is added only to hardware components, or to software components which already have an implementation.

In principle, the language can be extended with additional template stereotypes. However, when a detailed specification of dependability threats is needed, it is usually more practical to use other advanced facilities provided by DEP-UML like the error model construct described later.

3.5.3 *Error propagation*

Once components are instantiated and connected to form the overall system architecture, information concerning error propagation can be properly modeled with DEP-UML, by specifying the probability that error propagation occurs between two communicating components, and the delay after which it will occur.

Functional connections between components identify possible propagation paths within the system. Propagation may occur between two components instances of the same kind (i.e., hardware or software) which are connected together by a functional relation. Propagation can also occur from hardware to software component instances, whenever allocation (deployment) relations exist between them.

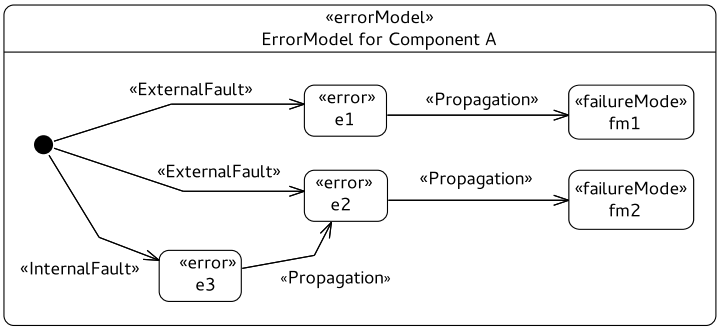


Figure 3.7: DEP-UML Error Model example.

The actual information about error propagation is specified through the «`Propagation`» stereotype (Figure 3.6). Such stereotype may be applied to those language constructs that are used, in the functional model of the system, to specify functional relations and deployment information, i.e.:

- `UML::Connector` elements, which specify functional relations by connecting the different components instances through ports (both software and hardware);
- `MARTE::Assign` comments, which are used to specify deployment information, i.e., allocation relations between software and hardware components instances.

The stereotype allows the user to specify the probability that propagation occurs between the two components (through *prob* attribute), and the delay after which it will occur (through the *propDelay* attribute). If the stereotype is not applied, or its parameters are not specified, it is assumed that propagation between two communicating components occurs instantaneously, with probability 1.

Finally, the «`Propagation`» stereotype can also be applied to `InstanceSpecification` elements, thus allowing different propagation properties to be specified for different connector instances (i.e., instances of the same `UML::Connector` element).

3.5.4 *Error Model*

Dependability templates described above provide a reasonable balance between convenience and modeling power; however, if greater expressiveness is needed, it can be achieved by attaching a specific "Error Model" to components.

Error models allow designers to provide more details on faults, errors, and failure modes affecting system components. The ability to define detailed error models has been recognized as a useful feature in other dependability-oriented

ADLs. The DEP-UML error model has been inspired by the Error Model Annex for AADL [172], and the EAST-ADL2 language [57], both featuring a similar facility, although with some limitations as discussed in Section 3.4.

The DEP-UML error model provides a simple UML-based graphical notation (like EAST-ADL2), in which however error propagation is constrained to dependency relations already existing in the architectural model of the system (like in AADL).

Specifying the error model

The DEP-UML Error Model is defined as a particular kind of state machine, which is identified with the ad-hoc «`ErrorModel`» stereotype (Figure 3.7). Once an error model is created, it is then applied to a component by means of the «`DependableComponent`» stereotype and its *errorModel* attribute.

For consistency with UML2, an «`ErrorModel`» state machine must contain an *initial state* (i.e., a `Pseudostate` with *kind=initial*); this state represents the component's "healty" state, i.e., the state in which the component is correctly working. Errors and failure modes are modeled as «`UML::State`» elements, using the DEP-UML «`Error`» and «`FailureMode`» stereotypes, respectively.

Faults are modeled as transitions (`UML::Transition` elements) between the initial state, and states stereotyped as «`Error`». Two kinds of faults can be modeled in a DEP-UML Error Model. The «`InternalFault`» stereotype represents faults that occur spontaneously; conversely the «`ExternalFault`» stereotype represents faults that are originated by propagation, i.e., they are coming from the environment where the current component is (or can be) instantiated.

«`InternalFault`» elements own the *Occurrence* attribute, which allows the probability distribution of fault occurrence to be specified, and *permanentProb* and *transientDuration*, which describe the probability of the fault to be a permanent fault, and optionally the average duration of a transient fault. The «`ExternalFault`» stereotype has two attributes, *fromPort* and *propagationCondition*, which allow the source of fault to be specified.

The relations between erroneous states of the component, including the relations between errors and failure modes, are specified with `UML::Transition` elements. Non-functional attributes that may be applied to such transitions are the propagation probability and propagation delay, in a similar way as for connection between components. For this purpose, the «`Propagation`» stereotype is extended (technically with a "merge increment" [150]) in order to be applied to `UML::Transition` element as well.

Specifying affected services

With respect to the need of specifying dependability information out of component's context, but still support detailed specification of component's threats, one of the most interesting features provided by the DEP-UML Error Model

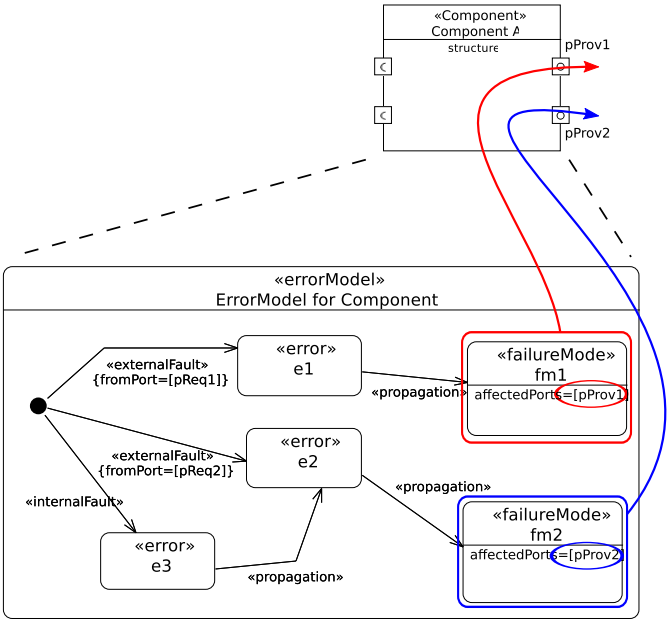


Figure 3.8: Connecting failure modes to provided services, through the "affectedPorts" attribute.

is the ability to link failure modes and external faults to the services provided and required by the component, respectively.

By default it is assumed that all the failure modes defined in a component's error model affect all the services provided by the component. However, it is not necessarily the case: the classic taxonomy of dependability [6] defines a failure as *the event that occurs when the provided service becomes incorrect*. It is therefore evident that the failures of a component are strictly related to the services it provides; if a component provides several services, the occurrence of a failure may involve only one, a subset, or all of them.

In DEP-UML such behavior can be represented by linking `«FailureMode»` elements to a set of output ports owned by the component, through the specific *affectedPorts* stereotype's attribute. Such relation makes it possible to identify the services affected by the failure among those provided by the component (Figure 3.8). More in general, the *affectedPort* attribute is used to select a subset of ports from those which may propagate incorrect data, i.e., any service port with an output data flow (for software components), and output data ports (for hardware components).

Similarly, a fault may occur in a component even when only a subset of the services it requires fail. The failure of a service on which the component relies is perceived as an external fault [6], which may generate errors and finally lead the involved component to fail. If a component requires different services, their failure may have different effects on the internal state of the component, pos-

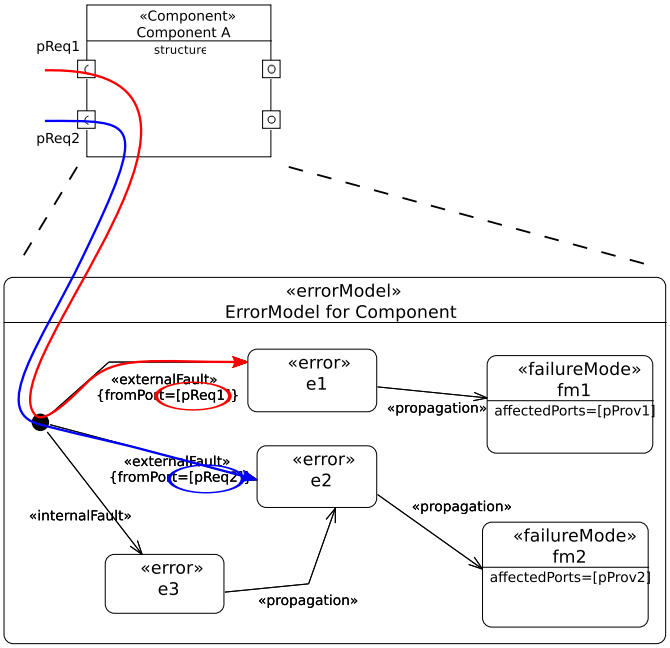


Figure 3.9: Connecting external faults to required services, through the "fromPort" attribute.

sibly activating different propagation paths and finally leading the component to different kind of failures (i.e., different failure modes).

To model this aspect of error propagation, «ExternalFault» elements are linked to a set of input ports owned by the component through the *fromPorts* attribute (Figure 3.9). More in general, the *fromPorts* attribute selects a subset of component's ports that can receive faulty data, i.e., any service port with an input data flow (for software components), and input data ports (for hardware components).

The additional *propagationCondition* attribute can be used to specify the condition required for the fault to occur, as a combination of failures occurred on the specified input ports. The grammar for specifying such conditions is reported in the following:

Listing 3.1: *Grammar for the specification of propagation conditions.*

```
<COND> ::= <SIMPLECOND> | <EXTCOND>
<SIMPLECOND> ::= AND | OR
<EXTCOND> ::= <EXTCOND> AND <EXTCOND> | <EXTCOND> OR <EXTCOND> | <F>
<F> ::= <PORT> | <PORT>[<FAILUREMODE>]
<PORT> ::= <text>
<FAILUREMODE> ::= <text>
```

The <COND> rule describes a propagation condition. A propagation condition may be specified as a simple condition (<SIMPLECOND>), or as an extended

condition (<EXTCOND>). A simple condition can be used to specify that a failure must occur on all the ports specified by the *fromPort* attribute ("AND"), or on any of them ("OR").

Extended conditions allow to specify more complex conditions; an extended condition may be either a logical AND or OR of two extended conditions, or the specification of a failure occurrence (<F> rule). A failure occurrence is specified as a `UML::Port`, and optionally the name of a failure mode; the port must be one of the "input" ports owned by the component for which the error model is being defined. Such condition specifies that propagation occurs when the component connected on the specified port fails. If the name of the failure mode is specified, only failures occurring with that specific failure mode are taken into account.

This facility allows the modeler to specify detailed propagation behavior between components, and it can also be used to model redundancy. In principle, the grammar can be extended to support the specification of more advanced conditions (e.g., k-out-of-n).

3.5.5 *Hierarchical and modular modeling*

DEP-UML pays special attention in supporting hierarchical and modular modeling of dependability properties. Hierarchical modeling and decomposition is a fundamental aspect of model-driven system development, since it allows progressive refinement of the model, thus enabling an iterative and incremental development approach. Two key features of DEP-UML in this perspective are described in the following.

Three ways to annotate composite components

The language supports three different ways to model the dependability properties of composite components, i.e., components whose internal structure is detailed in the functional model of the system. The first alternative is to use the set of dependability templates described in Section 3.5.2, thus disregarding the internal structure of the component, and considering it as an atomic component for the purpose of dependability analysis. This approach is particularly useful in the initial phases of model refinement, when the dependability properties of subcomponents may not be yet known.

The second way to attach dependability information to composite components is by means of the error model mechanism described in Section 3.5.4: the stereotype «`DependableComponent`» (and thus error models) can be applied to composite components as well. Also in this case, however, the internal structure of the component is not taken into account for dependability analysis, since the dependability behavior of the component is completely defined by the error model.

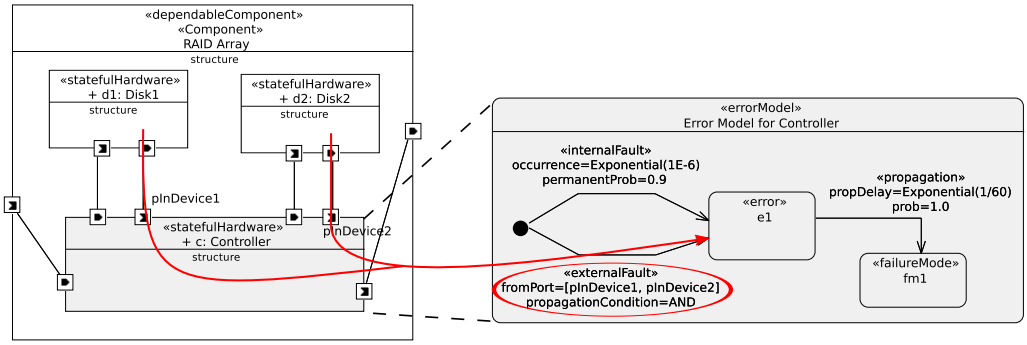


Figure 3.10: Modeling of redundancy structures in DEP-UML. The figure depicts the model of a redundant RAID array with 2 disks and a controller.

As a third alternative, the dependability behavior of the upper level component can be completely derived from the dependability behavior of its subcomponents, if available. To apply this strategy all the input and output ports of a component *must* be delegated to its subcomponents; in addition, the dependability properties of the subcomponents should have been specified. In this case, composite components are seen as logical containers with no functional elements. Since all the required and provided services are delegated to subcomponents, the failures affecting the composite component can be derived from those affecting the subcomponents to which service interfaces are delegated.

These three alternatives play a key role in the incremental and hierarchical modeling, providing different level of details corresponding to different stages of system design.

3.5.6 *Modeling of redundancy structures*

By combining the features presented above, DEP-UML provides a simple way to model redundancy in component-based architectures. Redundancy structures are in general composed of a set of variants, which provide the redundant service, an adjudicator, which checks the correctness and agreement of variants, and a redundancy manager, which is in charge of communicating with the rest of the system and possibly decide the redundancy strategy.

From the perspective of component-based architecture, the services provided by the redundancy structure are delegated to its subcomponents, with the redundancy manager being in charge to manage this delegation. The non-functional behavior of redundancy structures can thus be modeled in DEP-UML using the error model facility (Section 3.5.4) and composite components (Section 3.5.5).

Figure 3.10 shows the definition of a redundant RAID array composed of two identical disks in a mirrored setup and a controller. The whole redundancy structure is modeled as a composite component, containing two instances of

the "Disk" component and a controller. The controller is the interface of the redundancy structure with the rest of the system: input and output ports of the upper-level component are delegated to it. The controller is also connected in a bidirectional way to the two disks; failures from the two disks may then propagate to the controller and vice-versa.

The two disks are annotated with the «StatefulHardware» stereotype, while the redundancy mechanism implemented by the controller is described by attaching an error model to it. In the example of Figure 3.10 an «ExternalFault» connects the initial state of the controller with the error "e1". The external fault occurs in the controller when it receives faulty data from the disks; however, since the two disks are mirrored, the fault will occur only when both of them are failed. To represent this setup in the error model, the two input ports connecting the controller with the two disks are added to the *fromPorts* attribute, while the *propagationCondition* attribute is set to "AND".

When both disks are failed, the resulting error will then propagate as a failure of the controller, and (because of delegation) as a failure of the overall redundancy structure. Finally, an internal fault of the controller can also lead to a failure of the redundancy structure, regardless of the working state of the two disks.

3.5.7 *Maintenance activities*

When dependability templates are used, maintenance information can be attached using the *repairDelay* attribute, which specifies an exponentially distributed delay after which the component is restored to its original healthy state.

While such simple specification of repairs is useful, it is not powerful enough to specify more advanced system-level maintenance strategies. DEP-UML provides an additional and more detailed mechanism to model maintenance strategies, based on the concept of activity: a maintenance strategy is a collection of activities that are performed on the system when certain conditions hold. Such mechanism is also used to specify maintenance policies for components for which an error model is defined; the error model itself in fact does not provide any information on repairs.

Activities are specified using extensions of the «MMActivity» abstract element, which extends `UML::Activity` and `UML::Action`. The conditions that trigger the execution of an activity are specified by the *when* attribute, through an expressions in the grammar below.

Listing 3.2: *Grammar for specifying conditions for the execution of activities.*

```
<S> ::= <T> [<EX>] | <T> [<EX>] {<L>}
<T> ::= Immediately | AtTime(<realnumber>) | Periodic(<realnumber>)
<EX> ::= (<EX> and <EX>) | (<EX> or <EX>) | not <EX> | true | <FD>
```

```
<FD> := Failed(<FailureMode>) | Detected(<Error>)
<L> := Before(<RealNumber>) | After(<RealNumber>) |
    Interval(<RealNumber>,<RealNumber>)
```

Expressions of such grammar are constituted of three parts: i) a description of the scheduling of the activity with respect to time in the system's lifetime (<T> rule); ii) a condition on the system's state that must hold in order for the activity to be executed (<EX> rule); and iii) an additional (optional) condition that enables the execution of the activity only in a predefined interval of time (<L>). More in detail, the symbols that form the above grammar have the following meaning:

- *Immediately*. The activity is executed immediately as soon as the conditions specified by <EX> hold.
- *AtTime(<RealNumber>)*. The activity is executed at the instant of time specified by the <RealNumber> element, starting from the beginning of the scenario.
- *Periodic(<Distribution>)*. The activity is executed periodically, at intervals of time following the probability distribution that is specified. For periodic activities, we consider the interval of time starting from the beginning of an activity execution and the beginning of the subsequent one. If the activity duration is greater than this interval of time, the activity is executed immediately as soon as the previous execution completes.
- *Failed(<FailureMode>)*. Predicate on the state of a component; this predicate is true if the component has failed with the failure mode specified by the <FailureMode> element.
- *Detected(<Error>)*. Predicate on the state of a component; this predicate is true if the error <Error> has been detected by error detection mechanisms.
- *Before(<RealNumber>)*. The activity can be executed only at instants of time prior to the value specified by <RealNumber>.
- *After(<RealNumber>)*. The activity can be executed only at instants of time after the value specified by <RealNumber>.
- *Interval(<RealNumber>,<RealNumber>)*. The activity can be executed only in the interval of time identified by the two <RealNumber> values. The boundaries are included in the interval.

As an example, the expression "*AtTime(10.000)*" describes a condition for which the activity is triggered exactly when 10.000 units of time are elapsed. Conversely, the expression "*Immediately [Failed(A.fm1)]*" describes a condition for which the activity that is executed immediately after failure mode "fm1" of component "A" has occurred.

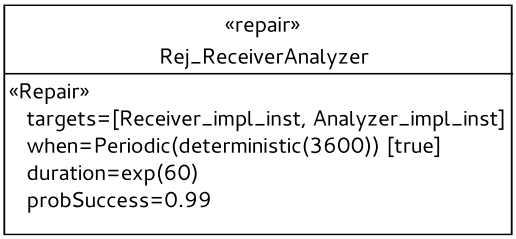


Figure 3.11: A repair activity modeling the periodic software rejuvenation of two component instances.

The additional *duration* attribute specifies the probability distribution of the time required to perform the activity. Two kinds of maintenance-related activities are considered: "error detection" and "repair", which are identified by the «ErrorDetection» and «Repair» stereotypes respectively, both extending the «MMActivity» element.

The «ErrorDetection» stereotype describes an error detection activity that is performed on a component. In this context "error detection" is intended in a general way as any activity that has the objective to discover an erroneous state of a component. This includes, for example, periodic checks on memories, assertions checking on variables, overhaul of mechanical components. Based on the result of this activity, a repair activity can subsequently be triggered. The attributes of the activity allow specifying a set of component instances (i.e., InstanceSpecification elements) that are the targets of error detection, its duration, and its success probability, i.e., the coverage of the detection mechanism.

The «Repair» stereotype describes a generic maintenance activity, and can represent both preventive and corrective maintenance based on the expression specified in the *when* attribute. Depending on the kind of component, the such stereotypes may be used to model different maintenance operations. For example, a «Repair» or a hardware component may involve its replacement with another (new) component, while for a software component it may simply represent its restart. Figure 3.11 depicts the specification of a repair activity that is periodically executed every 3600 time units and targets the two software component instances "Receiver_impl_inst" and "Analyzer_impl_inst" (e.g., for software rejuvenation [93]). The completion of the activity takes 60 time units on the average, and succeeds 99% of the times.

3.5.8 *Metrics specification*

One of the main features enabling the automatic derivation of analysis models is the ability to clearly define the metrics that should be evaluated during the analysis.

In CHESS ML, the support for metrics specification is provided by reusing a subset of MARTE, in particular the Generic Quantitative Analysis Modeling (GQAM) sub-profile. The «`GQAM::GaAnalysisContext`» stereotype is used as the basis to model the proper analysis context in CHESS ML; extensions are then be provided to allow the specification of properties that are specific to the different analysis methods.

Within DEP-UML, measures that should be evaluated by quantitative dependability analysis are specified using the «`StateBasedAnalysis`» stereotype. Such stereotype allows the modeler to specify the metrics to be evaluated (through the *measure* attribute), as well as the specific component instances to take into account (*platform*, *targetDepComponent*, *targetFailureMode* attributes).

The *metric* attribute expects a string value formatted according a specific grammar described below.

Listing 3.3: *Grammar for the specification of metrics of interest.*

```
<METRIC> ::= <R> | <A>
<R> ::= Reliability { <INST> }
<A> ::= Availability { <INST> } | Availability { <INTV> }
<INST> ::= instantOfTime = <T>
<INTV> ::= intervalEnd = <T> | begin = <T>, end = <T>
<T> ::= <RealNumber>
```

The language supports the specification of the *reliability* and *availability* metrics. The reliability metric is specified with respect to a given instant of time t, while availability can be specified either with respect to an instant of time t, an interval of time $[0, t]$, or more in general with respect to an interval of time $[a, b]$.

Once the kind of metric has been specified, the modeler has to define the "targets" for the metric, i.e., the components with respect to which such metric should be evaluated; this task is performed as follows.

The *platform* attribute, inherited from the «`GaAnalysisContext`» stereotype, specifies the instance of the overall platform (i.e., the system) on which the analysis should be executed. This attribute should refer to the `UML::InstanceSpecification` representing the overall system, which is automatically generated when the user triggers the "Build Instances" command within the CHESS editor (see Section 3.1.4).

Within the target platform, the *targetDepComponent* attribute allows the user to select a component instance of interest, on which the metrics will be evaluated. The metrics will then be evaluated considering the system failed when the selected component instance is failed. Typically, the selected (sub-)component would be the one which provides the desired system service or functionality, i.e., one to which a specific system functionality is directly delegated, or to which a specific system requirement is assigned. The user is also allowed to

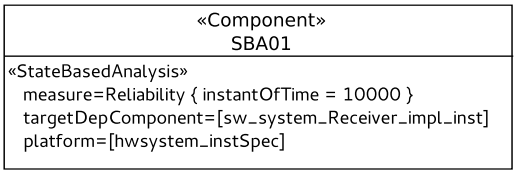


Figure 3.12: Specification of measures of interest: example for instant of time reliability.

select more than one component instance as target: in that case the system is considered failed when all the considered component instances are failed.

Using the *targetFailureMode* attribute, the target for the evaluation can also be restricted to a specific subset of failure modes, among those affecting the component instances specified in *targetDepComponent*. This attribute may therefore refer to «`FailureMode`» elements defined in a «`ErrorModel`» `StateMachine`. If this attribute is specified, then the system will be considered failed only if the component instances specified in the *targetDepComponent* attribute are failed with one of the failure modes specified with the *targetFailureMode* attribute.

An example of metric definition is provided in Figure 3.12, which depicts the specification of an instant of time reliability measure. The defined metric targets the component instance "sw_system_Receiver_impl_inst" within the platform "hwsystem_instSpec". The expression specified as "measure" attribute states that the measure should be evaluated at the instant of time $t = 10.000$ time units. Finally, the «`StateBasedAnalysis`» stereotype has an additional attribute, the *measureEvaluationResult* attribute. This attribute is used for the back-annotation procedure: it is the attribute where the value of the evaluated metric will be stored by the toolchain.

It should be noted that, based on the actual adopted analysis technique, a set of additional background information may be needed. Such information may be related for example with the solution process, the adopted tool or the analysis method itself. In the CHESS framework, such information is provided by means of global parameters of the modeling framework, without the need to add further elements to the language (e.g., the path to the analysis tool on the filesystem, or tool-specific configuration).

3.6 SUMMARY

In this chapter we described the process that we adopted for enriching a component-based design process with support for quantitative dependability analysis. A set of requirements were first identified, which then guided the definition of a conceptual model collecting the main domain elements of interest; existing languages in literature were then identified and analyzed.

Such process led to the definition of the "CHESS Dependability Profile", which supports different analysis techniques. We focused on a specific set of

extensions, that we call DEP-UML, which provide support for the specification and analysis of quantitative dependability properties. DEP-UML extensions have been defined by taking into account the actual CHESS methodology and component model, thus ensuring the applicability of dependability extensions on functional models designed with CHESS ML.

The following chapters will describe a set of model-transformations for automated dependability analysis of DEP-UML models (Chapter 4), the concrete realization of the analysis plugin (Chapter 5), and the application of the approach to two case studies (Chapter 6).

4

AUTOMATED DEPENDABILITY ANALYSIS: TRANSFORMATIONS

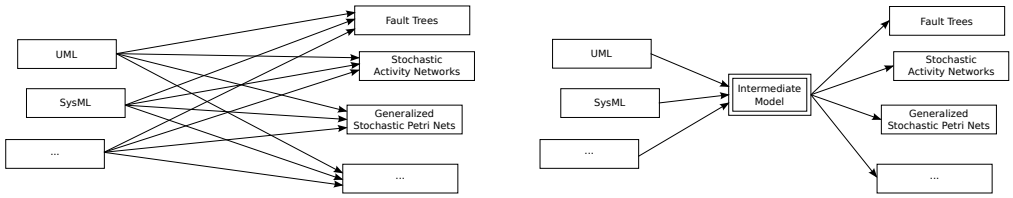
In this chapter we define a set of model-transformation rules that enable the automated generation of stochastic models for quantitative dependability analysis based on the DEP-UML extensions introduced in Chapter 3. An overview of the adopted approach is first presented in Section 4.1.

4.1 APPROACH

Most of the works adopting MDE principles for dependability analysis define a direct transformation from the high-level architectural model to the analysis model. The resulting transformation rules are usually characterized by low flexibility (i.e., they are hard to adapt to changes in the target languages) and low reusability (i.e., they are hard to adapt to different languages).

The approach that we use in this thesis solves this problem by relying on an *intermediate model*, which acts as a bridge between the high-level modeling language and the dependability analysis formalism. The intermediate model introduces an additional abstraction layer, through a representation that is independent of both the engineering modeling language and the analysis formalism. Although the introduction of an additional transformation step might seem to add unnecessary complexity, the definition of the two transformations will typically require less effort than the definition of a single, monolithic, one. Moreover, the adoption of an intermediate model generates more flexible transformations: should one of the two languages (i.e., the high-level language or the analysis formalism) change, only the transformation rules for that language would be affected, leaving the rules on the other side unchanged. In addition, if we consider n engineering languages and m analysis formalisms, $n \times m$ possible transformations between them exist; however, if using an intermediate model, only $n + m$ transformation algorithms are enough to cover all the possible combinations (Figure 4.1). With similar motivations, the importance of using an intermediate model was recognized in other work in the literature as well, e.g., see [81, 117].

As intermediate model we adopt the "Intermediate Dependability Model" (IDM) that we introduced in [130, 131]; such language was conceived exactly with the purpose of being an intermediate model to support automated quantitative dependability analysis. As analysis formalism we adopt the class of Stochastic Petri Nets as defined in [44], in which the firing delay of timed tran-



(a) Direct transformations: $n \times m$.

(b) Using an intermediate model: $n + m$.

Figure 4.1: An intermediate model reduces the number of transformation that need to be defined, considering n engineering languages, and m analysis formalisms.

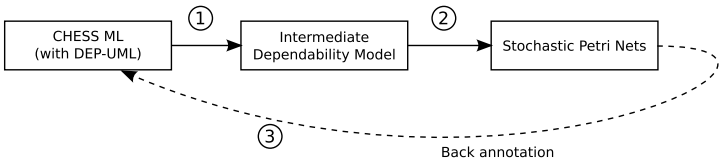


Figure 4.2: High-level view of the CHESS plugin for state-based dependability analysis.

sition is specified by an arbitrary probability distribution. The reason behind this choice is due mainly to the need of supporting: i) non-exponential occurrence of faults (e.g., for mechanical components), and ii) periodic maintenance schedules. The resulting workflow is then depicted in Figure 4.2:

i) a first transformation algorithm generates an IDM model from a CHESS ML architectural model enriched with DEP-UML dependability annotations;

ii) a second transformation algorithm generates a SPN model the IDM model;

iii) as a third step, the SPN model is analyzed and results are reported back to the original model.

The IDM language is briefly recalled in Section 4.2; its full detailed specification can be found in [42, 130]. Section 4.3 describes the first transformation algorithm (from DEP-UML to IDM), while the second one (from IDM to SPNs) is described in Section 4.4. The back-annotation process, which is more related to the implementation, is addressed in the next chapter.

4.2 THE INTERMEDIATE DEPENDABILITY MODEL (IDM)

A preliminary version of the IDM was introduced in [134]. The IDM has been later refined and aligned to the conceptual model of Section 3.3.

It is worthwhile to note that not all the elements in the conceptual model have a direct representation in the intermediate model: some of them (e.g., redundancy structures), are actually represented as by means of more elementary

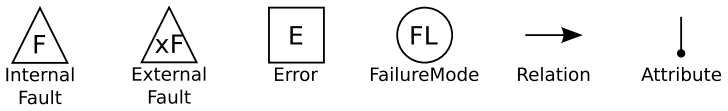


Figure 4.3: Graphical notation for IDM models.

IDM elements (i.e., logical conditions on propagation paths). This approach has been chosen in order to keep the IDM metamodel as simple as possible and avoid duplicated notation.

4.2.1 *Overview*

The IDM is composed of five logical packages: *Statistics*, *Dependable Components*, *Threats & Propagation*, *Maintenance & Monitoring*, *Dependability Analysis*. The main elements of the IDM model and their attributes are summarized in Table 4.1.

The *Statistics* package contains a definition of the main probability distributions that are commonly used in dependability analysis. The *Dependable Components* package contains the definition of an IDM `Component`: an element which contains a certain number of faults, errors, failure modes, and the propagation relations between them. The *Threats & Propagation* package contains the actual definition of threats, i.e., `InternalFault`, `ExternalFault`, `Error`, and `FailureMode`, and of propagation relations between them.

The *Maintenance & Monitoring* package contains the definition of maintenance and monitoring (mainly error detection) activities. To specify the conditions for performing activities, the IDM uses the *when* attribute in a mechanism similar to the one employed by DEP-UML (Section 3.5.7). Finally, the *Dependability Analysis* package provides support for specifying metrics of interest for the analysis.

The IDM is composed of nodes and relations, and it can be conveniently expressed using a graphical notation (Figure 4.3). In the IDM graphical notation, different nodes are distinguished by their shape: faults are represented by triangles, errors by squares, and failure modes by circles. This distinction permits to easily identify the elements involved in propagation paths. Relations between two elements of the model are represented by an arrow following the direction of the relation, while attributes are represented by short lines ending with a dot.

However, the main purpose of the graphical notation is to help understanding how model elements are organized within IDM and which are the relations between them, and to facilitate the description of model-transformation algorithms. As stressed throughout this thesis, the intermediate model is generated by automated transformations, and the user should not be able to modify or even access it.

—*Statistics*—

Distribution (*abstract*): –

Exponential: `attributes`: *Rate*; `extensions`: *Distribution*.

Deterministic: `attributes`: *Value*; `extensions`: *Distribution*.

Gaussian: `attributes`: *Mean, Variance*; `extensions`: *Distribution*.

Uniform: `attributes`: *Lower, Upper*; `extensions`: *Distribution*.

Gamma: `attributes`: *Alpha, Beta*; `extensions`: *Distribution*.

Weibull: `attributes`: *Alpha, Beta*; `extensions`: *Distribution*.

—*Dependable Components*—

Component: `attributes`: *Name*; `associations`: *Faults, Errors, FailureModes, FaultsGenerateErrrors, InternalPropagations, ErrorsProduceFailure*.

—*Threats & Propagation*—

Fault (*abstract*): `attributes`: *Name*.

InternalFault: `attributes`: *Occurrence, PermanentProbability, TransientDuration*; `extensions`: *Fault*.

ExternalFault: `associations`: *Source*; `extensions`: *Fault*.

Error: `attributes`: *Name, VanishingTime*.

FailureMode: `attributes`: *Name, Domain, Detectability, Consistency, Consequences*.

FaultsGenerateErrors: `attributes`: *Name, ActivationDelay, PropagationProbability, Weight, PropagationLogic*; `associations`: *Source, Destination*.

InternalPropagation: `attributes`: *Name, PropagationDelay, PropagationProbability, Weight, PropagationLogic*; `associations`: *Source, Destination*.

ErrorsProduceFailure: `attributes`: *Name, PropagationDelay, PropagationProbability, Weight, PropagationLogic*; `associations`: *Source, Destination*.

—*Maintenance & Monitoring*—

Activity (*abstract*): `attributes`: *Name, Duration, When*; `associations`: *Performer*.

RepairActivity: `attributes`: *SuccessProbability*; `associations`: *Targets*; `extensions`: *Activity*.

ReplaceActivity: `attributes`: *SuccessProbability*; `associations`: *Target, Replacement*; `extensions`: *Activity*.

DetectionActivity: `attributes`: *Coverage, FalsePositiveRatio, CorrectionProbability*; `associations`: *DetectableErrors, ControlledFailure*; `extensions`: *Activity*.

—*Dependability Analysis*—

DependabilityMeasure (*abstract*): `attributes`: *Name, RequiredMin, RequiredMax*; `associations`: *Target, Evaluations*.

Reliability: `extensions`: *DependabilityMeasure*.

Availability: `extensions`: *DependabilityMeasure*.

Safety: `extensions`: *DependabilityMeasure*.

EvaluationType (*abstract*): –

SteadyState: `extensions`: *EvaluationType*.

InstantOfTime: `attributes`: *TimePoint*; `extensions`: *EvaluationType*.

IntervalOfTIme: `attributes`: *Begin, End*; `extensions`: *EvaluationType*.

Table 4.1: Main elements of the IDM metamodel and their attributes.

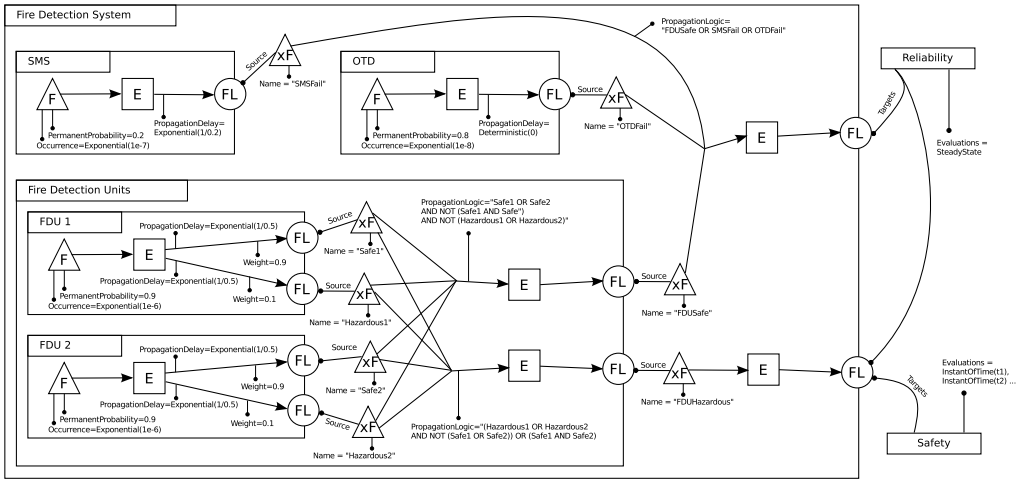


Figure 4.4: IDM model of the fire detection system.

4.2.2 *Usage Example*

In this section we provide an usage example of the IDM, through a simple but representative example of a fire detection system mounted on-board of automatic light train systems [134]. In such environment, both safety and reliability are of utmost importance for the service provider. On one hand, a reliable detection of fire events must be provided, to ensure the safety of passengers; on the other hand a transport system should provide a continuous service, and false alarm should then be avoided.

The fire detection system takes its decision based on a set of Smoke Sensors (SMS) and a set of Over-Temperature Detectors (OTD). These sensors are managed by two Fire Detection Units (FDU), which analyze the data received from them and trigger the alarm signal when a fire event is detected. Both the FDU are able to detect a fire event, but only one of them is allowed to control the sensors at the same time.

Each FDU is subject to two failure modes: a "safe" failure mode, and a "hazardous" failure mode. When a FDU fails in the safe mode, an alarm is triggered and the other FDU takes control of the sensors. When instead a FDU fails in the hazardous mode, it prevents the other from having access to the sensors data, thus making the system unable to detect a fire event. A system hazard occurs if: i) at least one FDU fails in a hazardous mode, or ii) if both the FDU fail, in any of the two failure modes. For what concerns sensors, their failures are always considered safe.

We are interested in two kinds of measures: the reliability of the system at steady-state, which can be expressed with its Mean Time To Failure (MTTF); and the probability that hazardous event has not occurred at different instants of time.

The IDM representation of the fire detection system described above is shown in Figure 4.4. For simplicity, the smoke sensors have been considered as a single hardware component; the same holds for over-temperature detectors.

The component corresponding to smoke sensors is depicted in the upper-left part of the figure (labelled "SMS"); after a certain delay (*Occurrence* attribute) a fault develops inside the block, and with a certain probability it may be transient or permanent (*PermanentProbability* attribute). The fault generates an error in the component, and after an additional propagation delay (*PropagationDelay* attribute) the error reaches the external interface of the component, causing a failure of the smoke sensors block. The component corresponding to over-temperature detectors is labelled "OTD" in the figure and its structure is similar to the one used for smoke sensors. However, temperature sensors are stateless component (they do not have an internal state) and any fault immediately causes them to fail. To represent this aspect in the intermediate model, the propagation delay has been set to zero for the "OTD" component. The two FDU are shown in the lower-left part of the figure. As for the model of sensors described above, each of the two FDU may be affected by a fault, which may then generate an error inside the component. In the components corresponding to the two FDUs, an error may cause two distinct failure modes, which correspond to the "safe" and "hazardous" failure modes of the units.

The two FDU are enclosed in the higher-level logical component labelled "Fire Detection Units"; the `FailureMode` elements of the single FDUs are connected to `ExternalFailure` elements of the higher-level component, through the *Source* attribute. The higher-level component is then affected by four different faults: the safe failure of FDU 1, the hazardous failure of FDU 1, the safe failure of FDU 2, and the hazardous failure of FDU 2. Different combinations of these failures (specified by the *PropagationLogic* attribute of propagation relations) propagate as two different failure modes for the higher-level component: a "safe" and a "hazardous" failure mode.

The highest-level component (labelled "Fire Detection System") represents the whole system and is affected by four different kind of faults corresponding to the failures of its subcomponents: i) the failure of the smoke sensors block, ii) the failure of the over-temperature detectors block, iii) the "safe" failure of the FDUs block, and iv) the "hazardous" failure of the FDUs block. When one of the first three events occurs, it propagates as a safe failure mode of the system-level component. The "hazardous" failure mode of the FDUs block propagates as a "hazardous" failure of the system.

The measures of interest are specified by the two `Reliability` and `Safety` model elements, which are connected to the `FailureMode` elements of the component representing the whole system. The *Evaluations* attribute specifies the type of measure that should be evaluated: `SteadyState` for reliability and `InstantOfTime` for safety.

4.3 FROM DEP-UML MODELS TO IDM MODELS

This section describes the transformation algorithm that generates an IDM model from a DEP-UML model; a preliminary version of the transformation was presented in [131]. The transformation algorithm is based on the following assumptions:

- the functional model of the system has been designed following the CHESS methodology described in Section 3.1.2;
- the DEP-UML model contains a proper description of component instances through `UML::InstanceSpecification` elements;
- to each system component (or component instance) it has been applied *at most one* of the following stereotypes: «`DependableComponent`», «`StatefulHardware`», «`StatelessHardware`», «`StatefulSoftware`», «`StatelessSoftware`»;
- the stereotype applied to a component instance matches the one applied to the related `Component` (or `ComponentImplementation` for software components);
- stereotypes «`StatefulHardware`» and «`StatelessHardware`» are applied on hardware components (or component instances) only;
- stereotypes «`StatefulSoftware`» and «`StatelessSoftware`» are applied on software component implementations (or component instances) only;
- each software component instance is deployed on at most one hardware component instance;
- all the provided/required ports of a composite component are delegated to provided/required ports of its subcomponents; the same holds for in-/out flow ports.

When using the CHESS framework (i.e., when the input model is a CHESS ML model) the fulfillment of such assumptions is guaranteed by CHESS views and other constraints enforced by the CHESS editor.

To simplify the description of the transformation algorithm we introduce some notation that we will use in the rest of the section. Given an `InstanceSpecification` $inst$, we denote with:

- $component(inst)$, the `Component` (or `ComponentImplementation` for software) from which the instance originates. This information is obtained from the *classifier* attribute of $inst$ [150].
- $subs(inst)$, the collection of `InstanceSpecification` elements representing sub-instances of $inst$. This information is obtained from the *value* attribute of `Slot` elements owned by $inst$ [150].

- $parent(inst)$, the `InstanceSpecification` representing the parent component instance of $inst$, i.e., the `InstanceSpecification` p such that $inst \in subs(p)$.

The transformation is organized in five phases: i) creation of components in the IDM model; ii) projection of DEP-UML dependability templates; iii) projection of DEP-UML error models; iv) projection of non-stereotyped components; v) projection of propagation relations; vi) projection of activities; vii) projection of analysis objectives.

As mentioned in Section 3.5, most attributes of DEP-UML stereotypes are derived from MARTE `NFP_CommonType`, thus allowing them to be defined through probability distributions. Attribute values defined through probability distributions are projected in the corresponding *Distribution* element of the IDM model. In case they are specified as numerical values, their interpretation depends on the actual attribute, and it is described as part of the transformation rules.

4.3.1 *Creation of components*

In this phase the `Component` elements that need to be created in the IDM model are identified. Based on the applied stereotypes, only a subset of `InstanceSpecification` elements present in the source model may need to be taken into account. In particular, when dependability information is applied to a given component, then its subcomponents are not taken into account.

To identify the component instances to be represented in the IDM model, two collection data structures are employed: "tocheck" and "idmcomp". The algorithm starts selecting the `InstanceSpecification` element specified in the *platform* attribute of the selected «`StateBasedAnalysis`» stereotype, and adding it to the "tocheck" collection.

Then, until "tocheck" is not empty, an element i is removed from the collection: if i, $component(i)$, or both have a DEP-UML stereotype applied to them, then i is added to the "idmcomp" collection; if no stereotypes are applied and $subs(i)$ is empty then i is still added to "idmcomp"; otherwise all elements in $subs(i)$ are added to "tocheck". When this procedure terminates, component instances contained in "idmcomp" are those that should be projected as IDM components.

This procedure selects all component instances such that i) their parents do not carry dependability information, and ii) they carry dependability information, or they do not have sub-instances. Once the components to be projected are identified, a `IDM::Component` element is created from each `UML::InstanceSpecification` contained in *idmcomp*.

4.3.2 *Projection of dependability templates*

In this phase we select from "idmcomp" all the elements i such that i or component(i) have a "dependability template" stereotype applied to them, i.e., «StatefulHardware», «StatelessHardware», «StatefulSoftware», or «StatelessSoftware».

Independently from the actual dependability template, the transformation creates the following IDM elements: an *InternalFault ft*, an *Error e*, a *FailureMode fm*, a *FaultsGenerateErrors fge*, a *ErrorsProduceFailure epf*, and a *RepairActivity ra*. Such elements are then associated to *Component c*, i.e., *ft* is added to *c.Faults*, *e* is added to *c.Errors*, *fm* is added to *c.FailureModes*, *fge* is added to *c.FaultsGenerateErrors*, *epf* is added to *c.ErrorsProduceFailure*.

Some parameters of these elements can be set independently of the actual dependability template, while others depend on the stereotype that is being processed. In particular, the duration of a transient fault, *ft.TransientDuration*, is always considered to be instantaneous and it is set to a *Deterministic* element *td* having *td.Value*=0.

For the *FaultsGenerateErrors fge* the following attributes are always set: *fge.Source=ft*, *fge.Destination=e*, *fge.PropagationLogic=ft*, *fge.PropagationProbability=1*, *fge.Weight=1*, *fge.ActivationDelay=Deterministic{0}*; i.e., error *e* is immediately generated from fault *ft*. Similarly, for the *ErrorsProduceFailure fge* relation, the following attributes are always set: *epf.Source=e*, *epf.Destination=fm*, *epf.PropagationLogic=e*, *epf.PropagationProbability=1*, *epf.Weight=1*.

Dependability templates assume that the repair of the component is started as soon as the component fails; therefore, *ra.When* is set to "*Immediately [Failed(fm)]*", where *fm* is the newly introduced failure mode, and *ra.Target* is set to *c*.

Other attributes of the newly introduced IDM elements are set based on the values of the faultOcc, probPermFault, errorLatency, and repairDelay attributes. All the "dependability template" stereotypes have a subset of these four attributes; in the following we then define the transformation rules that apply to these attributes, and the rule to apply in case the attribute is missing. In case the stereotype is applied to both i and component(i), only the instance-level attribute values (i.e., those specified on i) are taken into account.

FAULTOCC The faultOcc attribute specifies the fault occurrence rate of the component, which is assumed to follow an exponential distribution. Its value, if specified, is used to set the *Occurrence* attribute of the *InternalFault* element *ft*. More in detail, an element *fo* of type *Exponential* is created, with *fo.Rate* equal to the value of faultOcc; *ft.Occurrence* is then set to *fo*. If the value of faultOcc is not specified, then the elements *ft* and *fge* are removed from the IDM model, i.e., the component is not subject to internal faults, or their occurrence rate is negligible.

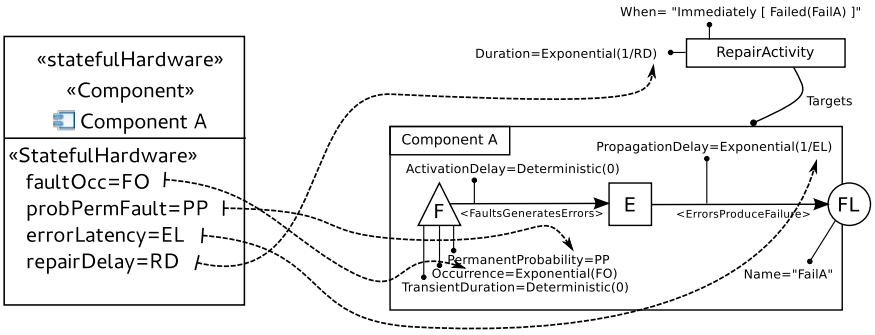


Figure 4.5: Projection of dependability templates in the IDM representation. The figure details the projection of the «StatefulHardware» stereotype.

PROBPERMFAULT The probPermFault attribute specifies the probability of permanent faults. The value of this attribute is used to set the value of the *ft.PermanentProbability* attribute. If the attribute is not present, or its value has not been specified, then the default value 0 is used.

ERRORLATENCY The errorLatency attribute specifies the error latency, i.e., the mean delay after which the presence of an error in the component leads to its failure. The value of this attribute is used to set the *epf.PropagationDelay* attribute. If the specified value is 0, then *epf.ProapagationDelay* is set to a *Deterministic* element *d* with *d.Value*=0. If the specified value is greater than zero, then *epf.PropagationDelay* is set to an *Exponential* element *el* having *el.Rate* equal to the inverse of errorLatency. If the value of this attribute is not specified, or it is not present in the stereotype, then the default value of 0 is used, i.e., an error immediately causes the component to fail.

REPAIRDELAY The repairDelay attribute specifies the delay after which the component, once failed, gets repaired. If its value is not specified then the component is never repaired; in this case the previously introduced *RepairActivity ra* is removed from the IDM model. If the attribute is not present in the stereotype (i.e., the involved stereotype is «StatelessSoftware»), then the value 0 is used, i.e., the component is immediately repaired. The value is used to set the *Duration* attribute of the *ra RepairActivity*. If the value is 0, then *ra.Duration=Deterministic{0}*, otherwise *ra.Duration* is set to an *Exponential* element *rd*, having *rd.Rate* equal to the inverse of the repairDelay attribute.

Based on these rules, the transformation performed for a component stereotyped with the «StatefulHardware» stereotype is depicted in Figure 4.5, using the graphical IDM notation.

4.3.3 *Projection of error model specifications*

In this phase we select from "idmcomp" all the elements i such that i or component(i) have the stereotype «DependableComponent» applied to them. Such components have their dependability properties specified through an UML::StateMachine stereotyped with the «ErrorModel» stereotype, which is referenced by the *errorModel* attribute. In this phase all the elements of the error model are projected, with the exception of «ExternalFault» elements, which are taken into account later, during the projection of propagation relations (Section 4.3.5).

The projection of error model elements is quite simple, since most elements have a similar representation in the IDM.

«Error»

For each DEP-UML «Error» euml contained in the error model, an *Error* element *e* is created in the IDM model. The *e.VanishingTime* attribute is set based on the euml.vanishingTime attribute. If euml.vanishingTime is specified as a numeric value, then *e.VanishingTime* is set to a *Exponential* element *vt*, having *vt.Rate* equal to the inverse of euml.vanishingTime. Otherwise, *e.VanishingTime* is set to an instance of the corresponding *Distribution* element. The newly created element *e* is then added to the *Errors* attribute of the IDM *Component* element corresponding to i.

«FailureMode»

Similarly, for each DEP-UML «FailureMode» fmuml contained in the error model, a *FailureMode* element *fm* is created in the IDM model. The newly created *fm* element is then added to the *FailureModes* attribute of the IDM *Component* element corresponding to i.

«InternalFault»

A DEP-UML «InternalFault» ftuml is a Transition element, connecting the initial PseudoState to a State euml stereotyped with «Error». For each of them, two elements are created in the IDM model: an *InternalFault ft*, and a *FaultsGenerateErrors fge*.

For the *InternalFault* element, the attributes ftuml.Occurrence, ftuml.permanentProb, and ftuml.transientDuration are directly mapped to IDM attributes *ft.Occurrence*, *ft.PermanentProbability*, *ft.TransientDuration*. For the *FaultsGenerateErrors* relation the following attributes are set: *fge.Source=ft*, *fge.ActivationDelay=Deterministic{0}*, *fge.PropagationProbability=1*, *fge.Weight=1*, *fge.Destination=e*, *fge.PropagationLogic=e*, where *e* is the IDM *Error* element corresponding to euml.

Both the introduced elements *ft* and *fge* are then added to the *Component* element corresponding to i, in the *Faults* and *FaultsGenerateErrors* attributes, respectively.

Internal propagation relations

The projection of internal propagation relations involves UML::Transition elements connecting «Error» elements with FailureMode elements, or Error elements with other Error elements.

For each Transition tr connecting a Error element with a FailureMode element, a *ErrorsProduceFailure epf* relation is created, and added to the *Component* corresponding to i. The attributes *epf.Source* and *epf.Destination* are set to the IDM elements corresponding to the Source and Target elements of the UML::Transition. The following attributes are then set: *epf.PropagationLogic=Source* and *epf.Weigth=1*.

If the Transition is stereotyped as «Propagation», the *PropagationProbability* and *epf.PropagationDelay* are set based on the values of the tr.prob and tr.propDelay attributes, respectively. If the transition is not stereotyped, then their are set to default values, which assume an instantaneous propagation with probability 1: *tr.PropagationProbability=1* and *tr.PropagationDelay=Deterministic{0}*.

For Transition elements connecting «Error» elements with «Error» elements a similar projection is performed, with the only difference that an *InternalPropagation* element is created in the IDM model.

4.3.4 *Projection of non-stereotyped components*

In this phase we select from "idmcomp" all the elements i such that neither i nor component(i) have a DEP-UML stereotype applied to them. These are component instances that do not have sub-instances and their parents do not carry dependability information. Nevertheless, these components need to be represented, sicne they can contribute to error propagation in case of failure of components interacting with them.

Such components are then projected as IDM components affected by a single failure mode, a single error, and *no internal faults*. Accordingly, for such components the following elements are created in the IDM model: a *Error e*, a *FailureMode fm*, and a *ErrorsProduceFailure epf*, and are added to the corresponding attributes of the *Component* corresponding to i.

For the *epf* element the following attributes are set: *epf.Source=e*; *epf.PropagationLogic=e*; *epf.Destination=fm*, *epf.PropagationProbability=1*; *epf.PropagationDelay=Deterministic{0}*; *epf.Weigth=1*.

4.3.5 *Projection of propagation relations*

Error propagation between component instances may take place essentially for three reasons:

- A component instance uses the service provided by another component instance. This kind of relation is modeled in DEP-UML through UML `Connector` elements connecting `ClientServerPort` elements.
- A component instance receives an input dataflow from another component instance. This kind of relation is modeled in DEP-UML through UML `Connector` elements connecting `FlowPort` elements.
- A software component instance is deployed on a hardware component instance. This kind of relation is modeled in DEP-UML through UML `Comment` elements stereotyped with the MARTE `Assign` stereotype.

Therefore, the algorithm iterates on all the the *InstanceSpecification* elements that are instance of a `UML::Connector` element, and all the «`Assign`» elements, within the *platform* specified in the considered «`StateBasedAnalysis`» element.

The iteration is repeated twice: in the first iteration *ExternalFault* elements are created for the involved components, in the second iteration *FaultsGenerateErrors* relations are created to specify how such faults propagate within components. Such approach is required to correctly transform DEP-UML external faults that occur as combination of multiple component failures.

Creation of IDM `ExternalFault` elements

If the element we need to transform is the instance of a `Connector` `c`, connecting two ports `a` and `b`, then the algorithm proceeds as follows. For each of the two connector ends, we check if the component instance owning the port has a representation in the IDM model. If not, then delegation links in its subcomponents are followed until a component instance having a representation in the IDM model is found. Based on the assumptions that we made, and previous transformation phases, such a component can always be found[1].

Once components *a_idm* and *b_idm* are found, corresponding to the IDM projection of components to which ports `a` and `b` are delegated, the transformation proceeds as follows. The direction of error propagation is obtained from the direction of the involved ports: propagation occurs from the "out" port to the "in" port in case of «`FlowPort`» elements, and from the "provided" port to the "required" port in case of «`ClientServerPort`» elements. The case of "inout" or

1 Actually, it could happen that the algorithm reaches a component instance that i) does not have an IDM representation, and ii) does not have subinstances. However, if such situation occurs, then it means that DEP-UML annotations exist on some component instance *owning* the involved connector; in this case the projection of the connector is not necessary and it is skipped.

"proreq" directions (i.e., bidirectional ports), is handled as if two pairs of ports with opposite directions existed in the model: two opposite propagation paths are created.

Let's assume that error propagation occurs from *a_idm* to *b_idm*. In this case, failures of *a_idm* are considered external faults for *b_idm*. If the InstanceSpecification element corresponding to *a_idm* has an error model, then the transformation creates an IDM *ExternalFault* element *xft* for each *FailureMode fm* that i) belongs to *a_idm*, and ii) in the error model was specified as affecting the involved port, through the affectedPorts attribute. The attribute *xft.Source* is then set to *fm*, and *xft* is added to the *b_idm.Faults* attribute. If *a_idm* does not have an error model, then an *ExternalFault* is created for the unique *FailureMode* element associated with *a_idm*.

The projection of propagation relations due to allocation is performed in a similar way. In this case the direction of propagation is from hardware to software, i.e., from the component instance specified in the to attribute of the MARTE::Assign element, to the component instance(s) specified in the from attribute.

Also in this case, propagation needs to be projected between two components that have a representation in the IDM model. Given an allocation relation of a software instance a on a hardware instance b, if a does not have a representation in the IDM model, then the propagation relation is projected as if each subcomponent of a was directly allocated on b. Similarly, if b does not have a representation in the IDM model, a propagation relation is created between a and each subcomponent of b.

Propagation due to allocation from the IDM component *a_idm* to the IDM component *b_idm* is projected in the same way as for the propagation due to connectors: for each *FailureMode fm* belonging to *a_idm*, an *ExternalFault* element *xft* is created, having *xft.Source=fm*. In this case, the presence of the error model has no influence.

Creation of IDM `FaultsGenerateErrors` *elements*

Given a propagation relation from IDM component *a_idm* to IDM component *b_idm*, the second iteration creates `FaultsGenerateErrors` model elements.

If the component "receiving" the propagation (i.e., *b_idm*, the one for which *ExternalFault* elements were created in the previous iteration) does not have an error model specification in the DEP-UML model, then any "incoming" external fault generates the same kind of error in the component, since only a single *Error* element has been created by the initial projection of *b_idm*. Accordingly, the transformation proceeds as follows. For each of the *ExternalFaults* elements *xft* in *b_idm* due to *a_idm*, a *FaultsGenerateErrors* element *fge* is created, having *fge.Source=xft*, *fge.PropagationLogic=xft*, and as *fge.Destination* the unique *Error* element present in *b_idm*.

Otherwise, if *b_idm* has an error model specification, a *FaultsGenerateErrors fge* relation is created for each «ExternalFault» Transition xftuml specified in the error model. The attribute *fge.Destination* is set to the «Error» State to which the transition is connected in the error model.

To set the fge.Source attribute, the xftuml.fromPort attribute is taken into account. In particular, all the IDM *ExternalFault* elements of *b_idm* originated from a propagation coming from the involved ports are added to *fge.Source*. The *fge.PropagationLogic* attribute is then set based on the value of the xftuml.propagationCondition attribute; if the attribute was not specified in DEP-UML, then the condition is the logical "OR" of all the faults in the *fge.Source* attribute.

In all cases, if the Connector or Assign elements originating the propagation is stereotyped as «Propagation», then the attributes *fge.ActivationDelay* and *fge.PropagationProbability* are set based on the respective propDelay and prob attributes. Otherwise, they are set to the default values of *fge.ActivationDelay=Deterministic{0}* and *fge.PropagationProbability=1*, i.e., immediate propagation with probability 1.

4.3.6 *Projection of activities*

After all the components, threats, and propagation relations are set in the IDM model, activities can be projected as well. The representation of activities in DEP-UML and in the IDM is similar, thus leading to a simple transformation process.

For each «Repair» activity rep, a *RepairActivity ra* is created in the IDM model. Its attributes *ra.Duration*, *ra.When*, *ra.SuccessProbability* are set based on the attributes rep.duration, rep.when, rep.probSuccess. Similarly, the targets of the repair activities, specified in *ra.Targets* are set to the corresponding IDM projection of elements specified in rep.targets.

Similarly, for each «ErrorDetection» activity umldet, an *DetectionActivity det* is created in the IDM model. Its attributes *det.Duration*, *det.When*, *det.SuccessProbability*, *det.CorrectionProbability* are set based on the attributes umldet.duration, umldet.when, umldet.probSuccess, umldet.correctionProbability. The attribute *umldet.DetectableError* is set to the collection of all the elements in *c.Errors*, where *c* is the *Component* corresponding to the component instance specified in the umldet.target attribute.

4.3.7 *Projection of analysis objectives*

The final step is the projection of analysis objectives, i.e., the measure of interest that should be evaluated by the analysis. This is specified using the «StateBasedAnalysis» stereotype and its attributes, as described in Section 3.5.8.

Given a «StateBasedAnalysis» element sba, a *Reliability* or *Availability* element m is created in the IDM model, based on the expression specified as measure attribute. The attribute *m.Evaluations* is set based on the parameters specified in the measure attribute.

The *m.Target* attribute is set based on the sba.targetDepComponent and sba.targetFailureMode attributes. The former specifies the target component instance(s) to be taken into account for the evaluation; the latter is used to specify specific failure modes to be taken into account. The *m.Target* attribute should refer to a collection of IDM *FailureMode* elements; such collection is identified in the following way.

First, sba.targetDepComponent is analyzed, and all the *FailureMode* elements in *c.FailureModes*, such that c is the IDM projection of a component instance specified in sba.targetDepComponent are collected. Then, if sba.targetFailureMode is specified, only *FailureMode* elements that are instance of «FailureMode» elements specified in sba.targetFailureMode are kept, while the others are removed.

4.4 FROM IDM MODELS TO STOCHASTIC PETRI NETS

This section describes the second macro-step for the automated generation of a dependability analysis models from UML models annotated with DEP-UML.

The class of Petri nets generated by our transformation algorithm are Stochastic Petri Nets as defined in [44], in which the firing delay of timed transition is specified by an arbitrary probability distribution. Moreover, we consider extensions of Petri net primitives that are commonly adopted in literature: arc multiplicities, inhibitor arcs, guards, transition priorities, marking-dependent arc multiplicities [47].

The description of the transformation does not follow the package structure of the IDM, since in the transformation different entities from different packages are related to each other, and the description may result difficult to follow. Rather, the description is organized in phases: i) projection of components and their threats; ii) projection of propagation relations; iii) projection of maintenance and monitoring activities; iv) projection of metrics of interest.

In the following, we use the expression MARK(p) to indicate the *marking* of place p, $\mu(p)$.

4.4.1 *Projection of components and threats*

In this phase the subnets corresponding to individual IDM components and their threats are created. Also, additional auxiliary elements are created, which are used to model the reset of the component to its initial state, e.g., in case of repair.

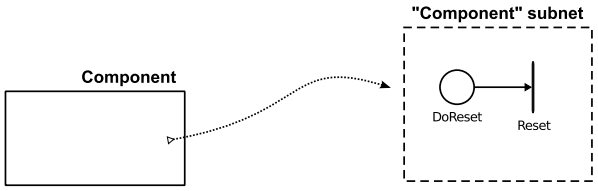


Figure 4.6: SPN elements generated from an IDM `Component` element.

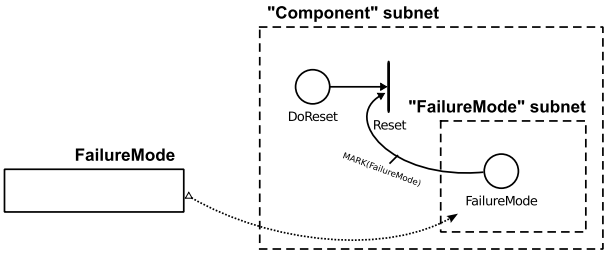


Figure 4.7: IDM to SPN transformation for "FailureMode" elements.

In particular, a place "DoReset" and an immediate transition "Reset" are created for each component (Figure 4.6). The "Reset" transition is used to reset the subnet corresponding to the component to its initial state, for example because the component has been repaired. The transition is enabled and fires when place "DoReset" contains a token. Arcs adding tokens to "DoReset" and additional input arcs to the "Reset" transition are added in subsequent phases of the transformation.

FAILUREMODE. The SPNs submodel associated with *FailureMode* elements is depicted in Figure 4.7. For each *FailureMode* element of the IDM model a new place is added, representing the model state in which the involved component has failed with such failure mode. In order to support the repair of the component, an input arc is also added, connecting the place corresponding to the *FailureMode* to the *Reset* transition of the component. The arc has a marking-dependent multiplicity, set to `MARK(FailureMode)`, i.e., it removes the exact number of tokens that are present in the "FailureMode" place (zero if it is empty). Such kind of arcs are also known as *reset arcs* [3, 46].

ERROR. The SPNs submodel associated with *Error* elements is depicted in Figure 4.8. The transformation of *Error* elements generates two places: "Error", which represents an erroneous state of the component, and "ErrorDetected", which holds a token if the error has been detected by the system. The timed transition "Vanishing" represents the delay after which an error may disappear from the component's state, and its time distribution is given by the *VanishingTime* attribute of the *Error* IDM element. This transition is created only if the optional *VanishingTime* attribute is set, otherwise it is assumed that an error

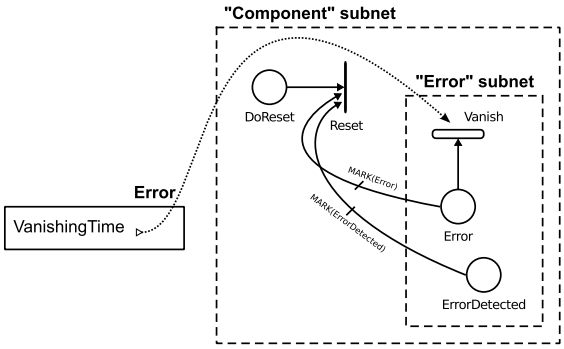


Figure 4.8: IDM to SPN transformation for "Error" elements.

may not disappear from the state of the component, until the component is repaired or replaced. Also in this case, two input arcs with marking-dependent multiplicity are added to connect "Error" and "ErrorDetected" places to the "Reset" transition of the component.

INTERNALFAULT. The SPNs submodel associated with *InternalFault* elements is depicted in Figure 4.9. The "FaultOccurrence" transition is a timed transition whose distribution is given by the *Occurrence* parameter of the *InternalFault* element. This transition is always enabled, unless the component has already failed; this condition is specified with a guard over places generated from component's failure modes.

When the "FaultOccurrence" transition fires it adds a token to place "FaultOccurred", which enables two instantaneous transitions, "Permanent" and "Transient", representing the occurrence of a permanent or a transient fault, respectively. Which of the two transitions will fire is probabilistically determined by their weights, which are set based on the value of the *PermanentProbability* attribute of the *InternalFault* element. When any of those two transitions fires it removes the token from place "FaultOccurred" and add a token to place "Fault", which represents the presence of a fault in the component; "Transient" also adds a token in place "TransientFault", which contains a token for each transient fault in the component. The transition "TransientDuration" represents the duration of a transient fault, and its time distribution is given by the attribute *TransientDuration* of the *InternalFault* element. When it fires it removes a token from place "Fault" and a token from place "TransientFault".

Places "Fault" and "TransientFault" are connected to the "Reset" transition in a similar way as described for *FailureMode* and *Error* elements.

EXTERNALFAULT. *ExternalFault* elements specified in the IDM model are not directly represented in the SPN model. When a reference to an element *xft* of type *ExternalFault* is encountered during the transformation (for example

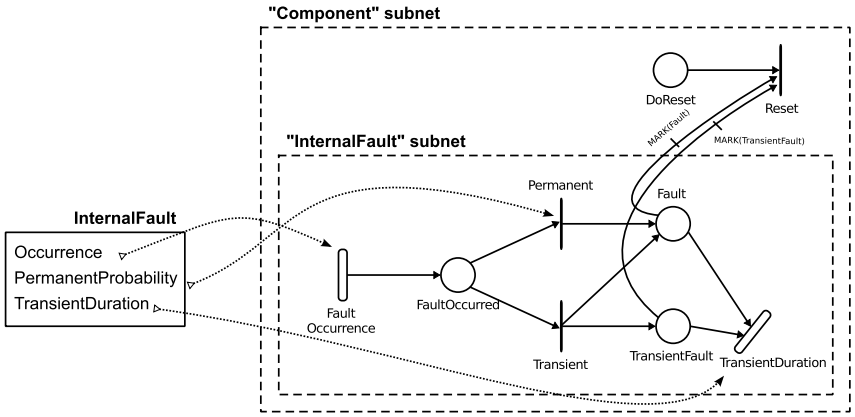


Figure 4.9: IDM to SPN transformation for "InternalFault" elements.

because it is referred by a *FaultsGenerateErrors* relation), then it is treated as a reference to the *FailureMode* element specified in *xft.Source*.

4.4.2 *Projection of propagation relations*

The IDM metamodel includes different kinds of propagation relations, namely *FaultsGenerateErrors*, *InternalPropagation*, and *ErrorsProduceFailure* relations. All those relations have similar attributes: a set of *Source* elements, a set of *Destination* elements, a *PropagationProbability* value, a *PropagationDelay* distribution, a *PropagationLogic* expression, and a *Weight* value. They only differ in the type *Source* and *Destination* elements that are allowed.

The SPN submodel generated from and *ErrorsProduceFailures epf* element is shown in Figure 4.10 and it is described in detail in the following; the transformations for the other propagation relations have the same structure. The "PropagationDelay" transition represents the delay after which error propagation occurs and its distribution is given by the *epf.PropagationDelay*; the enabling condition for this transition is provided by a guard expression, which is derived from the *epf.PropagationCondition*. When "PropagationDelay" fires it adds a token in place "Propagation". The token in place "Propagation" enables the two immediate transitions "Propagate" and "NoProp", which model the probability that propagation actually takes place; their weight is given by the "PropagationProbability" attribute of the relation. If propagation occurs (i.e., transition "Propagation" fires) a token is added to the place associated with each of the *FailureMode* elements referenced in the *epf.Destination* attribute.

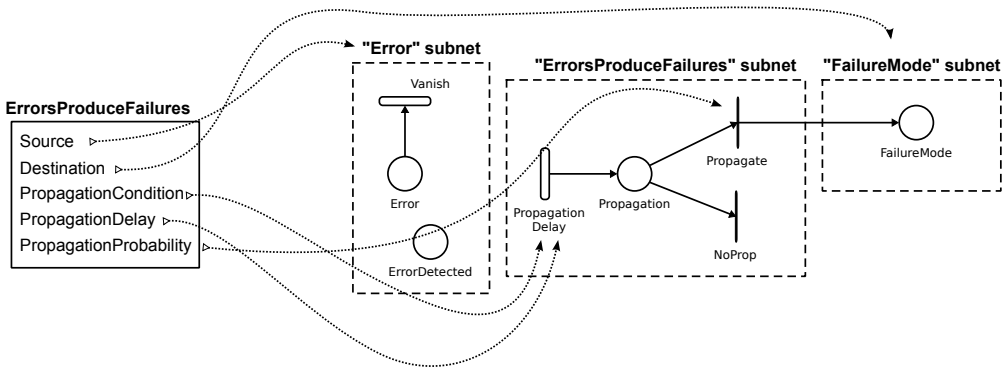


Figure 4.10: IDM to SPN transformation for "ErrorsProduceFailures" elements.

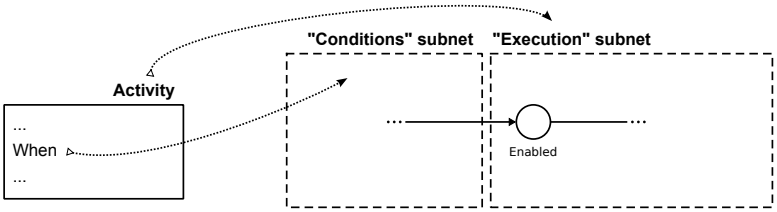


Figure 4.11: Structure of transformation rules for "Activity" elements of the IDM metamodel.

4.4.3 *Projection of activities*

The projection of an activity (i.e., an element of type *Activity* or one of its subtypes) can be thought as composed of two tasks:

i. Projection of the conditions that enable the activity to fire. This information is provided by the *When* attribute of the *Activity* element.

ii. Projection of the actual execution of the activity, and its effects on the system. This information is provided by the attributes that are specific of each activity type, and by the activity type itself.

Accordingly, the resulting SPNs model is composed of two subnets: the "Conditions" subnet that models the conditions needed for the execution of the activity, and the "Execution" subnet, which models the execution of the activity (or its failure), and its effects on the system. The interface between these two subnets is the "Enabled" place, in which the "Conditions" subnet adds a token if the enabling conditions are met.

The details of the "Conditions" subnet depend on the expression specified in the *When* field of the *Activity* element, while the details of the "Execution" subnet depends on the particular subtype of *Activity* that is involved.

Conditions

Conditions that enable a given activity to fire are specified in the *When* attribute, using the grammar defined in Listing 3.2. Conditions specified using such grammar are composed of three parts:

<T> CONDITION This part specifies at which instant (or instants) of time the activity should be executed, and the delay between one execution and the other. It is projected as a transition "Trigger" in the "Conditions" subnet, which adds a token in the "Enabled" place of the "Execution" subnet (Figure 4.12).

Depending on the actual element some differences in the projection occur:

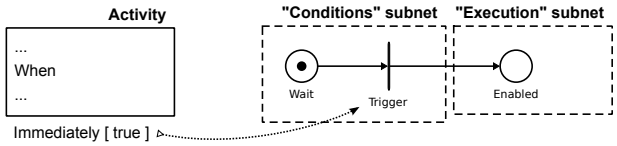
- *Immediately.* In this case "Trigger" is an immediate transition, and it is allowed to fire only once (Figure 4.12a).
- *AtTime(<RealNumber>).* In this case the "Trigger" transition is a timed transition, with a deterministic firing delay set to the value of <*RealNumber*> (Figure 4.12b). Also in this case the transition is allowed to fire only once.
- *Periodic(<Distribution>).* In this case the "Trigger" subnet is timed transition, and its firing delay is set based on the <*Distribution*> parameter. The transition is allowed to fire multiple times, since once fired it adds a new token in place "Wait" (Figure 4.12c).

The guard of the "Trigger" transition is given by the <EX> part of the expression, whose projection is described in the following.

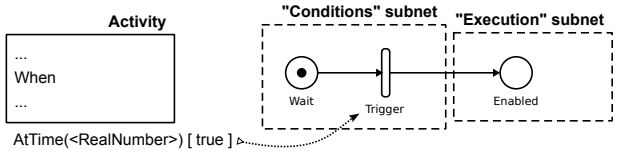
<EX> CONDITION This part specifies a Boolean predicate that must hold in order for the activity to be enabled. Such predicate is used in the transformation to build the guard of the "Trigger" transition described above. Atomic predicates are the following:

- *Failed(<FailureMode>)*, which holds if the given failure mode has occurred. This predicate is translated to the expression "MARK(FailureMode_x)>0", where "FailureMode_x" is the place corresponding to the *FailureMode* element that has been specified in the IDM condition.
- *Detected(<Error>)*, which holds if the given error has been detected. This predicate is translated to the expression "MARK(ErrorDetected_x)>0", where "ErrorDetected_x" is the "ErrorDetected" place corresponding to the *Error* element that has been specified in the IDM condition.

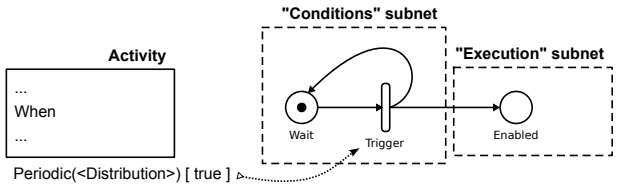
<L> CONDITION This part is optional and specifies a condition that limits the execution of the given activity to specific intervals of time.



(a) "Immediately" nodes



(b) "AtTime" nodes



(c) "Periodic" nodes

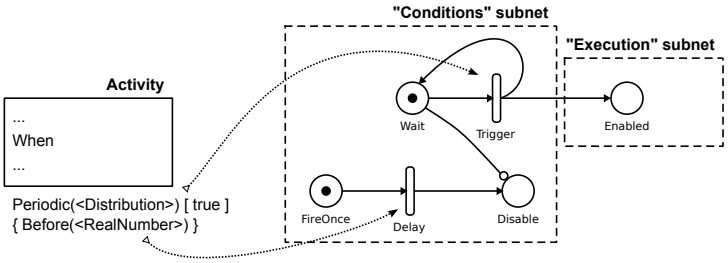
Figure 4.12: IDM to SPN transformation for <T> conditions.

In the SPN model this condition is modeled by adding a place "Disable" to the "Conditions" subnet, and adding an inhibitor arc that connects the place to the "Trigger" transition described above. Based on the actual value of the <L> condition, the token in place "Disable" is added or removed at precise instant of times (Figure 4.13). A <L> condition can take the following values:

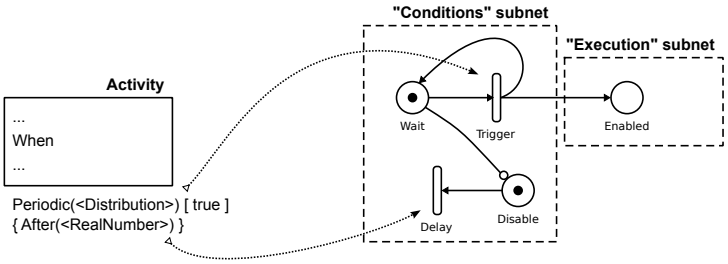
- *Before(<RealNumber>)*, which means that the activity is enabled only before the instant of time that is specified. In this case the "Disable" place is initially empty. An additional timed transition "Delay" is added to the "Conditions" subnet, having a deterministic firing delay equal to the value of the *<RealNumber>* parameter. When this transition fires, it adds a token in place "Disable", thus disabling the "Trigger" transition (Figure 4.13a).

- *After(<RealNumber>)*, which means that the activity is enabled only after the instant of time that is specified. In this case the "Disable" place initially contains one token. An additional timed transition "Delay" is added to the "Conditions" subnet, having a deterministic firing delay equal to the value of the *<RealNumber>* parameter. When this transition fires, it removes the token from the "Disable" place, thus enabling the "Trigger" transition. The transition is enabled to fire exactly once, because of the "FireOnce" place (Figure 4.13b).

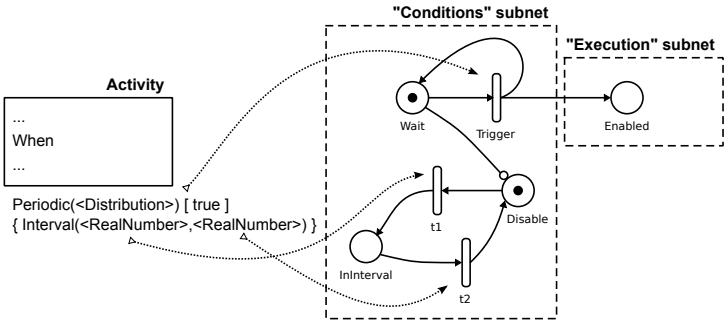
- *Interval(<RealNumber>,<RealNumber>)*, which means that the activity is enabled only in a specific interval of time. Also in this case the "Dis-



(a) "Before" nodes



(b) "After" nodes



(c) "Interval" nodes

Figure 4.13: IDM to SPN transformation for <L> conditions.

able" place initially contains one token. Two additional timed transitions "t1" and "t2" are added, both having a deterministic firing delay. A place named "InInterval" is also added to the "Conditions" subnet. The firing delay of "t1" is set to the difference between the second and the first <*RealNumber*> parameter, i.e., the duration of the specified interval. When "t1" fires it removes the token from the "Disable" place and adds one token to the "InInterval" place; the "Trigger" transition becomes therefore enabled. The firing delay of "t2" is set to the value of the second <*RealNumber*> parameter. When "t2" fires it removes the token from the "InInterval" place and adds a token to "Disable", thus disabling the "Trigger" transition (Figure 4.13c).

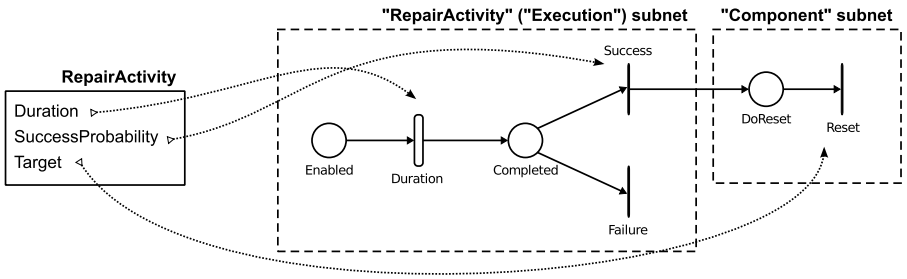


Figure 4.14: IDM to SPN transformation for "RepairActivity" elements.

Execution

The details concerning the execution of an activity and its possible effects on the system depend on the type of the activity. In particular, the following kinds of activities are supported by the current version of the transformation workflow.

REPAIRACTIVITY The *RepairActivity* element is used to model maintenance activities on the system (both preventive and corrective). The "Execution" subnet of a *RepairActivity* element contains a timed transition "Duration", which represents the time required to physically execute the repair, including the time required to call the support personnel, if required.

The "Duration" transition is enabled when a token is in place "Enabled", and when it fires it removes the token from "Enabled", and adds a token in "Completed". The distribution of the firing delay is given by the *Duration* attribute of the "Repair" element. The token in place "Completed" enables the two immediate transitions "Success" and "Failure", which represent the success and failure of the repair, respectively. The weight of the two transitions is given by the *SuccessProbability* attribute. If the repair succeeds, a token is added in the "DoReset" place of each "Component" referenced in the *Target* field of the *Repair* element (Figure 4.14). The firing of the "Reset" transition then resets the subnet related to the component to its initial healthy state.

ERRORDETECTION The *ErrorDetection* element is used to model error detection activities on the system. The "Execution" subnet of an *ErrorDetection* activity *ed* is similar to the one for the *RepairActivity* model element. The success and failure of the activity, and the weights of the immediate transitions in this case are provided by the *ed.Coverage* attribute.

For each *Error e* in *ed.DetectableErrors*, an output arc connecting "Success" and the place "ErrorDetected" corresponding to *e* is created. Such output arc has a marking-dependent multiplicity, `MARK(Error)`, where `Error` is the "Error" place corresponding to *e*. In this way, the token to the "ErrorDetected" places is added only when an error is actually present in the component.

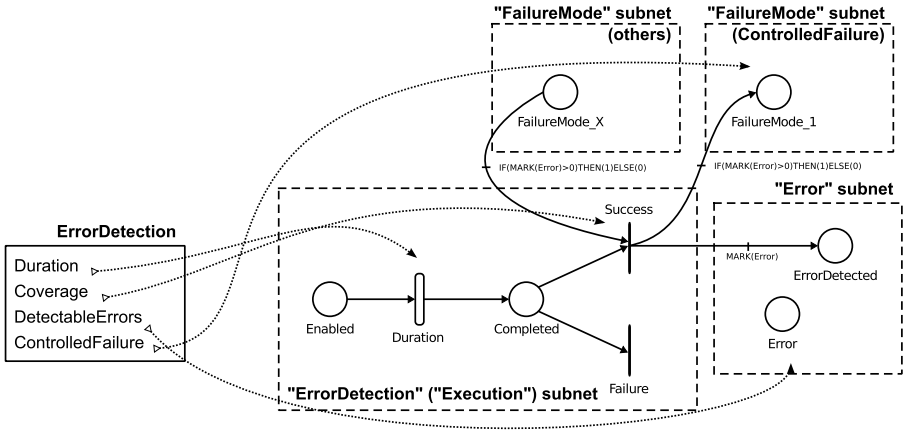


Figure 4.15: IDM to SPN transformation for "ErrorDetection" elements.

The optional attribute *ControlledFailure* can be used to specify a *FailureMode* in which the *Component* is driven if an error is detected (e.g., a safe failure mode). If this attribute is specified, then additional arcs are added in the SPN model. In particular, if *FailureMode_1* is the *FailureMode* element referenced by the *ControlledFailure* attribute, an output arc is added between the "Success" transition of the activity "Execution" subnet and place "FailureMode_1" corresponding to such failure mode. Also, an input arc is added between place "FailureMode_X", corresponding to other failure modes of the involved component. All these arcs have the following marking-dependent multiplicity:

IF(MARK(Error1)>0 OR ... OR MARK(ErrorN)>0)THEN(1)ELSE(0)}

where `Error1 ... ErrorN` are "Error" places corresponding to the *Error* elements specified in *ed.DetectableErrors*. In this way, the component is placed in the specified failure mode only when one of the specified errors is detected.

REPLACEACTIVITY The *ReplaceActivity* element is used to model the replacement of a component, i.e., a particular case of repair in which the component is substituted with another one, specified with the *Replacement* attribute, possibly with different dependability characteristics.

A possible way to support such element is to generate a SPN subnet similar to the one that is generated for *RepairActivity* elements, with the only difference that, in case of success, a token is added also to an additional "BeenReplaced" place (Figure 4.16). The rest of the transformation would then replace each parameter in the "Component" subnet related to the *Component* specified in the *Target* attribute with a marking-dependent expression based on the marking of place "BeenReplaced". The expression would return the parameter value related to *Target* if "BeenReplaced", and the value related to *Replacement* if "Been-

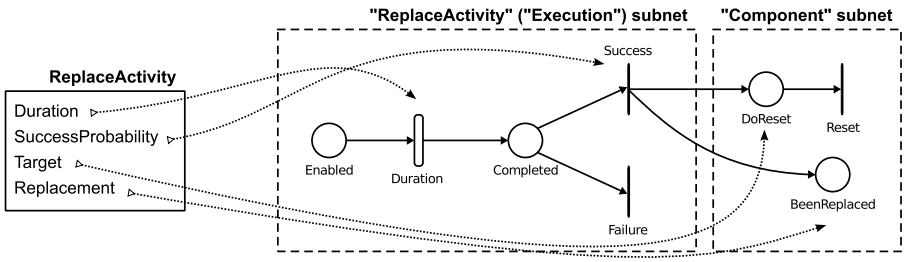


Figure 4.16: IDM to SPN transformation for "ReplaceActivity" elements.

Replaced" contains one token. In this way the replacement of a component with one with different parameters can be properly modeled.

4.4.4 *Projection of analysis objectives*

In the IDM, the metrics to be analyzed on the model are defined through subtypes of the *DependabilityMeasure* abstract model element.

The *Target* attribute of the *DependabilityMeasure* model element references a set of *FailureMode* elements, which correspond to a set of places in the SPN model. Such failure modes are those that should be taken into account in the evaluation of the specified measure, i.e. the system is considered failed if (at least) one of them has occurred.

For all the *DependabilityMeasure* elements (i.e., *Reliability*, *Availability*, and *Safety*), the measure of interest is obtained by defining a reward function of the state of the generated SPN model. The construction of the reward function is based on the *Target* attribute of the *DependabilityMeasure* element, and it is described in the following. Based on the type of measure specified in the IDM model, instant-of-time, interval-of-time, or steady-state analysis is then specified.

AVAILABILITY For *Availability* model elements the reward function is defined as a function that return 1 if all the places corresponding to *FailureMode* elements specified in *Target* are empty, i.e., the system is in a state which is free from the specified failure modes.

RELIABILITY The *Reliability* metric evaluates the probability that a specific failure mode, or set of failure modes have not occurred *until* time t. For this reason, the evaluation of this metric needs the introduction of an ad-hoc additional place, which is used to record if the specified failure modes have occurred at least once in the system lifetime. Such place, "SystemFailed", is connected to an immediate transition, "SystemFailure", which adds a token to it when the specified condition occurs, i.e., when at least one of the places corresponding to *FailureMode* elements specified in *Tar-*

get are not empty. Multiple firing of the transition are avoided by adding an inhibitor arc connecting it to the "SystemFailure" place.

SAFETY For what concerns quantitative evaluation, the *Reliability* and *Safety* model elements are projected in the same way: in this context, safety is evaluated as the reliability with respect to specific failure modes (i.e., catastrophic failure modes).

4.4.5 *Priorities and additional constraints*

Finally, to ensure the correct behavior of the generated model, additional constraints are imposed. In particular, we need to i) prevent the model from entering markings that could lead to *infinite sequences of vanishing marks*, through infinite firings of immediate transitions; and ii) ensure the correct ordering of events. This is achieved by setting different priorities to different kinds of immediate transitions in the model; we use priority values ranging from 8 (highest) to 1 (lowest).

The highest priority (8) is set to "SystemFailure" transitions, i.e., transitions associated with *Reliability* or *Safety* IDM elements. In this way the "recording" of the occurrence of a system failure is performed before any repair transition is triggered. Priority 7 is set to "Propagate" and "NoProp" transitions of propagation relations (Figure 4.10); the propagation process is therefore completed before any other actions (e.g., repairs) are performed.

Priority 6 is associated to "Reset" transitions of components, while priority 5 is associated to those "PropagationDelay" transitions which are immediate transitions. In this way, the firing of further propagation transitions in components that are going to be instantaneously repaired is avoided. Priority 4 is associated with immediate "TransientDuration" transitions; since they have a lower priority with respect to "PropagationDelay" transitions, transient faults whose duration is assumed to be immediate are still propagated as errors within the component.

Priority 3 is associated with the "Success" and "Failure" transitions of activities "Execution" subnet (e.g., see Figure 4.14); while priority 2 is set to "Duration" transition of activities, in case they are immediate transitions. This ensure that the outcome of an activity (i.e., success or failure) is selected before further executions on the same activity. All the other immediate transitions in the model have priority 1.

Finally, an additional condition is added to guards of *Reset* transitions associated with components. The guard constrains the activity to fire only when i) no external faults are affecting the component, i.e., when the corresponding "FailureMode" places are empty, or ii) the component causing such external faults is going to be reset as well (i.e., a token is present in its "DoReset" place). This condition is necessary in order to avoids loops of immediate transitions in

situations where a component gets repaired with zero delay (e.g., «`Stateless-Software`» components), but at the same time it is affected by instantaneous error propagation from another system component. Moreover this condition models the aspect for which, once failed, a component cannot become fully functional again until all the service it requires are restored.

5 IMPLEMENTATION WITHIN THE ECLIPSE PLATFORM

The overall CHESS methodology is implemented as an UML2 profile and a set of plugins for the Eclipse platform [70, 80]. The CHESS toolset includes a diagram editor based on a customized version of Papyrus [82], and a set of plugins to perform code generation, model-transformation, and different kinds of analyses.

In this chapter we describe the implementation of the Eclipse plugin for quantitative dependability analysis, which implements the model-transformation algorithms described in Chapter 4. Moreover, the plugin automatically generates the input for the analysis tool, evaluates the specified metrics, and reports the results back in the original CHESS ML model.

While the toolchain originated within the CHESS project, in its design we focused on modularity and reusability, in order to be able to adapt the resulting toolchain in other contexts. This is a fundamental aspect of developing a model-transformation toolchain, as the MDE world is constantly evolving, with new models, languages, and tools being constantly introduced and refined.

The process of defining a reusable toolchain architecture is described in Section 5.1, while Section 5.2 describes its implementation within the CHESS framework. The plugin is available on the CHESS website [35]. Its usage is illustrated in a demonstration video [41], and within a tutorial that guides the user through the application of the analysis plugin on a CHESS ML model [133].

5.1 DESIGNING A REUSABLE TOOLCHAIN

The abstract toolchain architecture proposed in this section targets tools for the automatic execution of non-functional analysis of system architecture specified in UML-like modeling languages, with back-annotation of results. From a high level perspective, the plugin should be able to automatically i) generate an analysis model which complies with the architectural description of the system, ii) analyze the model for the specified metrics of interest, and iii) propagate the obtained results back in the architectural description of the system that has been received as input.

Additionally, our objective was to create a plugin which as much reusable as possible, in order to adapted with reduced effort to other contexts. For this reason, the plugin has also specific requirements concerning reusability. Toolchain requirements are summarized in Table 5.1, grouped in the "functionality" and "reusability" categories.

Table 5.1: Identified requirements for a toolchain performing non-functional analysis on a system architecture specified in a UML-like language.

	Functionality
F1	The plugin should take as input a description of the system architecture in a UML-like language, including the extra-functional properties needed for the analysis, and a set of metrics to be evaluated.
F2	If the input model contains all the necessary information, the plugin should be able to generate a stochastic model of the system for the evaluation of the specified metrics.
F3	The plugin should be able to analyze the generated model using external tools.
F4	The plugin should be able to extract the results from the analysis tool, and propagate them into the architectural model that was received as input.
	Reusability
F1	The plugin should take as input a description of the system architecture in a UML-like language, including the extra-functional properties needed for the analysis, and a set of metrics to be evaluated.
F2	If the input model contains all the necessary information, the plugin should be able to generate a stochastic model of the system for the evaluation of the specified metrics.
F3	The plugin should be able to analyze the generated model using external tools.
F4	The plugin should be able to extract the results from the analysis tool, and propagate them into the architectural model that was received as input.

The designed architecture for the analysis plugin is sketched in Figure 5.1 and it is described in the following. For higher flexibility the toolchain is divided into a client and a server process, communicating through a TCP/IP network. Of course, the server and client processes may reside on the same physical machine as well.

5.1.1 *Architecture Overview*

The client has the responsibility of performing the required model transformations in order to i) generate the analysis model in a format readable by the tool, and ii) perform the back-annotation of analysis results.

The *client process* takes as input an architectural model from the design environment, and it generates the analysis model targeted to a specific analysis tool. The model is then transmitted to the server process, which executes the analysis tool and forwards the results back to the client. The obtained results are then back-annotated into the original architectural model that has triggered the analysis by the client process.

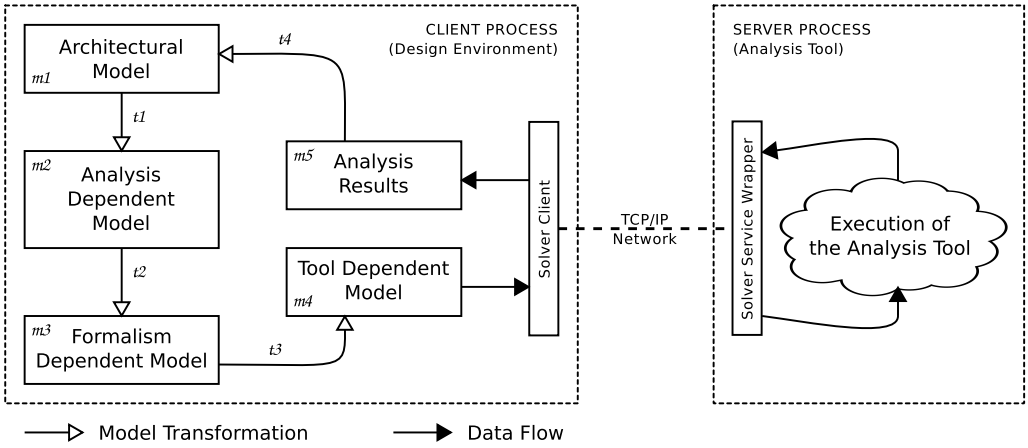


Figure 5.1: Abstract toolchain architecture for automated dependability analysis. Labels m1...m5 indicate the involved models in the toolchain, while labels t1...t4 indicate model transformation steps. For greater flexibility, the workflow is divided into a client and server process, which may however reside on the same physical machine as well.

The *server process* has the task of actually executing the analysis tool on the model generated by the client process, wait for results, and communicate the results back to the client process. Having a separate process for executing the analysis tool has a number of important advantages.

First of all, it allows the user not to be bound to the platform required by the selected analysis tool. Complex analysis tools often require a specific environment in order to work properly, i.e., specific operating systems, libraries, or tuning of system configurations. In such way, the analysis tool can be installed on a properly configured ad-hoc machine (possibly even a virtual machine), while the final user can continue to use its favorite environment for design purposes. Second, it lowers the hardware requirements of the user machine, by making it possible to move model evaluation, which is typically a resource-intensive task, to a dedicated machine. At the same time, this approach does not not prevent setting up a local-only configuration, where the analysis tool is run on the same machine as the client. Finally, this approach facilitates the management of licensing issues; for example it is possible to distribute the code of the frontend as open source, even if the backend relies on some proprietary tool which cannot be redistributed.

The transformation chain within the client process involves the use of five different metamodels (m1...m5), and four model-transformation algorithms (t1...t4). Adopting a multi-step transformation process improves the reusability of different portions of the toolchain. The role of each of these elements in the abstract toolchain is described in the following.

5.1.2 *Client Process – Metamodels*

Architectural Model (m1)

An m1 model contains an architectural description of the system in some kind of system engineering modeling language (e.g., UML, SysML). The model should contain all the information that is needed to perform the analysis, e.g., by using some ad-hoc extension to the a general purpose modeling language, or by using a domain-specific language that is able to represent all the required information. Typically, at this level the model contains a large number of details that are unnecessary for the analysis, which are expression of different concerns of different stakeholders.

This metamodel can be reused in toolchains that use the same architectural language to describe the system architecture.

Analysis-Dependent Model (m2)

An m2 model contains all the information that is necessary and sufficient to perform the intended analysis, filtered with respect to the original engineering model, and organized in a convenient way to facilitate the subsequent model-transformation steps. For example, when performing dependability analysis, only dependability-related information is retained; similarly, for performance analysis, the metamodel should be able to describe performance-related information.

The IDM described in Section 4.2 is a m2 model tailored to quantitative dependability analysis. The KLAPER language of [81] is another example of m2 model developed in literature. An m2 metamodel could be reused in developing toolchains that perform, possibly in different contexts, the same kind of analysis.

Formalism-Dependent Model (m3)

An m3 model contains an implementation of the analysis model in the formalism that has been selected for the analysis (e.g., Stochastic Petri Nets, Fault Trees, PEPA...). At this level, the model is however still an abstract representation and it is not yet bound to any specific analysis tool. Formalism-dependent models exist in literature, often in the form of "interchange format" based on XML, e.g., the Performance Model Interchange Format (PMIF) [170], an interchange format for Queuing Networks models, and the Petri Nets Markup Language (PNML) [90], an interchange format for Petri net models.

This metamodel can be reused in toolchains that employ the same formalism for performing the target analysis technique. It should be noted that this metamodel can be reused even if the *kind* of analysis is different; for exam-

ple, Stochastic Petri Nets are used both for dependability and for performance analysis.

Tool-Dependent Model (m4)

An m4 model is the concrete analysis model in a format specifically tailored to the selected analysis tool. Typically, this model is a file that can be directly provided as input to the tool.

This metamodel can be reused in toolchains that use the same tool to perform the analysis. It should be noted that it may be possible to reuse this metamodel even if a different formalism is used for the analysis. It may be the case for example when multi-formalism tools (e.g., Möbius [77]) are used for the evaluation.

Analysis Results (m5)

An m5 model is a model describing the results provided by the analysis tool, and it is used for the back-annotation process.

As above, when using multi-formalism tools, this metamodel can be reused even if the adopted formalism varies. Moreover, it can be reused even when different tools are used for the analysis, but use some standard format for the produced output (e.g., CSV, XML).

5.1.3 *Client Process – Transformations*

Filtering (t1)

The first model-transformation has the task of filtering out the information required for the analysis from the mass of information that is typically present in the architectural model of the system. Usually, this is the most complex algorithm of the entire chain, since it requires to navigate the entire engineering model, which may consist of several different diagrams, and relate concepts that refer to the same system entities. This model-transformations is applied on m1 models to generate m2 models.

Analysis Model Implementation (t2)

The second transformation implements the analysis model in the selected analysis formalism. The definition of this model-transformation step requires an expert in the selected analysis technique, and knowledge of the analysis formalism. This model-transformation is applied on m2 models to generate m3 models.

Code Generation (t3)

The third model-transformation has the task to generate the actual input file needed for the analysis tool. For this reason, this step is typically more oriented towards code generation rather than model-transformation, since the final goal is to generate a source file that should be read by the adopted analysis tool. This model-transformation is applied on m3 models to generate m4 models.

Back-Annotation (t4)

The back-annotation is a particular kind of model-transformation that has the task to propagate the results of the analysis back into the model that has triggered it. This transformation takes as input also the original model, which is refined with the new information, i.e., it is applied on a (m1,m5) pair of models, to generate a modified m1 model.

5.2 THE "STATE-BASED ANALYSIS PLUGIN"

The abstract architecture described in Section 5.1 has been concretely implemented within the CHESS project to realize the CHESS "Plugin for State-Based Analysis" (CHESS-SBA), by instantiating all the elements described in Figure 5.1.

CHESS ML (with DEP-UML extensions) is used as m1 language; the IDM of Section 4.2 as m2, and PNML as m3. The adopted analysis tool is the discrete-event simulator provided with the DEEM tool [20, 137]. The tool receives as input a "DEEM Input File" (m4) and produces a "DEEM Results File" (m5) having a specific format. Table 5.2 summarizes how the individual elements of the abstract toolchain have been realized within CHESS.

5.2.1 *Client Process*

The client process is written in Java, as a plugin for the Eclipse platform [80]; therefore, it can be used on any platform for which Eclipse is available.

CHESS ML (m1)

For the "Architectural Model" level the adopted language is the CHESS ML language developed within the CHESS project, which includes the DEP-UML extensions of Section 3.5. The full specification of the CHESS ML language is available in [36].

Table 5.2: The elements constituting the abstract toolchain, and their implementation in the CHESS plugin.

Abstract Toolchain Element		CHESS-SBA Implementation
Client Process		Java (Eclipse Plugin)
Architectural Model	$\mathtt{m1}$	CHESS ML
Analysis-Depenent Model	$\mathtt{m2}$	IDM
Formalism-Depenent Model	$\mathtt{m3}$	PNML
Tool-Depenent Model	$\mathtt{m4}$	DEEM Input File
Analysis Results	$\mathtt{m5}$	DEEM Results File
$\mathtt{m1} \longrightarrow \mathtt{m2}$	$\mathtt{t1}$	ATL Module
$\mathtt{m2} \longrightarrow \mathtt{m3}$	$\mathtt{t2}$	ATL Module
$\mathtt{m3} \longrightarrow \mathtt{m4}$	$\mathtt{t3}$	ATL Query
$\mathtt{m4}, \mathtt{m1} \longrightarrow \mathtt{m1}$	$\mathtt{t4}$	Java
Server Process		Java (Standalone)
Analysis Tool		DEEM Simulator

IDM (m2)

For the "Analysis-Dependent Model" level, we adopted the Intermediate Dependability Model (IDM) of Section 4.2. The IDM has been defined exactly with the purpose of being an intermediate language to support model transformation for quantitative dependability analysis.

PNML (m3)

The Petri Net Markup Language (PNML) [90] is a proposal for a Petri net interchange format based on XML that is under development as an ISO/IEC standard. ISO/IEC 15909 [97, 98] aims to provide a standard for the representation of Petri Nets models, and it is organized in three parts, describing: 1) formal definitions and graphical notations, 2) the transfer format (i.e., the concrete PNML language), and 3) Petri net types and extensions. Such characteristics make PNML a good choice for implementing the m3 metamodel in our toolchain: it is specific of the formalism selected for the analysis (i.e., Petri nets and their extensions), but it is not tailored to any specific analysis tool.

However, to date only Part 1 [97] and Part 2 [98] of the standard have been published, while Part 3 has not been disclosed yet. Part 2 defines the PNML language and the way to represent basic *non-timed* P/T Petri nets, as well as the extension mechanisms to attach additional properties to Petri net elements (e.g, the firing distribution of transitions). The actual standardized extensions to support different classes of Petri nets will however be included in Part 3 of the standard, and are not available yet.

As such, an ad-hoc PNML extension has been defined to represent the class of Stochastic Petri Nets needed for our analysis, which adds to the basic P/T Petri

nets defined in the standard the following features, that we use in the analysis model generated by our transformation algorithm: i) timed transitions; ii) inhibitor arcs; iii) priorities for immediate transitions; iv) weights for immediate transitions; v) guards; vi) marking-dependent model parameters; vii) metrics of interest for the evaluation.

As a simple example, the PNML code corresponding to a Stochastic Petri Net composed of two places and an exponential transition with rate 10.0 is shown in the listing below.

Listing 5.1: *PNML example.*

```
<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">
  <net id="n1" type="http://www.pnml.org/version-2009/grammar/ptnet">
    <name><text>pnml_example</text></name>
    <place id="p1">
      <name><text>p1</text></name>
      <initialMarking><text>1</text></initialMarking>
    </place>
    <place id="p2">
      <name><text>p2</text></name>
      <initialMarking><text>2</text></initialMarking>
    </place>
    <transition id="t1">
      <name><text>t1</text></name>
      <exponential><rate>10.0</rate></exponential>
    </transition>
    <arc id="a1" source="p1" target="t1">
      <inscription><text>1</text></inscription>
    </arc>
    <arc id="a2" source="t1" target="p2">
      <inscription><text>1</text></inscription>
    </arc>
  </net>
</pnml>
```

DEEM Input File (m4)

A "DEEM Input File", which is used as the m4 model, is essentially a text file composed of different sections, containing (in the following order): i) the header, which is almost fixed for any input file; ii) the definition of variables to be used in the study definition; iii) the studies to be performed on the model, i.e., the combination of different values for the variables specified above; iv) the list of places; v) the list of transitions; vi) the list of arcs; vii) the list of measures of interest that should be evaluated.

DEEM Results File (m5)

Similarly to the input file, the "DEEM Results File" is also a text file, containing the results of the evaluation. More in detail, the file contains: i) an header, which is almost fixed, ii) the parameters that have been used to run the analysis, including model parameters and simulator parameters, iii) the time for which the analysis has been run, and finally iv) a set of evaluated metrics. For each metric the tool provides its mean, as well as the confidence interval, and the number of values on which it has been computed.

CHESS ML → IDM (t1)

The transformation algorithm for generating IDM models from DEP-UML (or CHESS ML) models has been defined in Section 4.3. This transformation step has been implemented as a "Module" in the ATLAS Transformation Language (ATL) [5].

IDM → PNML (t2)

The transformation algorithm for generating SPNs models from IDM models has been defined in Section 4.4. This transformation step is implemented as a "Module" in the ATL language as well, using an Ecore metamodel of PNML as target language.

As an example, the listing below describes the ATL implementation of the rule for projecting IDM `Component` elements (Figure 4.6). From an IDM `Component` element, three PNML elements are generated: a place, an immediate transition, and an arc connecting the two.

Listing 5.2: *ATL implementation of the rule for projecting IDM components (see Section 4.4.1).*

```
rule Component {
  from
    c : IDM!Component
  to
    doreset : PNML!Place
    (
      id <- c.Name + '_doreset',
      initialMarking <- thisModule.newPTMarking(0)
    ),
    reset : PNML!GSPNImmediateTransition
    (
      id <- c.Name + '_reset',
      Priority <- thisModule.priReset,
      Weight <- 1.0
    ),
    arcreset : PNML!GSPNArc
    (
      id <- c.Name + '_arcreset',
      source <- doreset,
```

```
        target <- reset
    )
}
```

*PNML → DEEM Input File (*t3*)*

The DEEM Input File is generated from the PNML model by looping through the list of places, transitions, and arcs, and generating a string for each of them. While the first three sections of the file (header, variables, and studies) are mostly fixed in the current implementation, the others are generated based on the content of the PNML model as follows.

PLACES For each place in the PNML model the following string is generated in the DEEM model:

```
PLACE:"N",X,Y,M,C;
```

where N is the name of the place, X and Y its coordinates for a possible graphical representation, M the initial marking, and C the maximum capacity (i.e., the maximum number of tokens that the place is allowed to contain).

TRANSITIONS In the DEEM input format transitions are specified as:

```
TRANS:"N",X,Y,D,T,P,DIST,'COND','V1',
(
  V
);
```

where N is the name of the transition, X, Y and D are parameters for a possible graphical representation, T is the type of the transition (i.e., "Immediate" or "Timed"), P is the priority for immediate transitions, DIST is the probability distribution of firing time, COND is the enabling condition, V1 is the first parameter of the probability distribution for timed distributions, and the weight for immediate distributions, V repeats V1 and in addition it specifies any other parameters of the probability distribution if more than one parameters are needed (e.g., for the Normal distribution both the mean and the variance are needed).

ARCS For each arc in the PNML model the following string is then added to the model:

```
K1:"P","T",M,K2,'F',
  (X1,Y1,...,Xn,Yn),1;
```

where K1 is a string that specifies the type of the arc ("IARC" for input arcs, "OARC" for output and inhibitor arcs), P is the place connected to the arc, T is the transition connected to the arc, M is the multiplicity value of the arc, K2 is a numerical code that identifies the type of the arc, F is an (optional) multiplicity function, and X1, Y1, . . . , Xn, Yn are parameters that identify the points touched by the arc in its graphical representation.

MEASURES OF INTEREST In the DEEM input format the measures of interest are specified as:

```
RES_FUNC:"N",'F',T;
```

where N is a name used to identify the measure, F is the reward function, and T is the type of measure, with "0" identifying an instant of time measure, "1" a measure accumulated over an interval of time, and "2" a measure accumulated and averaged over the length of the interval. Each measure defined at the higher levels results in a specific string, defining a measure of interest in the DEEM Input Format. For example, consider the measure "Reliability", having as target a failure mode "FM0" of component "C1", and let "C1_Failed0" be the place representing such failure mode in the PNML model. Such measure is defined through the following string in the DEEM Input File:

```
RES_FUNC:"ReliabilityC1_0",
  'IF(MARK(CI_Failed0)>0)THEN(0)ELSE(1)',1;
```

This transformation step is implemented as a "Query" in the ATL language [5]. While ATL modules perform model-transformations, queries are used to compute primitive values from source models. When strings are computed from source models, ATL queries can be used to perform code generation.

*DEEM Results File → CHESS ML (*t4*)*

This transformation step takes as input the specific output of the DEEM tool and the CHESS ML model that triggered the analysis, and modifies the latter by including the results of the analysis.

At UML level (m1 model) the metric to be evaluated is defined by means of the «`StateBasedAnalysis`» stereotype. The name of the UML classifier to which the stereotype is applied provides the search key to lookup the results value in the DEEM Results File. In the current implementation, once the resulting value has been identified, the obtained result is back-annotated in the *measureEvaluationResult* attribute of the «`StateBasedAnalysis`» stereotype that defined the metric. Given its simplicity, this transformation step is implemented as pure Java code.

5.2.2 *Server Process*

The server process is realized in Java as well, and it consists in a wrapper to the DEEM Simulator: it implements the TCP/IP communication with the client, and interacts with the tool. The server starts by listening on a predefined TCP port (9977 has been selected as the default one), and it waits for connections from client processes.

The server is multi-threaded, and it allows multiple instances of the DEEM Simulator to be executed in parallel, thus taking advantage of multi-processor or multi-core systems. When a new client connects, the server starts a new thread to handle the connection, and returns in a listening state. The thread receives the m5 model (i.e., a "DEEM Input File") from the client and saves it to a temporary folder on the server machine. To avoid conflicts, the name of the folder is derived from a combination of the current time on the server, and the IP address of the connecting client. The thread then runs the DEEM Simulator with the provided input file, and waits for the analysis to complete.

While the analysis is running, the server periodically updates the client on the current progress. Once the simulation is finished, the results file is read from the temporary directory, and transmitted back to the connected client. The temporary directory is then removed, and the connection is closed.

5.3 SUMMARY

The model-transformation approach described in the previous chapter has been concretely realized as a plugin for the Eclipse platform. The main focus in defining the architecture of such plugin was to enable its reuse and adaptation to different environments and contexts. This led to specific requirements concerning toolchain reusability, and the definition of a generic architecture for a reusable toolchain to be use as a reference in the development of the CHESS-SBA tool. In the following we discuss the context in which the toolchain can potentially be reused, and how.

REUSE WITH OTHER ADLS (R1). The adaptation of the toolchain to ADLs other than CHESS ML (e.g., AADL, SysML) requires to modify only few elements of the architecture, namely m1, t1, and m4. All the other elements (including the server process) can be reused as they are. Of course, the new m1 language should be able to express all the dependability information that is needed for the analysis.

REUSE WITH OTHER EVALUATION TOOLS (R3). Reusing the toolchain with other evaluation tools for Stochastic Petri Nets, metamodels m1, m2, m3, as well as model transformations t1 and t2 can be completely reused. The other elements of the client process, which are more dependent on the

adopted analysis tool, need however to be modified. In order to interact with the new analysis tool, the server process needs of course to be modified as well.

REUSE WITH OTHER ANALYSIS FORMALISMS (R2). Also when adapting to another analysis formalism (e.g., PEPA), some of the existing toolchain elements can be reused. In particular, metamodels m1 and m2, as well as the associated t1 transformation can be reused as they are. In certain cases it may be possible to reuse also m4, m5, t3, t4, e.g., when the analysis tool is a multi-formalism tool supporting the new analysis formalism.

REUSE FOR OTHER ANALYSES. Although the need to reuse the toolchain for other analysis purposes was not part of initial requirements, parts of the implemented toolchain can in principle be reused also for other kind of analyses. For example, Stochastic Petri Nets and similar formalisms are also used for performance, power efficiency, and other kinds of analyses. In such case, metamodels m3, m4, m5, transformations t3, t4, as well as the server process can be reused. Of course, the source language m1 would be a different language containing information for that particular kind of analysis, which would then be filtered into a different analysis-dependent intermediate model (m2). Accordingly, related model-transformations (t1 and t2) would need to be changed as well.

PLATFORM INDEPENDENCY (R4). The requirement of platform independency is achieved by i) using Java as the main language for developing the toolchain, and ii) dividing the toolchain into a client and a server process. Although the server process is bound to the platform required by the evaluation tool (Linux in case of the DEEM Simulator), the client process runs within the Eclipse framework [80], which is available for a wide range of platforms. The final user of the toolchain, which will interact with the client process only, is therefore not bound to any specific computing platform.

Finally, another important benefit in using such toolchain architecture is the convenience in its implementation: the different elements of the plugin can be developed in isolation, and can therefore be assigned to different teams or individuals, possibly based on their skills in different areas of expertise (e.g., metamodeling, model-transformation, coding).

6 CASE STUDIES

In this section we apply the approach presented in the previous chapters to two different case studies, modeling the system architecture with CHESS ML, and performing quantitative dependability analysis through the CHESS-SBA tool.

Within the CHESS project, the approach has been applied to a simple use-case of an onboard system in the railways domain [42]. The detailed descriptions of CHESS case studies is however confidential material restricted to project partners only. Also, since the CHESS use-case had to take into account for all the project concerns (i.e., code generation, schedulability, dependability...), it addressed only a subset of DEP-UML capabilities, while the case studies presented in this chapter and are more focused on DEP-UML features. For this reason, the use-case developed within CHESS is not reported here; application of the methodology and plugin to such use-case can however be found in a demonstration video [41] developed within the project and the associated tutorial material [133].

All the metrics that are reported in this chapter have been evaluated by discrete-event simulation, and are computed on at least 10^3 and at most 10^5 simulation batches, with a confidence level of 99% and relative confidence interval of half-width 1%. It is however important to emphasize that the focus here is not on the actual number obtained from evaluations, but rather on the evaluation process itself.

6.1 MULTIMEDIA PROCESSING WORKSTATION

Digital multimedia content is becoming an integral part of our modern society. Many public and private organizations are providing access to some kind of multimedia service, e.g., for education [111], or entertainment purposes. Such applications are often required to support real-time streaming, to provide remote access some kind of event that is currently ongoing. Real-time multimedia streaming is also employed in critical applications, e.g., computer-assisted surgery [122], or "access to medical expertise" [22]. More in general, real-time processing of data streams (not necessarily of multimedia kind) is employed for online monitoring applications [26, 66], which play a key role in achieving dependability of critical systems and infrastructures.

The case study presented in this section involves a multimedia processing system, having the task to perform real-time processing of data. The focus is on real-time audio/video processing, e.g., for Distributed Interactive Multime-

dia Applications (DIMAs) [108], in which the main dependability requirement concerns with their availability.

6.1.1 *System Description*

The system is composed of a dedicated hardware workstation, running an ad-hoc multimedia processing software application.

The multimedia processing software provides its clients with a streaming service, exposing the functions needed to control the stream. Internally, the software application is composed of four main components: the *processing* component, performing the actual data processing; the *input* component, handling the real-time acquiring and buffering of data to be processed; the *output* component, for the transmission of processed data; and the *supervisor* component, having the task to coordinate the other components, and exchange control data with the clients.

The hardware architecture of the workstations is composed of a main board, a CPU, and an input/output board. Additionally, the hardware architecture includes a Graphics Processing Unit (GPU), which is used to aid the CPU in the computations. Actually, the recent growth of GPUs capabilities, and the introduction of General Purpose GPUs (GPGPUs), has made it common to use the GPU as a coprocessor for performing intensive data processing [62]. A prominent example is the Nvidia's Compute Unified Device Architecture (CUDA) which provides GPGPU capabilities to off-the-shelf GPU devices. The *processing* component is thus realized as a CUDA application, to gain advantage of the processing power of the GPU.

For the purpose of our analysis, we consider such a system to be affected by two possible failure modes: *stopped*, in which the data stream that should be produced by the application is interrupted, and *degraded*, in which the data stream that should be produced by the application is present but not completely correct, e.g,. delayed or with lower quality.

We assume that the system is used without interruptions (24 hours per day, 7 hours per week), and we are interested in its dependability properties in the timespan of 1 month, e.g., in order to evaluate the provided QoS during a billing period, after which the system is completely checked and repaired, if needed. Concerning the metrics of to be evaluated on the system, we are interested in i) its *availability* during the month; and ii) its *failure distribution* with respect to each failure mode, i.e., the probability that such failure mode has occurred at least once, after one month of continuous operation.

One acknowledged problem in the usage of CUDA modules for high-integrity operations is the lack of basic error detection and correction capabilities in GPUs, conversely to CRC or ECC mechanisms that are typically present in traditional processing units and memories [121]. For this reason, the *processing*

component is deemed to be more subject to faults with respect to the other components of the software architecture. To contrast this behavior, the *processing* component is periodically restarted, in order to remove latent errors that might have been accumulated. Since it may be affected by propagation, the *supervisor* component is restarted as well. One of the objectives of the analysis is also to evaluate the effect of this rejuvenation [93] countermeasure with respect to the metrics of interest, and tune the restart period.

6.1.2 *System model with CHESS ML and DEP-UML*

In this section we describe how such system is designed and analyzed following the CHESS ML methodology, and using DEP-UML and the related model-transformations to perform the analysis of the target dependability metrics.

Functional Software Model

According to the CHESS methodology, the first step in system design is the creation of the software architecture. The main entities involved in the design of the software architecture are shown in the *Class Diagram* of Figure 6.1. For convenience, the figure depicts the final version of the diagram; it should be noted however that, according to the CHESS methodology, the diagram should be built incrementally in different design views. In particular, the enrichment with dependability information is typically performed at a later step, after the overall system architecture has been defined.

The first step is to create the interfaces that will be used by software components of the system. This involves four interfaces: `IStream`, the main interface of the system to the environment; `IProcessing`, which enables the control of the processing component; `IInput` and `IOutput`, which let the supervisor interacts with the input and output components, respectively.

The second step involves the creation of component types: the `Processing` component type realizes the `IProcessing` interface, i.e., it exposes such interface to the other components of the architecture. Similarly, component types `StreamReader` and `StreamWriter` realize the `IInput` and `IOutput` interfaces, respectively. Such relationships are specified with `Realization` relations in the diagram. Finally, the `Supervisor` component type realizes the `IStream` interface, i.e., the main interface of the multimedia processing application to the environment. However, in order to realize such interface, the `Supervisor` depends on (i.e., it requires) the other three interfaces defined before: `IProcessing`, `IInput` and `IOutput`. Such dependencies are specified with `Dependency` relations in the diagram.

The subsequent step involves the definition of component implementations. In this case, one implementation per component type is defined: `Supervisor_impl`, `StreamReader_impl`, and `StreamWriter_impl` realize component types `Su-`

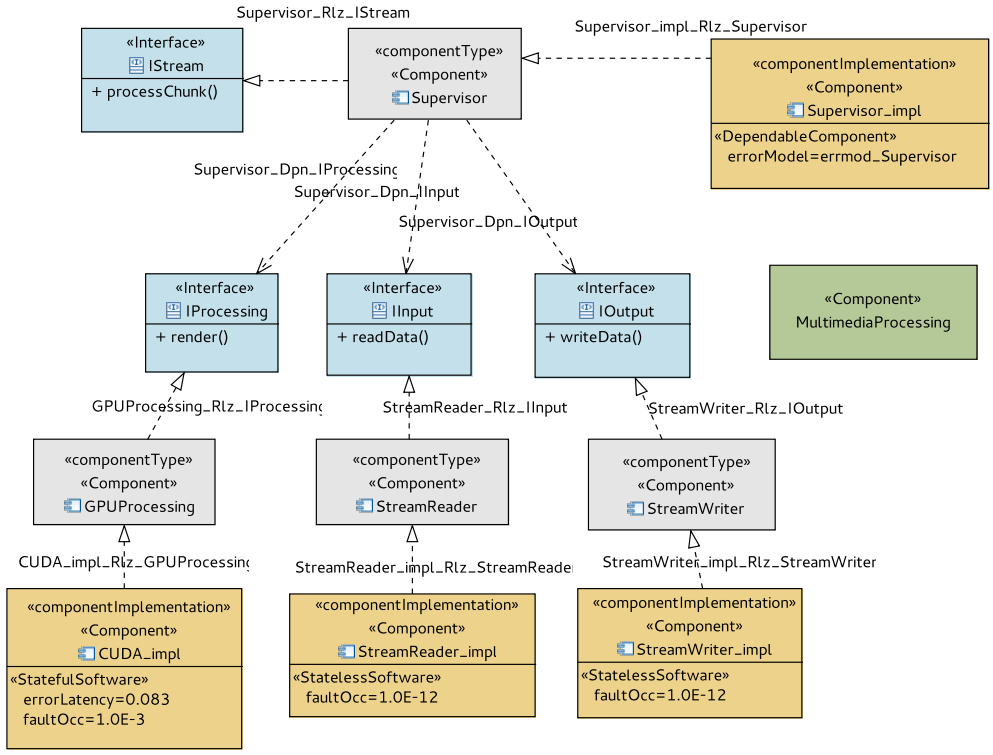


Figure 6.1: Software entities involved in the design of `MultimediaProcessing` application, enriched with DEP-UML dependability annotations.

`pervisor`, `StreamReader`, and `StreamWriter`, respectively. Component implementation `CUDA_impl` realizes the implementation of component type `Processing`, through GPGPU code for the CUDA platform. Finally, component `MultimediaProcessing` represents the overall software application.

The overall software architecture is then specified, as a composition of connected component instances. Instances are specified by means of a *Composite Structure Diagram* (Figure 6.2), in which the internal structure of the `MultimediaProcessing` component is detailed. The `MultimediaProcessing` component contains four component instances: one for each of the defined component types. The instance of `Processing_impl` exposes a provided port of type `IProcessing`, through which it is connected to a matching required port of the `Supervisor_impl` instance. In a similar way, instances of `StreamReader_impl` and `StreamWriter_impl` are connected to the instance of `Supervisor_impl`, through its required ports.

Hardware Platform and Allocation

Once the software architecture has been completely specified, the hardware platform is then defined. The hardware platform is constituted of four components (Figure 6.3): i) the main board to which other components are connected

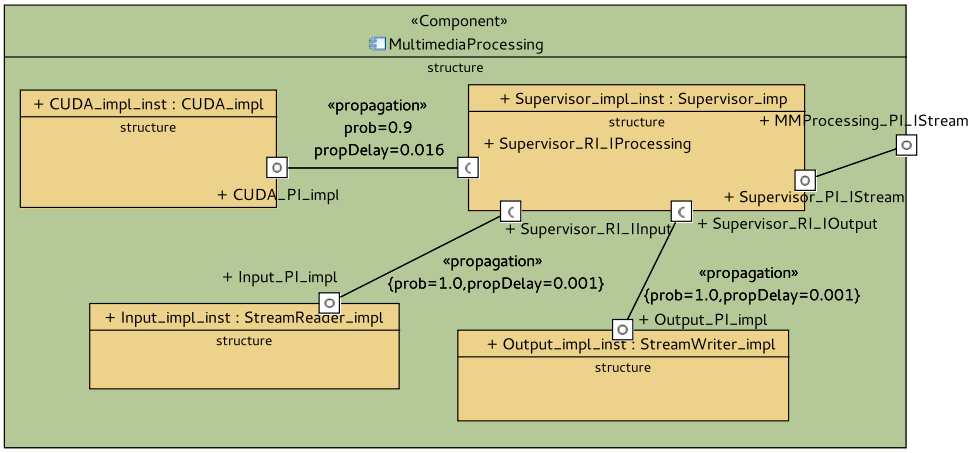


Figure 6.2: Software architecture of the `MultimediaProcessing` application, enriched with DEP-UML dependability annotations.

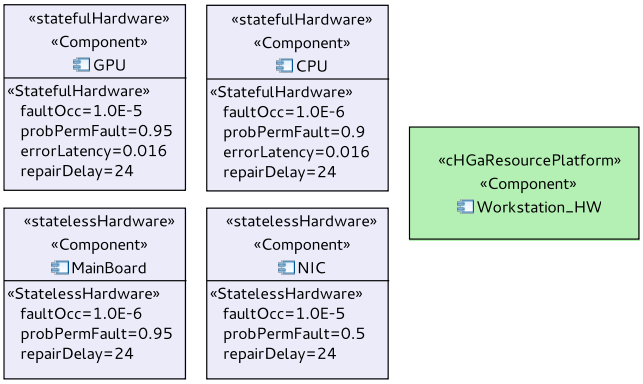


Figure 6.3: Hardware components constituting the multimedia workstation, enriched with DEP-UML dependability annotations.

(`MainBoard` component); ii) the Central Processing Unit (`CPU` component); iii) the Graphics Processing Unit (`GPU` component); and iv) the Network Interface Card (`NIC` component). Finally, the `Workstation_HW` component represents the hardware platform itself.

In a similar way as for the software architecture, the architecture of the hardware platform is specified by defining instances of the specified components (Figure 6.4). The hardware platform contains four component instances, one for each of the specified components. The component `MainBoard` has three «`FlowPort`» ports of kind *inout*, meaning that bidirectional communication occurs on them. Such ports connect the `MainBoard` instance with instances of components `CPU`, `GPU`, and `NIC`. Finally, the `CHGaResourcePlatform` stereotype is applied to the `Workstation_HW` component, in order to allow the resulting `InstanceSpecification` to be targeted by the analysis.

After both the software and hardware architecture have been completely defined, allocation of software component instances on hardware component instance is performed. However, for allocation to be specified, the actual `InstanceSpecification` elements need to be generated from *Composite Structure Diagram* specifications.

The creation of `InstanceSpecification` elements is performed by the executing the "Build Instance" command of the CHESS framework on the components for which a *Composite Structure Diagram* has been defined [38]: in this case the `MultimediaProcessing` and `Workstation_HW` components.

The result of this action is shown in the left part of Figure 6.4, in which the generated software and hardware `InstanceSpecification` elements are depicted. The *Composite Structure Diagram* of Figure 6.2 leads to the creation of 8 `InstanceSpecification` elements: 1 for the containing component, 1 for each of the subcomponents, and 1 for each connector. Similarly, 8 `InstanceSpecification` elements are generated from the *Composite Structure Diagram* in Figure 6.4.

Once both software and hardware instances are available, two UML comments, stereotyped with «`MARTE::Assign`» are used to specify allocation information (Figure 6.4). In particular, the rightmost comment specifies the allocation of the CUDA processing module on the GPU, i.e., of `MultimediaProcessing_CUDA_impl_inst` software instance on the `Workstation_HW_gpu` hardware instance. The leftmost comment specifies instead that the other instances of software components implementations are allocated on the CPU, i.e., on the `Workstation_HW_cpu` instance.

Enrichment with Dependability Information

The modeled system architecture is then enriched with dependability information, using DEP-UML stereotypes. Component implementations `CUDA_impl`, `StreamReader_impl` and `StreamWriter_impl` are annotated with "dependability template" stereotypes (see Figure 6.3). `StreamReader_impl` and `StreamWriter_impl` are considered «`StatelessSoftware`» components: although they might have some internal buffer for processing stream I/O, they are of are of reduced size, and error latency is thus considered to be negligible. Accordingly, only the *faultOcc* attribute is specified for them.

Conversely, component implementation `CUDA_impl` is a «`StatefulSoftware`» component: internal variables are used for different aspects of stream processing. Faults may therefore require some time before they reach the service interface; in the figure, *errorLatency* is set to 0.083 hours, which correspond to approximately 5 minutes.

A more detailed dependability specification is provided for the `Supervisor_impl` component implementation, for which an error model is defined (Figure 6.5). The «`ExternalFault`» `xft1` occurs when a failure occurs on one of the two required ports of type `IInput` and `IOutput`, i.e., when the components con-

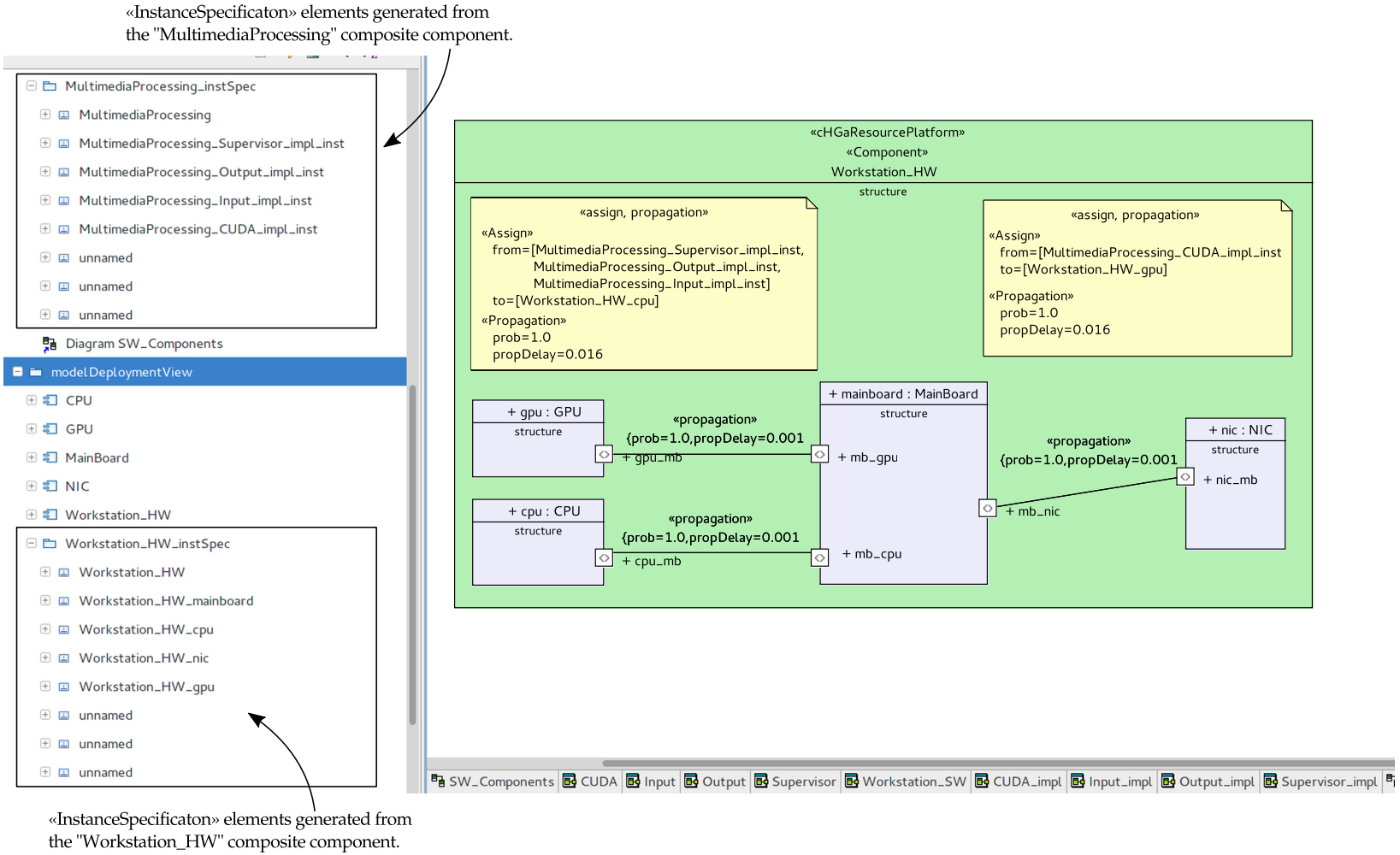


Figure 6.4: Hardware architecture of the multimedia workstation, with allocation information.

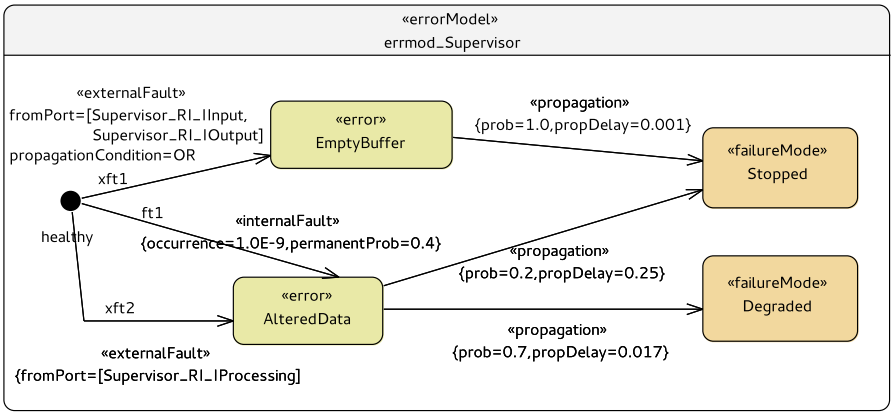


Figure 6.5: Error model for the `Supervisor_impl` component implementation.

nected on the other end of such ports fail. Such event generates the «`Error`» `EmptyBuffer`, meaning that the supervisor is no longer able to read from (write to) the input (output) buffer. After a short delay (0.001 hours, approximately 3.6 seconds) the error causes the occurrence of the `Stopped` failure mode.

Instead, a failure occurring on the port of type `IProcessing` generates an error of kind `AlteredData` within the supervisor, meaning that the stream data has been altered during its processing. The same error can also be generated by an internal fault of the supervisor component, e.g., for a wrong handling of data exchanged through the interfaces. With probability 0.7, and a delay of 1 minute, the `AlteredData` error may cause the occurrence of the `Degraded` failure mode, in which the streaming is still active, but produces wrong or degraded output. The presence of the `AlteredData` error can also lead to the `Stopped` failure mode; we assume however that the supervisor has a high chance to recover (e.g., by skipping some frames), and thus propagation occurs in this case with a lower probability (0.2) and a higher delay (0.25 hours, corresponding to 15 minutes).

Since `Supervisor_impl_inst` is the component instance to which the overall system service is delegated (see Figure 6.2), the two failure modes specified in the error model are also the two possible failure modes of the system.

"Dependability template" stereotypes are also used to attach dependability information on hardware components (see Figure 6.3). In particular, the `CPU` and `GPU` components are considered «`StatefulHardware`» components, since they have internal memories in which latent errors may accumulate; *errorLatency* is set to 0.017 hours, which correspond to approximately 1 minute. The `MainBoard` and `NIC` components are instead considered «`StatelessHardware`» components. The `CPU` and `MainBoard` are the most reliable ones, both having a fault occurrence rate of 10^{-6} faults/hour; the `GPU` and `NIC` have instead a fault occurrence of an order of magnitude higher, 10^{-5} faults/hour. After the fail-

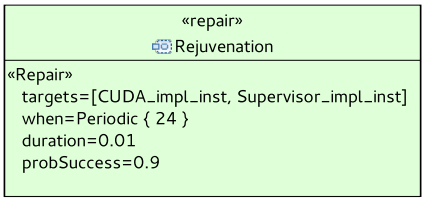


Figure 6.6: Definition of the maintenance activities performed on the multimedia processing application.

ure, an hardware components is repaired after an exponential delay of mean 24 hours (*repairDelay* attribute).

Connectors connecting component instances, both software and hardware, are annotated with the «Propagation» stereotype (see Figures 6.2 and 6.4). Similarly, «Assign» allocation specification are enriched with the «Propagation» stereotype. Propagation from hardware to software is assumed to always occur (i.e., with probability 1), and with a mean delay of about 1 minute.

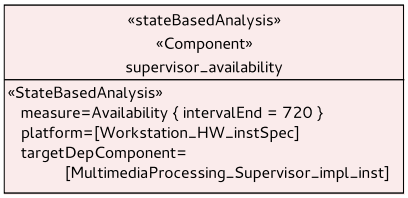
Finally, the specification of the periodic restart of the *processing* and *supervisor* components is specified as a maintenance activity, through the «Repair» stereotype (Figure 6.6). The activity targets the CUDA_impl_inst and Supervisor_impl_inst instances of the CUDA_impl and Supervisor_impl component implementations, and it is executed periodically every 24 hours. The activity succeeds 90% of the times, and requires 0.01 hours to be completed, i.e., 36 seconds.

Finally, metrics of interest for the analysis are defined. Figure 6.7 depicts the definition of three Component elements stereotyped with the «StateBasedAnalysis» stereotype, which define three metrics that should be evaluated by the CHESS-SBA plugin.

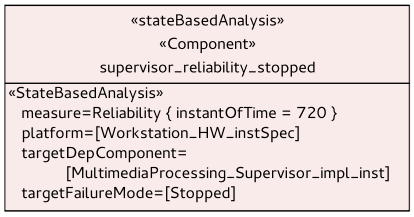
Figure 6.7a depicts the definition of the *interval-of-time availability of the supervisor*; since it is the component that interacts with the external environment by providing the "streaming" service, it is the one that is targeted by the analysis. Accordingly, the *targetDepComponent* attribute refers to the MultimediaProcessing_Supervisor_impl_inst InstanceSpecification element. The *measure* attribute specifies that the metrics is an interval-of-time availability, in the interval $[0, t]$, with $t = 720$ hours, corresponding to 1 month. In the following, we will refer to this metric as the *Availability* of the system, $\mathcal{A}(0, t)$.

Figures 6.7b and 6.7c depict the definition of the *instant-of-time reliability of the supervisor*, with respect to the failure modes "Stopped" and "Degraded", respectively. In this case also the *targetFailureMode* attribute needs to be specified, and it is set to the corresponding «FailureMode» elements in the error model of Supervisor_impl (see Figure 6.5). In the following, we refer to these metrics as $\mathcal{R}^{stop}(t)$ and $\mathcal{R}^{deg}(t)$, respectively.

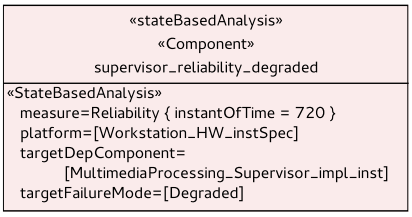
For all the three metrics, the *platform* attribute, which identifies the hardware platform to be taken into account, refers to the Workstation_HW_instSpec.



(a) $\mathcal{A}(0, t)$, $t = 720$ hours.



(b) $\mathcal{R}^{stop}(t)$, $t = 720$ hours.



(c) $\mathcal{R}^{deg}(t)$, $t = 720$ hours.

Figure 6.7: Definition of the metrics of interest for the evaluation of the multimedia processing application.

Table 6.1: Main parameters used in the evaluation of the multimedia workstation.

Symbol	*Corresponding model element*	*Default value*
λ_{CPU}	CPU.faultOcc	10^{-6} faults/hour
λ_{GPU}	GPU.faultOcc	10^{-5} faults/hour
λ_{MB}	MB.faultOcc	10^{-6} faults/hour
λ_{IO}	IO.faultOcc	10^{-5} faults/hour
T_{Rej}	Rejuvenation.when.Deterministic.Value	24 hours
λ_{CUDA}	CUDA_impl.faultOcc	10^{-3} faults/hour

6.1.3 *Analysis and Results*

The system model is then evaluated using the CHESS-SBA plugin described in Chapter 5. The generated IDM model contains 198 elements, including all the auxiliary elements (e.g., instances of `Distribution` elements). In particular, the model includes 8 `Component`, 8 `InternalFault`, 13 `ExternalFault`, 9 `Error`, 9 `FailureMode`, 17 `FaultsGenerateErrors`, 10 `ErrorsProduceFailure`, 5 `RepairActivity`, 1 `Availability`, 2 `Reliability` elements. The PNML model generated from it contains 1674 elements, including 132 places, 155 transitions, and 405 arcs. Finally, the resulting DEEM input file contains 1580 lines.

During the evaluation, some key system parameters have been varied in order to assess their impact on the metrics of interest. The main parameters of this study are summarized in Table 6.1, together with their default values, and the corresponding DEP-UML attributes in the previously presented diagrams.

As a first study, we evaluate the impact of the fault occurrence rate of the `CUDA_impl` component implementation (λ_{CUDA} parameter) on the probability

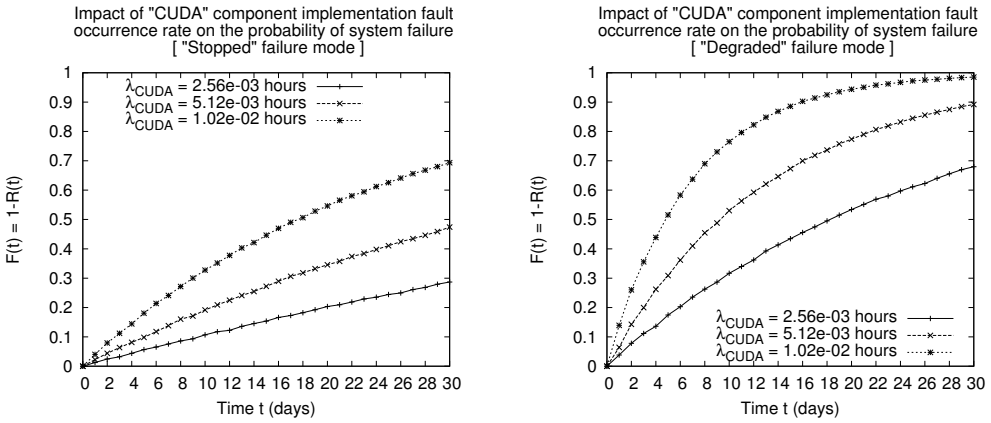


Figure 6.8: Impact of the fault occurrence rate of the `CUDA_impl` component implementation on the failure probability of the system.

of occurrence of the two system's failure modes. Such analysis helps in understanding the behavior of the system in case of faults, and thus define the proper QoS contracts to offer. For varying the fault occurrence rate, we selected the `CUDA_impl` component since it is the one with the higher fault occurrence rate; a similar analysis could however be performed on the other components as well.

The results of such evaluation are shown in Figure 6.8, both for the "Stopped" failure mode (6.8a) and for the "Degraded" failure mode (6.8b). More in details, the figure depicts the quantities $F^{stop}(t)$ and $F^{deg}(t)$ as a function of time, evaluated as $1 - R^{stop}(t)$ and $1 - R^{deg}(t)$, respectively. The evaluation has been performed on a timespan 1 month (30 days, 720 hours).

Of course, an increase of the fault occurrence rate of the component causes an increase of failure probability for both the failure modes. However, the results clearly show that a failure of the CUDA component is more likely to cause a degradation in the multimedia stream, rather than an interruption of the processing Actually, even with a fault occurrence rate one order of magnitude higher than the default, there is still a 30% probability that the "Stopped" failure mode never occurs within one month of operation, while the probability that degradation has occurred at least once is near to 1.

It should be noted however that reliability is not the main requirement of the system, which instead should provide a high availability level in its operational period of one month. In the following we then evaluate the impact of system parameters on the $\mathcal{A}(0, t)$ metric, with a particular focus on assessing the effectiveness of the rejuvenation mitigation measure that has been defined during system design. Figure 6.9 shows the impact of the period at which software rejuvenation is performed, i.e., the period at which component instances `CUDA_`-

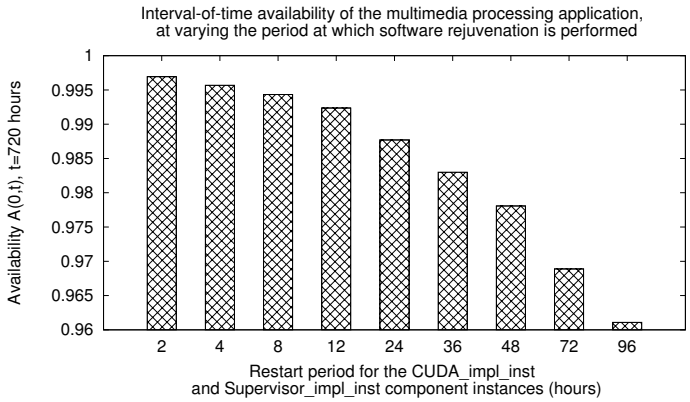


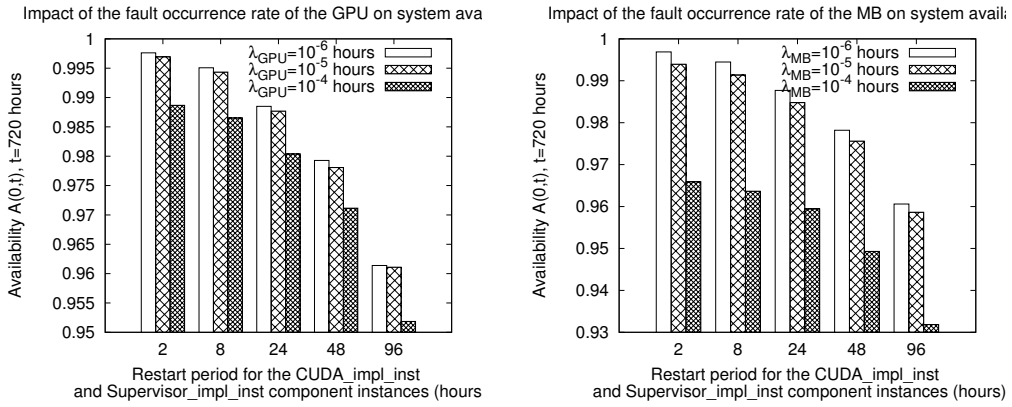
Figure 6.9: Availability of the multimedia processing application at varying the rejuvenation period.

`impl_inst` and `Supervisor_impl_inst` are restarted. Results in the figure show that the countermeasure is actually effective in improving system availability, and that the restart period should be tuned based on the actual availability requirements of the system. For example, the default configuration is not enough for providing an availability level higher than 99%, which is instead achieved by triggering the restart every 12 hours.

The next evaluation focus on assessing the ability of the introduced mechanism to contrast software faults that are originated from failures of the hardware platform. For this purpose, we evaluate the impact on $\mathcal{A}(0,t)$ of the fault occurrence rate of hardware components, at varying the T_{rej} parameter. Figure 6.10 shows the results of this evaluation for the `GPU` (6.10a) and `MB` (6.10b) components. The obtained results show that the restart of selected software components actually provides some benefits also with respect to hardware failures, probably due to the occurrence of transient faults.

This mechanism is however effective only if the increase in the fault occurrence rate is limited. Considering the `GPU` component, for example, an increase of one order of magnitude still allows an availability level of 99% to be reached, by adopting a restart period of 8 hours. However, if λ_{GPU} increases to 10^{-4} faults/hour, the level of 99% cannot be reached anymore, even if using a restart period of 2 hours.

The impact of the mainboard is even higher (Figure 6.10b): in this case, a fault occurrence rate of 10^{-4} faults/hour makes the availability metric unable to even reach the level of 97%. This is mainly due to the fact that a failure of the `MB` component propagates to all the other hardware components, possibly also leading them to failure; the failure of the mainboard has thus a higher impact on the ability of the system to provide a correct service.



(a) (b)

Figure 6.10: Impact of GPU and MB fault occurrence rates on the availability of the multimedia processing application.

6.2 FIRE DETECTION SYSTEM

While the previous case study was mainly descriptive, in this section we aim to "emulate" the design and assessment of a safety-critical system using the proposed methodology. The case study is a revisited and extended version of the fire detection system modeled with the IDM in [134] and recalled in Section 4.2.2.

This case study also demonstrates how the DEP-UML, while compatible with it, does not need to strictly follow the CHESS development methodology, in which the design of the *software* architecture plays a primary role. Instead, the approach can be applied also with a more hardware-oriented view of the system, for the analysis of hardware architectures only.

With respect to the previous one, this case study also highlights the support of DEP-UML for incremental modeling and evaluation. Actually, during our fictional path to the design of the system, we will take into account two subsequent refinement steps, corresponding to different stages of system development.

6.2.1 *System Description*

We here recall the description of the system, and provide additional details that will be taken into account during the modeling process.

The fire detection system takes its decision based on a set of Smoke Sensors (SMS) and a set of Over-Temperature Detectors (OTD). These sensors are managed by two Fire Detection Units (FDU), which analyze the data received from

them and trigger the alarm signal when a fire event is detected. Both the FDU are able to detect a fire event, but only one of them is allowed to control the sensors at the same time; accordingly, they periodically switch between "master" and "slave" status: while both receive sensors data, only the "master" is able to properly command the sensors.

During normal operation, each FDU emits a "No Fire/Smoke Detected" (NFSD) signal, to indicate that no fire or smoke have been detected by reading the controlled sensors. When fire or smoke is detected, the FDU disables the signal, thus raising an alarm. A hardware fail-safe comparator receives both NFSD signals from the FDUs, and provides a global system alarm if at least one of the two FDUs has triggered an alarm; therefore, the comparator triggers the alarm also if the two NFSD signals become different, e.g., because one of the two FDUs is not correctly working.

In addition to the NFSD signal, each FDU has the following additional interfaces: "TLoop", which connects to the set of over-temperature sensors, "SLoop", which connects to the set of smoke sensors, and "FDU_SC", which connects each FDU to the other one through a serial cable. The "FDU_SC" interface is used by the two FDUs to agree on their master/slave roles.

An FDU is affected by two possible failure modes. In the "No Alarm" failure mode the fire detection functionality of the FDU is compromised but no other consequences of the system occur; in this case the FDU moves to a safe state and signals its failure. In the the "Holds Control" failure mode fire detection is compromised as well, and there is also the possibility that an FDU prevents the other to become "master", thus preventing it to access the sensors. A catastrophic event occurs when both the FDUs are in a state such that the system would not be able to detect a fire event. For the purpose of reliability, the system is considered failed when at least one of the two FDUs would not be able to detect a fire event.

Each FDU is composed of the following boards: i) Backplane (BKP) board, on which all the other boards are connected; ii) Power Supply (PSP) board, which provides the power to the backplane board and consequently to all the other boards; iii) Temperature Loop (TLP) board, which interfaces with the over-temperature sensors through the "TLoop" interface; iv) Smoke Loop (SLP) board, which interfaces with the smoke sensors through the "SLoop" interface; v) Input/Output (I/O) board, which manages the NFSD interface and triggers the alarm; vi) Central Processing Unit (CPU) board, which analyzes the data coming from the smoke and over-temperature sensors, possibly triggering the alarm signal on the I/O board. The CPU board is directly connected to other FDU's CPU board through the "FDU_SC" interface.

The "Holds Control" failure mode can be caused by a failure of the CPU itself, or by specific failure modes of the SLP and TLP boards, which prevent the CPU to release the previously acquired "master" status.

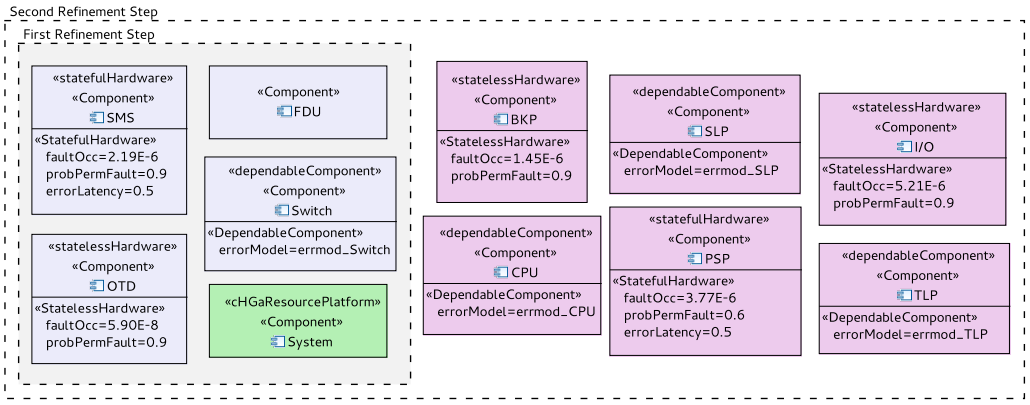


Figure 6.11: Hardware components which are involved in the definition of architecture of the fire detection system, in two subsequent refinement steps.

The system is subject to a complete maintenance every $10^4 = 10.000$ hours of operations, after which every single sensor as well as all components of both FDU are checked for correct operation. After 10^4 hours of operation the system is therefore considered to be as good as new.

System safety requirements impose a Safety Integrity Level (SIL) [31] equal or higher to SIL-1. Therefore, the aim of the analysis is quantify the probability of occurrence of an hazard, i.e., of an event for which both the FDUs are not able to detect the fire. To fulfill the SIL-1 quantitative requirements [31], the mean time to the occurrence of an hazard, Mean Time To Hazardous Event (MTTHE), should be higher than 10^5 hours. Moreover, we are also interested in system reliability, measured as the MTTF, which should be higher than the planned maintenance interval.

6.2.2 *System Model – Early Phase*

The system is designed and analyzed in two subsequent refinement steps. In the first step the two FDUs are seen as black-boxes components, i.e., their internal structure is not detailed.

The hardware components involved in the definition of the system architecture are depicted in the *Class Diagram* of Figure 6.11. For convenience, the figure shows components involved in both the phases of system modeling; leftmost components are those involved in the first refinement only; rightmost components are introduced in the second refinement step.

The first refinement step involves five components: `OTD`, representing the set of over-temperature sensors; `SMS` representing the set of smoke sensors; `FDU`, representing a FDU; `Switch`, representing the fail-safe comparator; and `System`, representing the overall fire detection system.

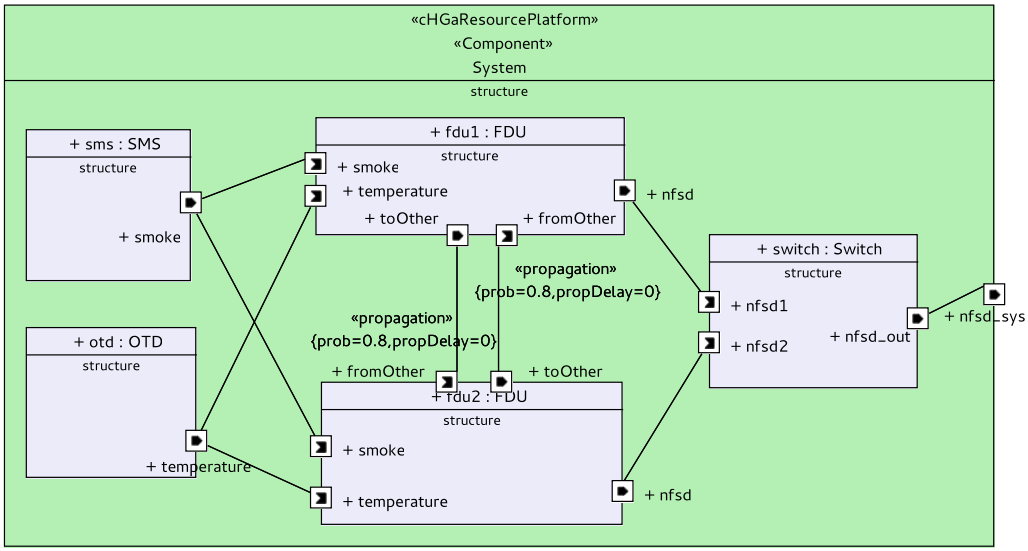


Figure 6.12: Hardware architecture of the fire detection system.

Such components are instantiated in the *Composite Structure Diagram* of the `System` component (Figure 6.12), in order to define the overall system (hardware) architecture. The hardware architecture includes an instance of the `SMS` and `OTD` components, both having a «`FlowPort`» of kind *out*, on which the data acquired by sensors is retrieved.

The architecture also includes two instances of the `FDU` component (*fdu1* and *fdu2*), and an instance of the `Switch` component. The instance of the `Switch` component has two «`FlowPort`» elments with *in* direction, corresponding to the two "NFSD" signals received from the FDUs.

Each `FDU` instance has three *in* «`FlowPort`» and two *out* «`FlowPort`» elements. A couple of ports with opposite direction correspond to the "FDU_SC" interface, which connects each `FDU` component instance with the other one. The other «`FlowPort`» with *out* direction corresponds to the "NFSD" interface, and it is connected with the instance of the `Switch` component. Finally, the other two ports with *in* direction are connected with the instances of the `SMS` and `OTD` components, in order to retrieve the data from sensors, and correspond to the "SLoop" and "TLoop" interfaces, respectively.

Concerning dependability attributes, the `SMS` component is considered a «`StatefulHardware`» component, with an *errorLatency* of 30 minutes, while `OTD` is a «`StatelessHardware`». In fact, smoke sensors perform short-term averages of the acquired value, while over-temperature detectors are simply heat-sensible electrical elements. For the same reason, over-temperature detectors are more reliable than smoke sensors, and have a lower fault occurrence rate (*faultOcc* attribute). Both components have a 10% chance of transient faults. The repair

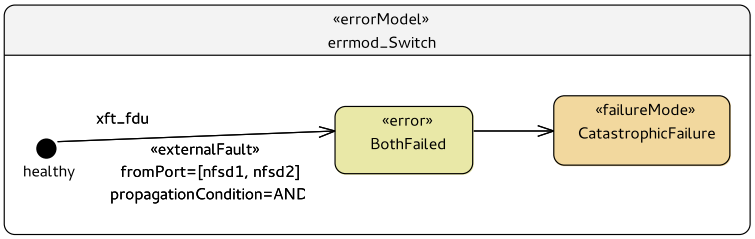


Figure 6.13: Error model for the `Switch` component.

delay is not specified, since it is assumed that no repairs are performed until 10.000 hours of system lifetime have elapsed.

The dependability properties of the `FDU` and `Switch` components are instead specified using «`DependableComponent`» stereotype and the error model facility. The «`ErrorModel`» for the `Switch` component is depicted in Figure 6.13. The error model contains a single «`Error`» `BothFailed`, which is generated by the «`ExternalFault`» `xft_fdu`. Its *fromPorts* and *propagationCondition* attributes specify that such fault occurs when both the components connected through the `nfsd1` and `nfsd2` ports are failed, i.e., when both the FDUs are failed. The presence of the `BothFailed` error within the `Switch` immediately leads to the `CatastrophicFailure` failure mode of the comparator, i.e., the failure of the system-level "NFSD" signal, which corresponds to the hazardous event.

The «`ErrorModel`» associated with the `FDU` component is instead depicted in Figure 6.14. The FDU is subject to two failure modes, `NoAlarm` and `HoldsControl`. The `NoAlarm` failure mode, in which the NFSD signal cannot be correctly triggered, affects the *nfsd* port only. `HoldsControl` corresponds to the "Holds Control" failure mode, in which the FDU may prevent the other one from accessing the sensors; accordingly, this failure mode affects both the *nfsd* and the *toOther* ports of the FDU. The two failure modes are caused by the `e1` and `e2` errors, respectively. Error `e2`, leading to the "Hazardous" failure mode can be caused by an internal fault of the FDU only; conversely, error `e1` can occur also because of propagation from smoke or temperature sensors.

Once all the required properties are specified, the «`CHGaResourcesPlatform`» is applied to the `System` component, in order for it to be targeted by the analysis, and component instances are generated. At this point, the metrics of interest for the analysis can be specified using the «`StateBasedAnalysis`» stereotype (Figure 6.15).

The probability of occurrence of a hazardous event (6.15a) is specified as an instant-of-time reliability metric, having the component instance *System_switch* as *targetDepComponent*. Such specification means that an hazardous event has occurred if the service provided by the switch component has failed. In the following evaluations we will refer to this metric as the *Safety* of the system, $\mathcal{S}(t)$.

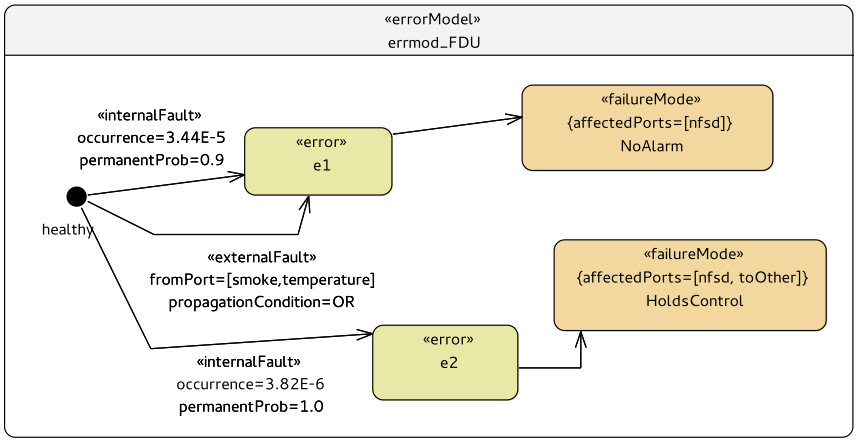
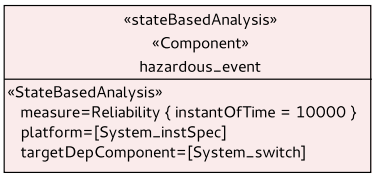
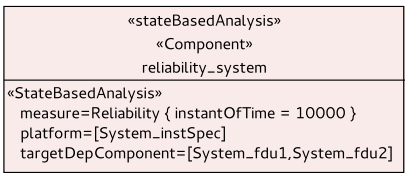


Figure 6.14: Error model associated with the FDU component (first refinement step).



(a) $\mathcal{S}(t)$, $t = 10^4$ hours.



(b) $\mathcal{S}(t)$, $t = 10^4$ hours.

Figure 6.15: Definition of metrics of interest for the evaluation of the fire detection system.

The probability of system failure (e.g., an alarm has been raised because of the failure of a component), is specified as an instant-of-time reliability metric as well, having both instances of the FDU component, *System_fdu1* and *System_fdu2* as *targetDepComponent* (6.15b). Such specification means that, from a functional point of view, the system is failed if at least one of the FDUs is failed: in this case the system will enter a safe state and halt its service[1]. In the following, we will refer to this metric as the *Reliability* of the system, $\mathcal{R}(t)$.

Since we assume that the system is as good as new after 10^4 hours of operation, the probabilities that it "survives" two different maintenance cycles are independent, and the number of intervals the system survives before experiencing a failure follows a geometric distribution. The MTTF and MTTHE metrics can then be obtained from $\mathcal{R}(t)$ and $\mathcal{S}(t)$, respectively, as the mean of

1 It should be noted that such behavior could be modeled also by specifying two different failure modes for the switch, and then defining two different metrics based on them. This approach is not adopted here since it produces a more complex error model for the switch, and would also add additional complexity later in the refinement phase (see Section 6.2.4).

Table 6.2: Main parameters used in the evaluation of the fire detection system.

Symbol	*Corresponding model element*	*Default value*
λ_{OTD}	OTD.faultOcc	$5.90 \cdot 10^{-8}$ faults/hour
λ_{SMS}	SMS.faultOcc	$2.19 \cdot 10^{-6}$ faults/hour
λ_{FDU}^{ft1}	errmod_FDU.ft1.occurrence	$3.44 \cdot 10^{-5}$ faults/hour
λ_{FDU}^{ft2}	errmod_FDU.ft2.occurrence	$3.82 \cdot 10^{-6}$ faults/hour
p_{prop}	fdu1_to_fdu2.prob; fdu2_to_fdu1.prob	10%

a geometric distribution multiplied by the length of the interval between two maintenance periods i.e.:

$$\begin{aligned} MTTF &= \frac{\mathcal{R}(T)}{1-\mathcal{R}(T)} \cdot T, \qquad T = 10^4 \text{ hours.} \\ MTTHE &= \frac{\mathcal{S}(T)}{1-\mathcal{S}(T)} \cdot T, \qquad T = 10^4 \text{ hours.} \end{aligned} \tag{6.1}$$

6.2.3 *Analysis and Results – Early Phase*

We now report some evaluations on the early system design. The generated IDM model contains 126 elements, while the resulting PNML model contains 1152 elements (including 252 arcs, 86 places, and 93 transitions). The resulting DEEM input file contains 988 lines. The main parameters used in this phase, together with corresponding attributes in the DEP-UML model and their default values are listed in Table 6.2. Such default values have been derived from the real industrial project to which this case study is inspired; in general, they can be obtained from datasheets [180] or records from previous similar projects.

The values obtained for the target metrics, with default values as reported in the table are the following:

$$\mathcal{S}(10^4) = 8.460727 \cdot 10^{-1}, \qquad \mathcal{R}(10^4) = 4.541527 \cdot 10^{-1},$$

which yield:

$$MTTHE = 54965 \text{ hours}, \qquad MTTF = 8320 \text{ hours}.$$

Unfortunately, both the metrics are below the values required to to satisfy system requirements; a more extended analysis to identify design solutions needs therefore to be performed. In particular, the most critical components with respect to the defined metrics need to be identified. This can be accomplished by evaluating the impact of the fault occurrence rates of the different system components on the metrics of interest. The results of such evaluations are reported in Figure 6.16.

Figure 6.16a depicts the impact on the target metrics of varying the fault occurrence rate of the over-temperature detectors block; the vertical line marks

the default value for the λ_{OTD} parameter. As shown in the figure, the OTD component has very little impact on the overall system metrics, mainly due to the fact that it has a very low fault occurrence rate. A wrong estimation of the reliability of this component has a reduced impact on the system: even if the fault occurrence rate is one order of magnitude higher, the system metrics would remain practically constant. On the other hand, further improving the fault occurrence rate of this component would not bring significant improvements on system properties as well.

A similar trend is observed for the SMS component (Figure 6.16b); however, the different default (i.e., estimated) value for λ_{SMS} lead us to different considerations. In this case, an optimistic estimation of this parameter has a significant effect on system reliability and safety; for this reason the value of this parameter should be carefully assessed. On the other hand, achieving a lower fault occurrence rate for the SMS component, e.g., by adopting higher-quality sensors, has little impact on the overall system metrics. In particular, even with λ_{SMS} two order of magnitude lower, $\mathcal{S}(10^4)$ is still below 0.9, which is not sufficient to satisfy system safety requirements.

The impact of the fault occurrence rate of the FDU component is highlighted in Figure 6.16c. Since the FDU component is affected by two different internal faults, having different occurrence rates, the x axis shows the sum between λ_{FDU}^{ft1} and λ_{FDU}^{ft2}, while the ratio between them is kept constant. As shown in the figure, the FDU component is the most critical component, and the dependability bottleneck of the system. The default value falls in a very critical point, and a little error in the estimation of its value can yield very large differences on the overall system metrics. Moreover, the figure shows that a relatively small improvement on the fault occurrence rate(s) of the FDU component (less than an order of magnitude) can be sufficient to let the system satisfy its requirements. As an example, the value $\mathcal{S}(10^4) \approx 0.916$, which is obtained for $\lambda_{FDU} \approx 2.2 \cdot 10^{-5}$ faults/hour, is sufficient to achieve MTTHE $\approx 1.09 \cdot 10^5$ hours.

Further analysis of the impact of the FDU component is reported in Figure 6.17, which focuses on assessing the impact of error propagation between the two FDUs. The two figures depict the metrics $\mathcal{S}(t)$ (6.17a) and $\mathcal{R}(t)$ (6.17b) as functions of time and propagation probability p_{prop}. As expected, results show that the propagation probability has impact on the $\mathcal{S}(t)$ metric only. From a modeling point of view, such results are a confirm that the system model has been correctly specified using the DEP-UML language; from a system design point of view, such results highlight and quantify the importance of FDUs error propagation with respect to system safety. However, the impact on safety seems to be relatively modest with this configuration, since in the worst case the $\mathcal{S}(t)$ metric is reduced by about 5%.

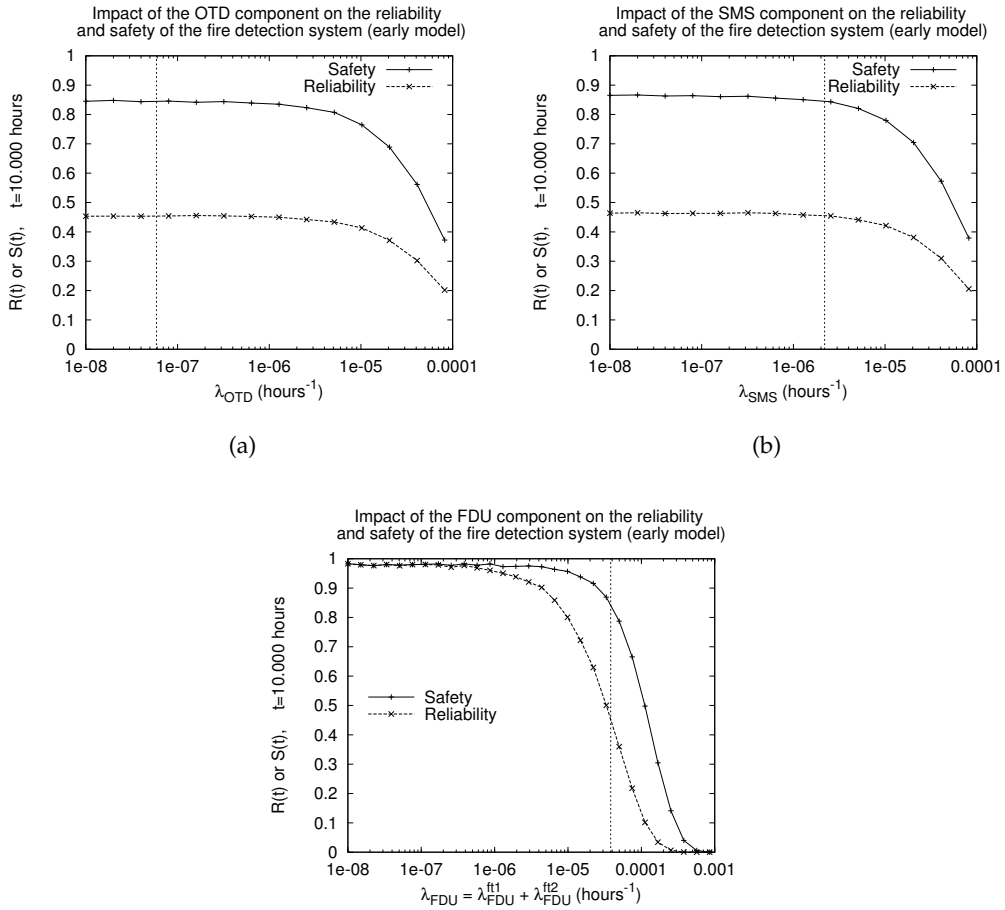
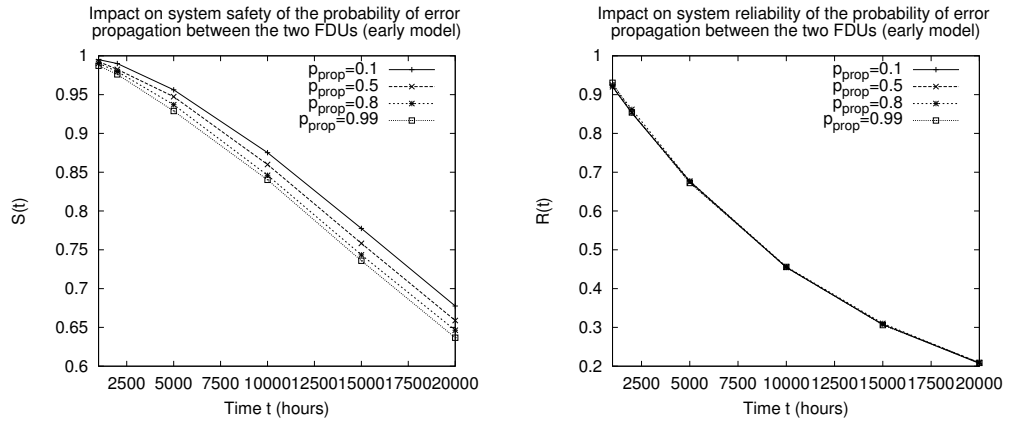


Figure 6.16: Impact of fault occurrence rates of different components on the target system-level metrics.

6.2.4 *System Model – Refinement*

At a later stage, more detailed analysis is performed on the system, following a refinement of the system architecture. The results obtained from the previous evaluation suggest to focus on the FDU component, which appears to be the most critical one. Accordingly, the second refinement details the internal `FDU` architecture, by adding to the already introduced components also the `BKP`, `PSP`, `SLP`, `TLP`, `CPU`, and `I/O` components (see Figure 6.11). Instances of newly introduced components are defined and connected together in a *Composite Structure Diagram* which details the internal architecture of the `FDU` component (Figure 6.18). Input and output ports through which the `FDU` component interacts with the other components of the system architecture are delegated to the sub-



(a) (b)

Figure 6.17: Impact of propagation probability between the two FDUs on the metrics of interest.

component who is in charge to implement such interface. The *smoke* and *temperature* ports are delegated to instances of the SLP and TLP boards, respectively; the *nfsd* port to the instance of the I/O board; the *toOther* and *fromOther* ports to the CPU board. We emphasize that — due to the component-based approach that has been adopted — the previously specified system model has not been modified, but only extended.

The CPU instance is directly connected to the SLP and TLP instances, and to the I/O instance. All the boards are connected with bidirectional ports with the BKP instance, with the exception of the PSP instance, having only an output port on which it provides power to the BKP instance.

Concerning dependability attributes, for the BKP and I/O board, the «StatelessHardware» stereotype is used, while we assume that the PSP board has an error latency of 30 minutes, and therefore we apply the «StatefulHardware» to it. For the SLP, TLP, and CPU boards an error model is instead defined, and applied through the «DependableComponent» stereotype. Finally, the «DependableComponent» stereotype that was previously attached to the FDU component is removed, in order for the transformation algorithm to take into account for the newly defined internal structure of the component.

The error model for the SLP board is shown in Figure 6.19; the one defined for the TLP board is very similar and it has been omitted. The SLP board is affected by two failure modes, Safe, in which smoke sensors data cannot be acquired by the CPU, and Hazardous, in which smoke sensors data cannot be acquired by the CPU and the release of the master status is prevented. The two failure modes are caused by the NoSensorsData and StuckMaster errors respectively, which in turn are caused by two different «InternalFault» elements (*slpft1* and *slpft2*) having different occurrence rates. Moreover, the NoSensorsData error can

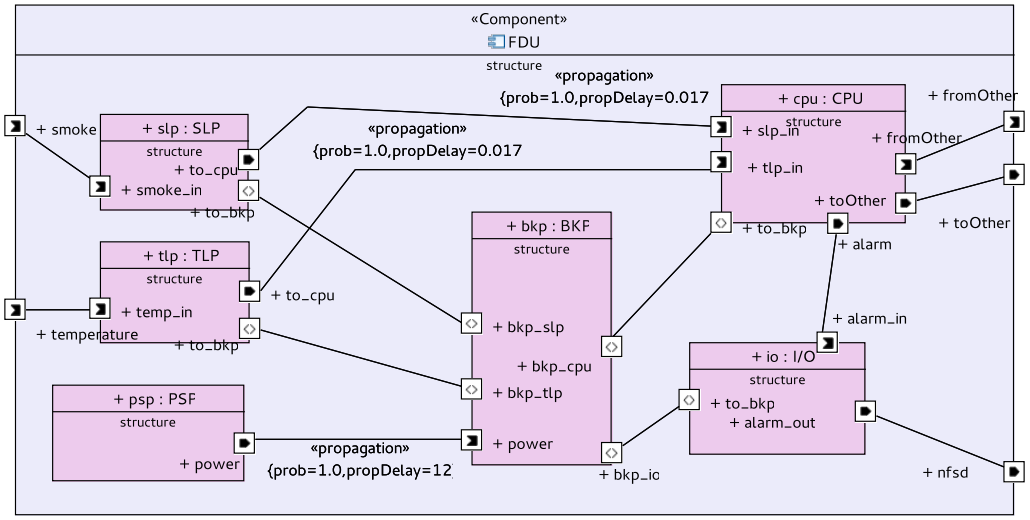


Figure 6.18: Internal architecture of the FDU component.

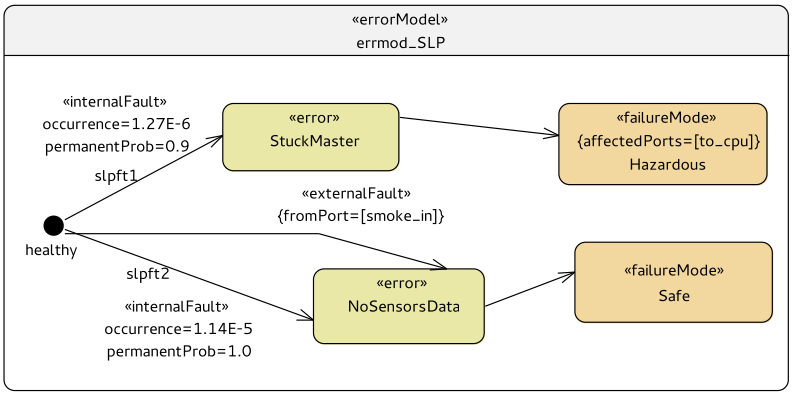


Figure 6.19: Error model for SLP/TLP boards. The figure shows the error model for the SLP board; the TLP board has a similar behavior.

also be caused by an external fault occurring on the *smoke_in* port of the `SLP` component, i.e., by the failure of the smoke sensors block itself.

Figure 6.20 depicts the error model defined for the `CPU` component. This error model is similar to the error model that was defined for the whole FDU component in the previous refinement step (see Figure 6.14). In particular, the same errors and failure modes are considered: a "NoAlarm" failure mode, which propagates through the *alarm* port only, and a "HoldsControl" failure mode, which propagates through the *toOther* port as well. Errors `e1` and `e2`, leading to the `NoAlarm` and `HoldsControl` failure modes respectively, can occur due to two different kind of internal faults, having different occurrence rates. Moreover, they can also occur due to external faults occurring on ports *tlp_in* and *slp_in*, which connect the `CPU` with the `SLP` and `CPU` boards: `e1` occurs when at least one of the two boards fail with the "Safe" failure mode (and none with

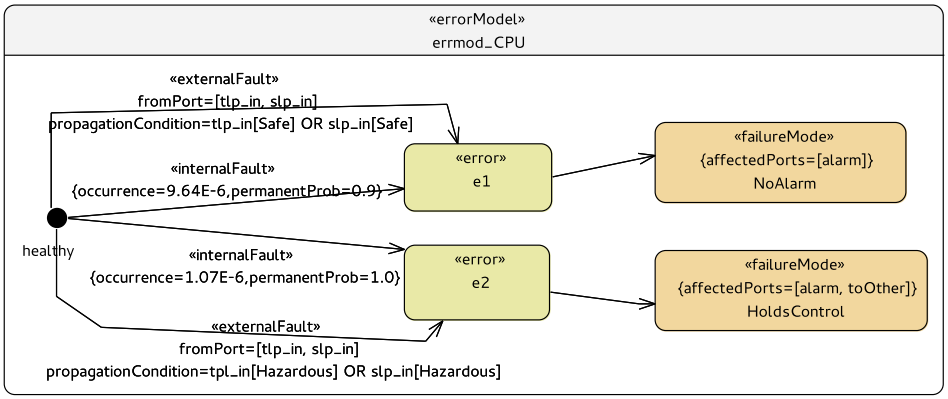


Figure 6.20: Error model for the CPU.

Table 6.3: Additional parameters adopted in the evaluation of the fire detection system.

Symbol	*Corresponding model element*	*Default value*
λ_{SLP}^{ft1}	errmod_SLP.slpft1.occurrence	$1.27 \cdot 10^{-6}$ faults/hour
λ_{SLP}^{ft2}	errmod_SLP.slpft2.occurrence	$1.14 \cdot 10^{-5}$ faults/hour
λ_{TLP}^{ft1}	errmod_TLP.tlpft1.occurrence	$4.74 \cdot 10^{-7}$ faults/hour
λ_{TLP}^{ft2}	errmod_TLP.tlpft2.occurrence	$4.266 \cdot 10^{-6}$ faults/hour

the "Hazardous" failure mode), while `e2` occurs when at least one of them fails with the "Hazardous" failure mode.

In order to properly execute the model transformation, `InstanceSpecification` elements need to be re-generated using the "Build Instances" command. This process generates a total of 52 `InstanceSpecification` elements, including instances of connectors: 20 instances for each of the two FDUs (and its subcomponents), 3 as instances of the `SMS`, `OTD`, and `Switch` components, and 9 as instances of connectors at the higher level of abstraction. Metrics of interest are defined exactly in the same way as in Figure 6.15, and do not need to be modified.

6.2.5 *Analysis and Results – Refinement*

The refined design of the system is then analyzed for a more precise evaluation of system metrics, and for the identification of dependability bottlenecks. The generated IDM model for the refined DEP-UML model contains 402 elements, which result in a PNML model containing 3706 elements (including 852 arcs, 280 places, and 315 transitions). Finally, the resulting DEEM input file consists of 3287 lines. With respect to parameters adopted in the earlier analysis, in this section we use some additional ones (Table 6.3)

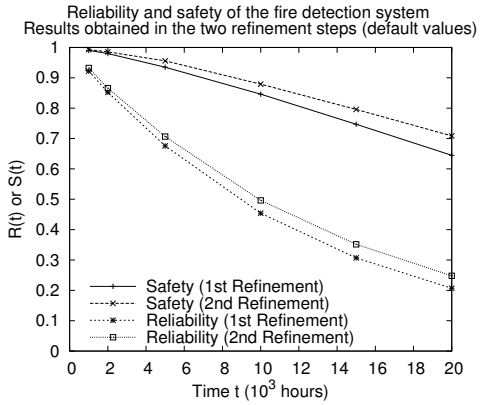


Figure 6.21: Comparison between the metrics of interest evaluated in the two phases. In the earlier design phase reliability and safety of the system have been underestimated.

From the new evaluations performed on the refined system model, it results that the reliability and safety of the system have been underestimated in the earlier design phase, as shown in Figure 6.21. More in details, the values obtained for the target metrics in the refined model are the following:

$$\mathcal{S}(10^4) = 8.789582 \cdot 10^{-1}, \qquad \mathcal{R}(10^4) = 4.962623 \cdot 10^{-1},$$

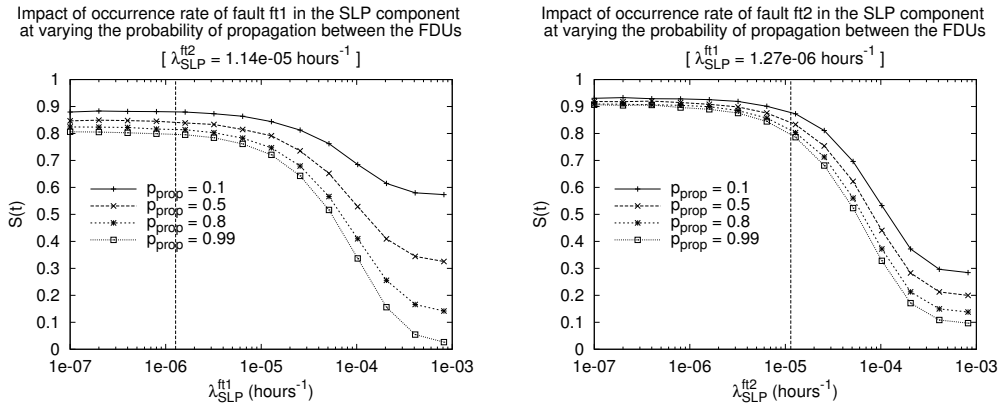
which yield:

$$\text{MTTHE} = 72616 \text{ hours}, \qquad \text{MTTF} = 9852 \text{ hours}.$$

Although there has been a slight improvement in the metrics of interest, such increase is not yet enough to satisfy system requirements. The design process then proceeds by identifying the components which still prevent the system to reach the target values for the metrics of interest. Moreover, the earlier design stage has shown that error propagation between the two FDU has a modest impact on system safety; this evaluation will also aim to further investigate this aspect.

The most peculiar components in the refined architecture are most likely the `SLP`, `TLP`, and `CPU` components, since they all exhibit two different failure modes, one of which may lead to error propagation towards the other FDU. For the above reasons, system evaluation in the second refinement step will focus mainly on the `SLP`, `TLP`, and `CPU` components.

Figure 6.22 shows some of the results that have been obtained from the analysis of the `SLP` component. In particular, the evaluation reported in the figure aimed at assessing the impact on $\mathcal{S}(t)$ of: i) the occurrence rates of the two internal faults affecting the component, and ii) the probability of error propagation between the two FDUs. Results in Figure 6.22a are obtained by varying the



(a) (b)

Figure 6.22: Impact of occurrence rates of the two internal faults of the SLP component on system safety, at varying of the propagation probability between the FDUs.

occurrence rate of *slpft1*, λ_{SLP}^{ft1}, while λ_{SLP}^{ft2} holds its default value; conversely, results in Figure 6.22b are obtained by varying λ_{SLP}^{ft2} only. The vertical lines in both plots highlight the default value of the parameter under study.

The first interesting result is that the two internal faults of the SLP component have a different behavior with respect to error propagation between the FDUs. While in both cases an increase of the fault occurrence rate leads to a decrease of system safety, the decrease due to *slpft2* only slightly depends on the p_{prob} parameter (6.22b); conversely, the decrease due to *slpft1* heavily depends on it (6.22a). However, this behavior occurs only for high values of λ_{SLP}^{ft1}, while for values near its default value the $S(t)$ metric is almost constant.

Another aspect of interest for system design is that further decreasing the occurrence rate of *slpft1* does not yield any improvement in the metrics of interest. Thus, it is not a viable way to achieve the target safety and reliability metrics. Reducing the occurrence rate of the *slpft2* fault can instead provide some improvements to the $S(t)$ metric, which might be sufficient to satisfy the target requirements.

Following such results, and similar ones obtained for the TLP component, we decide to evaluate three different system configurations, in which:

A: the λ_{SLP}^{ft2} parameter is decreased by 50%;

B: the λ_{TLP}^{ft2} parameter is decreased by 50%;

C: both the λ_{SLP}^{ft2} and the λ_{TLP}^{ft2} parameters are decreased by 50%.

From a practical point of view, a decrease of fault occurrence rate can be obtained in different ways, e.g., by adopting more reliable elements in the SLP and

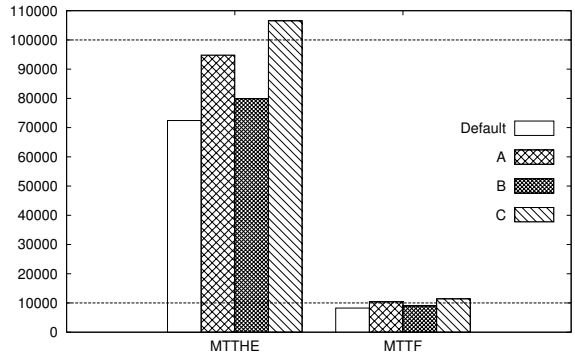


Figure 6.23: Impact of reducing the occurrence rates for faults `slpft2` and `tlpft2` on system safety and reliability metrics.

`TLP` boards, or redesigning the circuitry affected by faults `slpft2` and `tlpft2`, using a different layout.

The results obtained from this evaluation are reported in Figure 6.23, using the MTTF and MTTHE metrics, computed as described in Equation 6.1. The horizontal lines mark the corresponding safety (10^5) and reliability (10^4) requirements. As shown in the figure, both configurations A and C are able to satisfy the MTTF requirement, although by a very small extent. Concerning the safety requirement however, only configuration C is able to reach the required MTTHE value.

A design satisfying system requirements has been then identified (configuration C); this refinement step imposes therefore new requirements on FDUs subcomponents, in particular on the SLP and TLP boards. Further refinement could, for example, proceed to a more detailed analysis of them. It should be noted that, although the MTTHE requirement is met by this configuration, additional countermeasures could be introduced to further increase system safety. For example, additional maintenance actions could be planned to periodically check failed FDUs, and possibly repair or replace them.

6.3 SUMMARY

This chapter focused on the application of the DEP-UML language and tools to two case studies. The case studies highlighted different characteristics of the approach and language, and demonstrated its applicability for quantitative dependability evaluation of different kinds of systems and different kinds of requirements. The support for hierarchical and incremental modeling allows models to be extended and refined in different phases of system development.

The remaining part of this dissertation builds on the observation that, for certain systems, it is not always possible to completely follow their design and

development process. Still, even the evaluation of these systems should be able to benefit from MDE techniques and approaches.

7 MODELING LARGE-SCALE COMPLEX SYSTEMS

While for embedded systems it is often possible, or even mandatory, to follow and control the whole design and development process, the same does not hold for other classes of systems and infrastructures. In particular, large-scale complex distributed systems don't fit well in the paradigm proposed by the CHESS project, and alternative approaches are therefore needed. In this chapter we discuss the challenges in model-based evaluation of large-scale complex systems, highlighting some existing countermeasures and current gaps, with the aid of a motivating example of a Distributed Interactive Multimedia Application (DIMA). In the subsequent chapter we will then propose an approach which applies MDE techniques to fill the identified gaps.

7.1 LARGE-SCALE COMPLEX SYSTEMS

Critical Infrastructures (CIs), meaning those infrastructures whose disruption would have a debilitating impact on the society (e.g., telecommunications, power systems, transportation), are typically deployed on very large geographic scales (often nation-wide), and thus comprise a very large number of components, organized in a hierarchical structure, and driven by complex interactions.

For several decades, such systems have been mostly isolated systems, disconnected from the outside world and using proprietary components and protocols. This has slowly but continuously changed, to the point that it is not uncommon anymore for CIs to use off-the-shelf components or to be (at least partially) connected to the Internet. Such shift in their architecture introduces additional interdependencies [161] between system components, thus exacerbating the already severe complexity problem.

On the other hand, the reduced cost, increased performance, and overall improved quality of electronic devices is leading to an increasing trend to deploy highly distributed and interconnected systems. Furthermore, the wide availability of wireless connectivity is opening up new possibilities that were not possible in the past, e.g. car platooning via ad-hoc wireless networks [22]. Such new systems, or extensions to existing systems, exhibit strong commonalities with CIs, at least in the challenges raised in the model-based evaluation of their performability properties.

In general, model-based analysis of modern systems is facing great challenges: their scale is growing, they are becoming massively distributed, inter-

connected, and evolving. The high number of components, their interactions, and rapidly changing system configurations represent notable challenges.

It should be noted that a large amount of work in literature proposes techniques for the generation, handling, and numerical evaluation of large state-space models. Such techniques are mostly aimed at the *evaluation* of large state-space models, and employ numerical techniques for efficient state-space representation and/or efficient model solution.

While techniques for the efficient evaluation of large state-space models are paramount, the growing complexity of modern systems is also posing challenges for the *specification* of analysis models themselves. Models should be able to cope with the high dynamicity which typically characterizes such systems, evolving requirements and architecture, and events originated from the environment. Therefore, evaluation models should be able to rapidly adapt to new system conditions in order to evaluate if, and how, its properties have changed, and possibly suggest reconfiguration actions [30].

7.2 THE "TEMPLATE MODELS" APPROACH

A key principle in addressing the complexity in modeling large systems and infrastructures is *modularization*, i.e., following the principle to build complex models in a modular way through composition of smaller submodels. While compositional modeling approaches (e.g., [158, 166]) were initially introduced for the purpose of reducing the size of the generated state-space, such approaches have later gained importance also in improving the *specification* of models, since they also carry with them a number of other practical advantages: submodels are usually simpler to be managed, can be reused, can be refined, and can be modified in isolation from other parts of the model.

We focus on the domain of Stochastic Petri Nets, and in particular on their composition through *place superposition* (i.e., place sharing), which is quite common in performability modeling. Approaches using *transition superposition* (i.e., synchronization) have also been introduced in the literature [69]. However, meaningful composition in this case requires to understand how transition properties — firing delays, guards, priorities — are composed, which can be addressed in different ways, and does not have a unique solution. However, it usually requires further constraints to be imposed on submodels (e.g., synchronizing transitions must have the same rate), or additional information to be specified during composition.

7.2.1 *Template Models and Parameterization*

Focusing on place sharing, several approaches based on Petri nets have recognized the benefits in applying separation of concerns principles [65] to the

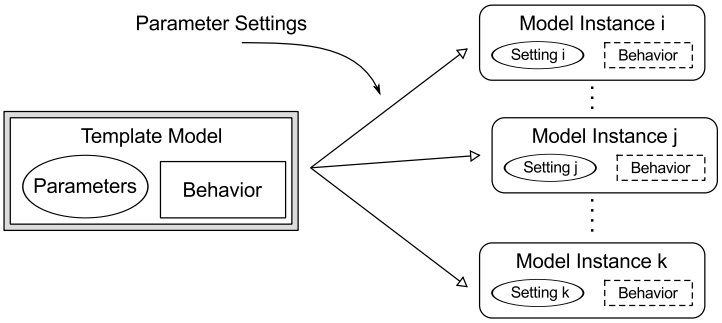


Figure 7.1: Template models and parameterization.

construction of performability analysis models. In such approaches (e.g., see [43, 102, 160, 169, 182]) the overall analysis model is built out of well-defined submodels, addressing specific aspects of the system, which are then composed by place superposition. The overall composed model is obtained by following *predefined composition patterns*, based on the actual scenario to be represented.

In performing system decomposition, particular attention is devoted to the identification of the *interfaces* between the different submodels, thus improving the modularity of the model: submodels can be modified in isolation from the rest of the model, can be replaced or refined as needed, provided that their interfaces remain the same, can be rearranged based on modifications in system configuration. This approach also eases the integration with external tools: a given submodel, implementing a specific function, may be replaced with a specialized external tool, either directly or through a "wrapper" model.

Another dual aspect that enhances the modularity of submodels is the identification of their parameters. In large-scale systems, different components may have a similar behavior, only differing by some parameters that are specific of a particular instance of the component, depending on its role in the system, or the environment in which it is operating.

This process leads to the definition of "template" submodels, which are composed of two parts: a part defining its behavior, and a part defining its parameters (Figure 7.1). In the construction of the overall model these templates are then instantiated multiple times, with different parameters settings. This approach saves the modeller from manually create (and maintain) multiple models for components having a similar behavior, which is a very time-consuming and error-prone task. Also, any change in a template model is automatically reflected in all the instances of that template.

When carried to its extremes, this approach leads to a modeling paradigm that resembles Object Oriented Programming (OOP): "libraries" of template submodels are created for a given system, having fixed "interfaces" and "parameters". Such templates are then "instantiated" multiple times and then connected through their interfaces. Actually, when referring to submodels and their

properties, to stress their encapsulation some of the existing work actually use OOP-derived terms like *template* [169], *interface*, *instance* [23], *inheritance* [13].

7.2.2 *Application Using Stochastic Activity Networks*

The Stochastic Activity Network (SAN) formalism [167], which can be considered an extension of SPNs, and the supporting Möbius framework [55] provide useful primitives for applying the aforementioned approach.

SAN models are composed using the Replicate/Join state-sharing formalism [166]: the *Join* composition operator is used to compose two or more submodels, by sharing places between them; the *Replicate* composition operator is used to combine multiple identical copies of a submodel, also sharing places between them; such operators can be applied iteratively, thus forming a tree structure.

The support for the definition of model interfaces and parameters is provided by special kinds of places that can be added to SAN models, called "extended places" [55]. Such places are not limited to hold an integer number of tokens, but instead can hold an instance of a given datatype, including most of C++ basic types, arrays, as well as structured data types. Thanks to extended places, interfaces between submodels can be defined in a very convenient way.

Template submodels are then instantiated and their interfaces combined using the Replicate/Join formalism. Parameters places are usually shared with an ad-hoc "Setup" submodel, which has the only task of initializing the parameters of the different submodels, based on the desired scenario (e.g., see [113]). It is worth noting that composed SAN models constructed in this way are most often solved by discrete-event simulation, due to i) the large state-space of the overall composed model; and ii) the violation of constraints for their analytical solution that are imposed by Möbius solvers [127], e.g., the existence of a vanishing initial state.

7.3 MOTIVATING EXAMPLE: A WORLD OPERA

As motivating example we introduce the World Opera (WO) system described in [183]. The WO consortium and its partners are engaged in conducting distributed, real-time, live opera performances across several world renowned opera houses. Each opera house represents a real-world stage with its own musicians, singers, dancers, and actors. Interaction between the artists is orchestrated by a single conductor present at a single selected stage. Participating artists from different real-world stages map to *virtual-world stages*, which are projected as video on display devices, and shown to the audience at the local opera house as well as audiences at geographically distributed (remote) opera houses. Additionally, virtual-world stages can display animated cartoon characters mimicking the behavior of the artists at remote stages. The virtual-

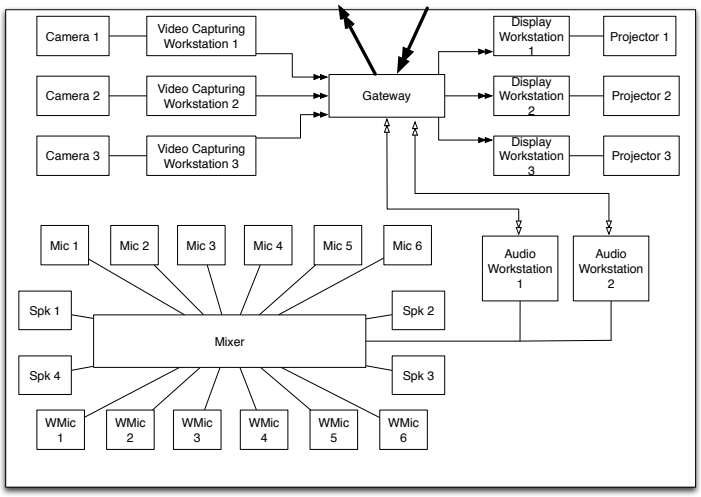


Figure 7.2: System architecture of a World Opera stage [183].

world and real-world stages together form a mixed-reality stage. A collection of distributed mixed-reality stages together constitute a WO application.

Even though it may not be strictly defined as a "critical" infrastructure, from an architectural point of view it exhibits many of characteristics typical of large-scale systems. It is composed of etherogeneous elements, which collaborate to provide different services within the system; the number of different types of such elements is small, while the actual number of instances of such elements is high; instances of such elements have different settings but similar behavior; based on the service provided by the system, the actual interconnection between system components may differ.

The typical setup for a World Opera performance consists of 3 to 7 real-world stages with different artists and possibly a different set of technical components (microphones, projectors, etc.). The activities at each stage in WO are logically divided into four tasks. *Capturing* involves corresponding components receiving activation signals and generating streams. There exist three principal stream types: video, audio and sensor (e.g., to track the movement of an artist on the stage). *Processing* is then performed on all generated streams to remove noise. Additionally, video streams are encoded to reduce the size of streams, timestamped, and processed using computer vision techniques for artistic reasons. *Streaming*, involves transmitting and receiving the streams to and from the remote stages. Finally, *rendering* involves processing (e.g., decoding) received streams, synchronizing them based on their timestamps and then rendering them to the virtual-world.

The architecture considered for a general WO stage is shown in Figure 7.2. A stage consists of: microphones and cameras to capture the multimedia streams from actors; projectors and speakers to render the streams to the audience; a

mixer to route audio streams; workstations for processing of captured streams; a gateway for transmission of streams to/from remote stages. The actual number and kinds of components present in each stage depend on the number of artists present in the stage, and the role of the stage in the overall show.

The infrastructure allowing for these applications includes a high number of specialized hardware and software components, whose slight malfunction of these components could severely affect the show[1] due to the strict functional requirements. Fault-tolerant architectural solutions are therefore necessary to ensure the correct execution of a WO show. However, in order to design such a fault-tolerant solution it is essential to understand the interaction between the components and the potential effects of their failure, from the audience perspective, on the overall show.

7.4 PERFORMABILITY MODEL OF THE WORLD OPERA SYSTEM

A set of meaningful metrics for the evaluation of the WO system have been defined in [182], and evaluated on a simplified case study. A more complex and faithful model of the system has been developed using a modular modeling approach as discussed before [132]. In the following, we briefly describe the identified template models and their interfaces, in order to support our discussion; further details can be found in [132].

The model considers *components* and *streams* as the basic elements of a WO show, both having different possible working states (e.g., working/failed for components, good/missing/delayed for streams). The state of a stream in a certain physical point of the architecture depends on the state of all the components that have processed it so far (including components that are capturing it). The state of different streams as they are reproduced to the audience provides insights on the QoS perceived by the users and it is therefore the main target of evaluation.

In the modeling of the WO system 4 basic SAN templates are involved [132]:

- *Component*: A physical component of the WO architecture. The interfaces of this submodel are `working_state`, holding the current state of the component, and `num_f` for each failure mode `f`, counting the number of components in the same group that have experienced failure mode `f`.
- *StreamCollector*: Models the capturing of a stream. Its interfaces are `num_f` for each failure mode `f` of associated capturing components, and `stream_out`, which represents the state of the captured stream. The model sets the state of the captured stream (place `stream_out`) based on the state of components that are capturing it.

1 In order to prevent ambiguity, here we avoid using the term "performance"to refer to the artistic exhibition of actors and musicians.

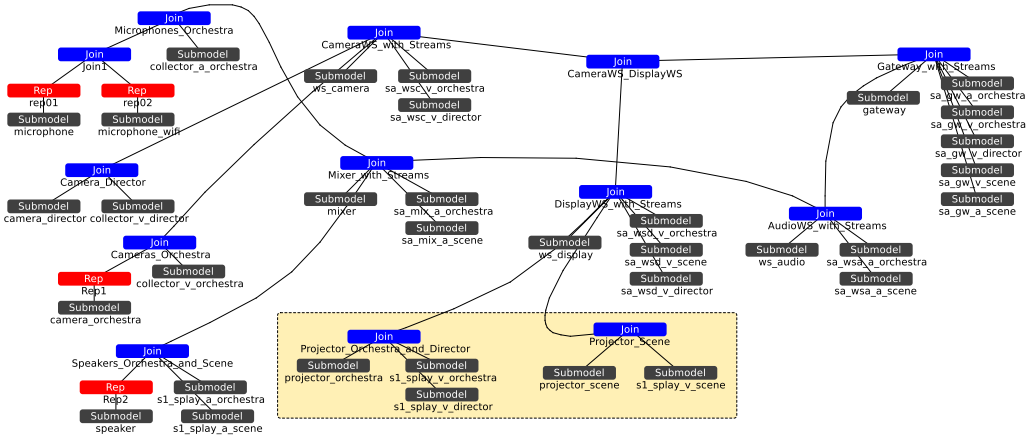


Figure 7.3: A SAN composed model for a WO stage, built out of the 4 identified SAN templates [132].

- *StreamAdapter*: Models the processing of a stream. Its interfaces are `component_state`, representing the state of the associated processing components, `stream_in`, representing the state of the stream received as input, and `stream_out`, representing the state of the stream produced as output. The model sets the state of the output stream based on the state of the state of the component, and the state of the received stream.

- *StreamPlayer*: Models the playback of a stream to the audience. Its interfaces are the state of the stream as received as input, `stream_in`, the state of associated playback components, `num_f`, and the state of the stream as reproduced to the audience, `stream_play`. The model sets the state of the reproduced stream based on the state of received stream, and the state of playback components.

These templates have also a number of parameters; for example, the failure rate and the number of spares in case of components. The actual implementation of these models depends on the behavior to be modeled; for simplicity, we assume here a hot-spare behavior for all components as in [182], thus leading to a single "Component" template.

By just composing multiple instance of these basic building blocks, a wide variety of different scenarios can be assessed. As an example, Figure 7.3 shows the composed model for a WO stage consisting of 6 microphones, 2 cameras, 4 speakers, 2 projectors, 3 workstations, 1 mixer and 1 gateway, and employed in a WO show consisting of 5 application streams (3 video streams and 2 audio streams). The model consists of 33 atomic model instances, created from the fixed set of 4 templates listed above. Actually, such model is only a part of a bigger model addressing a three-stage WO performance, comprising ~50 components, modeled by ~100 instances of the 4 template models described above.

However, the manual construction of such a model is a cumbersome task since, based on the target system configuration: i) the proper number of template instances must be selected and instantiated; ii) the graph structure of the model (i.e., which node participates in which Join and Replica nodes) should be defined; and iii) most importantly, the proper place superpositions must be applied in order for the overall model to exhibit the intended behavior.

7.5 CURRENT LIMITATIONS

Currently, the applicability of this approach is hampered by two major practical limitations: i) the lack of formalization of template composition patterns, and consequently ii) the lack of means for automated application of such patterns.

Following the introduced example, some rules for creating instances in the WO model are:

- *For each application stream to be modeled create an instance of the "StreamCollector" template model*
- *For each component of the stage create an instance of the "Component" template, and for each of the streams that are processed by the component, create and instance of the "StreamAdapter" template.*

Unfortunately, it is not sufficient to simply add atomic model instances to the overall model. Referring to Figure 7.3, for each "Rep" or "Join" node (red and blue nodes, respectively) there exist a precise pattern of place sharing to be followed in order to correctly assemble all the instances of atomic templates (black nodes). Manually performing such a task requires considerable effort at the increasing of model largeness. Examples of rules for connecting instances in the WO model are:

- *For each stream in the scenario, connect each "StreamCollector" instance with the instances of "Component" templates corresponding to components that are used to capture the stream. Interfaces to be connected are* `num_f` *in all models.*
- *For each processing component, connect "StreamAdapter" instances with the corresponding instance of the "Component" template. Interfaces to be connected are* `working_state` *and* `component_state`*.*

For large systems actually remembering or following such patterns is difficult; even more if modifications occur in system configuration. For example, the highlighted part of Figure 7.3 models the playback of three video streams (*v_orchestra*, *v_director*, *v_scene*) on two different projectors (*projector_orchestra* and *projector_scene*). Adding a third projector dedicated to the *v_director* stream only would require to: i) add a new instance of the "Component" template for

modeling the new projector, ii) removing the corresponding instance *s1_play_v_director* of the "StreamPlayer" model from the "Projector_Orchestra_and_Director" Join node, iii) properly connect the new "Component" instance and the existing *s1_play_v_director* instance into a new Join node, and iv) properly connect the new Join as child of the "DisplayWS_with_Streams" Join.

It should be noted that the Möbius framework [55] provides some means for reducing the effort required to specify complex models. Actually, its implementation of the Rep/Join composition formalism [166], allows multiple instances of the same SAN [167] model to be reused. However, instances of the same submodel are *completely identical* in Möbius, and each instance still needs to be manually connected to the rest of the composed model.

While established formalisms exist both for defining the submodels (e.g., SANs) and for physically composing them (e.g., Replicate/Join), the patterns to be followed for their composition — which depend on the system to be modeled, and the set of identified submodels — are typically not formalized. In many cases, they are provided either informally or by examples. Even worse, sometimes those "rules" are not even written somewhere, but they are only known to the person(s) that developed the submodel library. As a result i) submodels libraries are difficult to be shared and reused, and ii) the overall system model for different scenarios must be assembled by hand by people who know the appropriate rules to follow. Furthermore, even when rules for the composition of submodels have been properly specified, obtaining a *valid* (i.e., correctly assembled) model requires a lot of manual effort. In the next chapter we aim to address this problem using MDE techniques.

Within the performability community, the approach which is more related to ours is the one proposed by the OsMoSys framework [184]. In particular, it also promotes an Object-Oriented modeling approach, using OO-derived terms like "model class", which have strong analogies with terms adopted in the next chapter. The focus, and consequently the approach, is however quite different. OsMoSys provides a way to compose performability models created with different formalisms, and to orchestrate their solution in order to evaluate the global system model. Our focus is on the *reuse* and *automation* of composition patterns for a specific class of formalisms, aiming at enabling the automatic assembly of large performability models with reduced effort for the user.

8

A WORKFLOW FOR AUTOMATED ASSEMBLY OF COMPLEX MODELS

In this chapter we propose an approach aiming to fill the gap in the application of the previously presented "template models" approaches, by introducing a workflow based on: i) a custom language to precisely specify template models libraries and composition patterns, and ii) automated model-transformations to automatically instantiate the model templates and properly connect the overall system model, based on library-specific patterns.

8.1 WORKFLOW OVERVIEW

The proposed approach is based on the Template Models Description Language (TMDL), an ad-hoc language to describe template models, which is composed of two parts: a "Library" part, and a "Scenario" part. The overall workflow is summarized in Figure 8.1 and described in the following.

Step 1. Starting from system requirements and architecture, an expert in performability modeling develops a "Template Models Library", i.e., a library of reusable submodels and composition patterns. Such library is composed of two parts:

- *TMDL Library*: A specification in the TMDL language, containing i) the definition of elementary template models and their interfaces, and ii) a set of valid composition templates, which specify valid patterns for composing template instances.
- *Templates Implementation*: The internal implementation of template models that have been specified in the TMDL Library. This part depends on the specific tools and formalisms that are adopted in the workflow. As an example, the PNML language [90] can be used as storage format.

Step 2. Once the template models library has been established, the different system scenarios and configurations that should be analyzed are defined. This step is performed using the TMDL language as well, with a specification which corresponds in selecting and instantiating (i.e., assigning values to their parameters) a set of templates specified in the model library. From a practical perspective, TMDL "Scenario" specifications can be created manually starting from informal descriptions of scenarios to be analyzed, but could be also automatically generated from UML-like architectural models, by applying model-transformation techniques. In this perspective, TMDL "Scenario" can also be

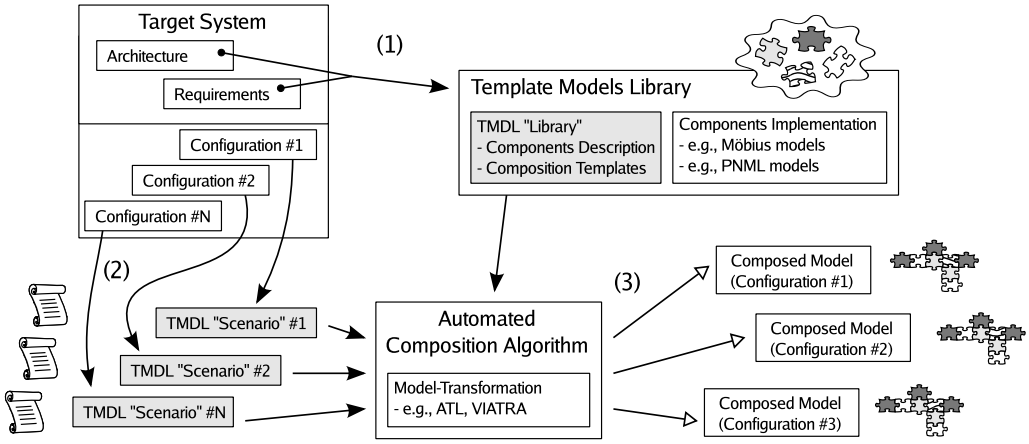


Figure 8.1: Our workflow for the automated generation of performability models. Elements depicted in gray are specified using the TMDL language.

considered as a sort of intermediate model between an architectural system description, and the SPNs model that is going to be generated.

Step 3. Starting from the Template Models Library defined in Step 1, and the description of scenarios provided in Step 2, the performability models for all the different scenarios and configurations are automatically assembled and then evaluated. The generation of composed models is accomplished by means of the "TMDL Automated Composition Algorithm", which takes as input a "TMDL Scenario" specification and generates the corresponding evaluation model by properly assembling instances of "Template Implementations" based on the patterns specified in the "TMDL Library" (see Figure 8.1).

It is important to note that the "Automated Composition Algorithm" of Figure 8.1 is *the same for every library of template models*, i.e., it is defined and implemented *only once*, and can be reused to automatically assemble models describing different systems[1]. This it the key point of our approach, and the main reason to develop the TMDL language. Also, we aimed at a more general approach with respect to defining a set of model transformations specifically tailored to the WO system.

The workflow defined above has strong analogies with concepts encountered in Component-Based Development (CBD), and in particular with the concept of "composition system". According to [4, 87], a *composition system* is defined by three elements: i) composition technique, ii) component model, and iii) composition language. The *composition technique* defines how components are physically connected, while the *component model* defines what a component is and how it can be accessed. The *composition language* is used to specify components and "composition programs", i.e., a kind of script that specifies which components should be connected, and how, in order to obtain the intended composed

1 Provided of course that different template models libraries have been defined.

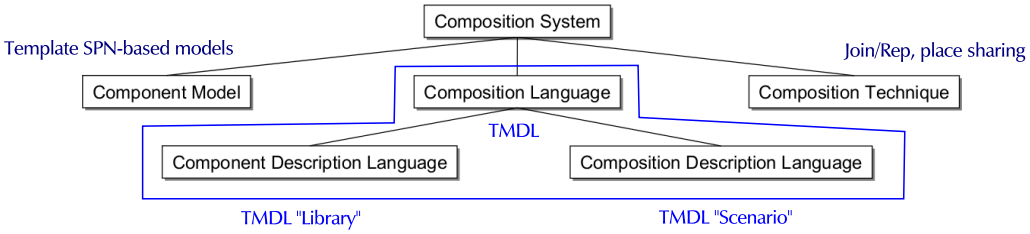


Figure 8.2: Relations of elements included in our workflow with the notion of composition system. Original picture from [87].

structure. Composition programs are interpreted by a *composition engine*, which in our workflow is represented by the "TMDL Automated Composition Algorithm".

Therefore, reformulating our previous discussion, for realizing our target "composition system", languages to specify components exist (i.e., template models based on SPNs and their extensions), as well as the composition technique (i.e., Rep/Join and state-sharing in general). What is currently missing is a composition language capable to specify how submodels should be assembled; Such role is played by the TMDL language introduced in the following. moreover, a composition language consists in a *component description language*, to specify components, and a *composition description language*, to specify composition programs [87]; such two aspects correspond exactly to the "Library" and "Scenario" portions of TMDL (Figure 8.2).

8.2 MAIN CONCEPTS

In the following we first introduce the main concepts on which the TMDL language builds.

The basic building blocks of TMDL are *model templates*. Model templates have a set of *interfaces*, which specify how they can be connected to other model templates, and a set of *parameters*, which specify variable elements in the component (e.g., the initial number of tokens in a certain place for a SAN model). Component templates can be either *atomic templates* or *composition templates*. A set of model templates constitutes a *library*.

Atomic templates are associated to an *implementation* in the selected state-based formalism (e.g., a PNML file, or a tool-specific format like the XML-based format used by the Möbius tool). Composition templates contain references to a set of *submodels*, i.e. other model templates. For each of the referenced submodels a *multiplicity* attribute may be specified. Composition templates can have parameters as well, which allow for example parametric multiplicity values to be specified. Composition templates include a set of *merging rules*, which specify the patterns for connecting the interfaces of their submodels.

A *model class* is obtained from a component template by associating concrete values to its parameters, and it is defined by a reference to a template model, and possibly a set of values for its parameters. In addition to the set of values for its parameters, a *composed class* can also contain references to other model classes, which are used as concrete submodels, provided that they are compatible with the specification of the template.

A model class can be used more than once in the overall composed system model, e.g., in case multiple identical elements are present in the system. A *model instance* is an individual instance (copy) of a model class. An *atomic instance* is a copy of the template implementation, where all the parameters have been set as specified by the model class. A *composed instance* is an instance of a composed model class, i.e., a collection of model instances composed according the specified patterns.

Submodels of a composition can in turn be other composed models. Therefore, a composed instance can be considered the root of a tree, in which internal nodes are associated with other composed instances, and leaves are atomic instances. The overall model that represents a certain scenario is therefore identified by an instance of a composed component. Accordingly, a *scenario* (i.e., our "composition program") is specified as a set of model classes, and a "root" instance that provides what could be called the "entry point" of the composition program.

Multiplicity allows multiple model instances to be specified by specifying the model class and a numeric value. In the model generation phase, an index is automatically assigned to each of the generated instances, allowing them to be distinguished from each other. By default, indices are set based on the multiplicity of the model instance, i.e. a multiplicity of n generates n instances, with indices ranging from 1 to n. For greater flexibility, TMDL allows the user to directly specify an array of indices in place of a *multiplicity* value. For example, by specifying $\{3, 4, 5\}$ as a multiplicity value, in the model generation phase 3 identical model instances will be created, having indices 3, 4, and 5. Moreover, indices should be associated with textual prefix, thus allowing to distinguish indices related to different dimensions.

When any interface of a submodel becomes an interface of a composed model, an index (and possibly a prefix) is appended to its name. Such name is the *instance name* of the interface.

8.3 TEMPLATE MODELS DESCRIPTION LANGUAGE

The metamodel of the TMDL language is shown in Figure 8.3. For simplicity, only the main language elements are shown in the figure: data types and other supporting elements have been omitted. As described before, the purpose of TMDL is twofold: to define libraries of template models, and to define scenarios

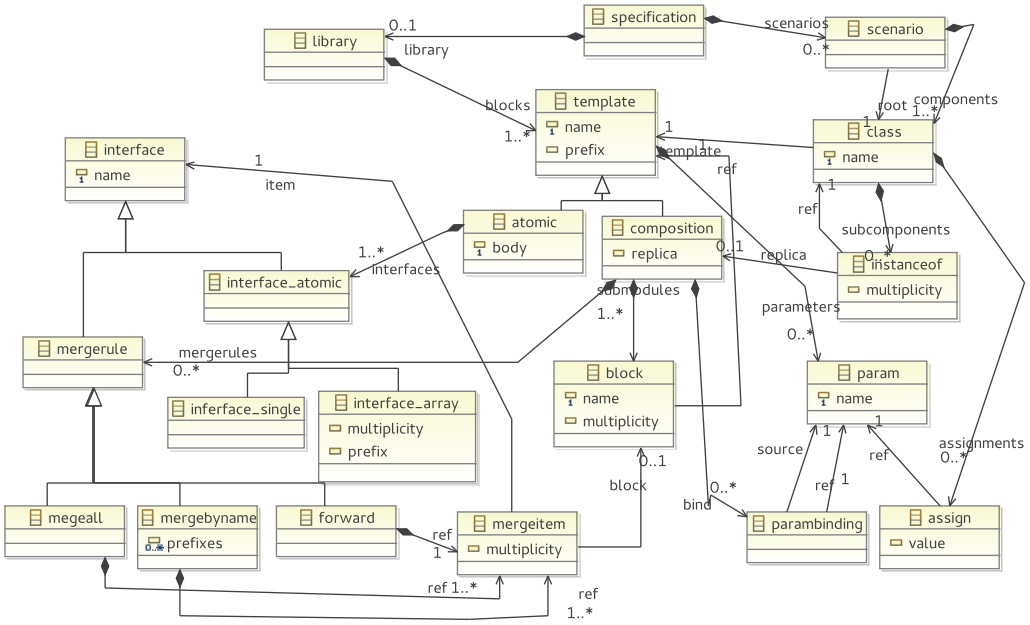


Figure 8.3: Simplified version of the TMDL metamodel. For simplicity, data types and other supporting elements (e.g., arrays) are not shown in the figure.

in which such templates should be instantiated. For this reason, a specification in the TMDL language (`specification` element) may contain a model library (`library` element), and a certain number of scenarios (`scenario` elements).

8.3.1 *TMDL "Library"*

A `library` is composed of a set of `template` elements. Each template has a distinguished *name*, and a set of parameters (`param`). The `template` element is an abstract element, which is refined in atomic templates (`atomic`) and composition templates (`composition`). Each template has a distinguishing *name*, and a textual *prefix*, which can be specified for indexing purposes. When the Boolean *replica* attribute is set, the composition template is simply a replica of another template; in this case the composed template can be used in any place where the template it replicates is expected.

An `atomic` template has a *body* attribute, which specifies where to actually find an implementation of such model, and a set of atomic interfaces (`interface_atomic`). An atomic interface can be either a single interface (`interface_single`), or an array of related interfaces (`interface_array`). In the latter case, the interface has associated a *multiplicity* value and, optionally, a textual `prefix`.

A `composition` template contains a set of `block` elements, which specify which kind of templates are allowed to be used as submodels for the composition. Each `block` element has a distinguished *name*, and references a `template`

element. A *multiplicity* value can be specified to define multiple instances of the same model class as submodels.

Additionally, a `composition` template specifies a set of rules that should be followed in connecting together its subcomponents (`mergerule`). Which interfaces are selected for each `mergerule` is specified by `mergeitem` elements. Each `mergeitem` element references a single `interface` element, and optionally, a specific `block` element and a *multiplicity* value. If a `block` is specified, the rule is restricted to subcomponents derived from such block only; similarly, if the *multiplicity* attribute is specified, the rule is restricted to subcomponents having the specified indices only. Each `mergerule` contains one or more `mergeitem` elements.

Three kinds of merge rules are supported by TMDL: `mergeall`, `mergebyname`, and `forward`. The `mergeall` rule specifies that all the selected interfaces should be connected together, to form a single interface of the composed component. The `forward` rule specifies that a single interface of a subcomponent should directly become an interface of the composed component; in this case no interfaces are joined together. The `mergebyname` specifies that, within the selected interfaces, those with the same *instance name* should be merged together, to form a single interface of the composed component. The instance name is formed by the "base name", i.e., the name specified in the composition template, and any indices and prefixes appended during model generation. Without further parameters, instance names need to be exactly the same for merging to occur. The user may however specify a set of *prefixes* to which the comparison should be restricted. If some of the selected interfaces cannot be merged with any other interface they are forward as interfaces of the composed model.

Finally, a `composition` template may specify a set of bindings between its parameters and parameters of its subcomponents (`parambinding`); in such case, parameters of submodels are constrained to hold the same value as parameter of the parent model.

8.3.2 *TMDL "Scenario"*

A `scenario` is composed of a set of classes (`class` elements). Each `class` has a distinguished *name*, and references a specific `template` in the model library. Moreover, a `class` may contain a set of `assignments`, which specify concrete values for the parameters specified in the component template.

In case of a composed class, i.e., a `class` element which references a `composition` template, submodels may be explicitly defined with `instanceof` elements. Each `instanceof` element references another model class in the same scenario, and possibly a *multiplicity* value. An `instanceof` element may also specify a *replica* behavior, and a "replica" `composition template`. In this case

the selected model class is first replicated using the specified replica template (provided that they are compatible).

It should be noted however that `instanceof` elements are not always needed. When a model template has no parameters, or only one model class derived from it exists in the scenario, submodel instances are automatically generated by the transformation algorithm, based on the template specified in the submodels library, since in this case no ambiguity exist on parameter values to apply.

Finally, the *root* attribute of the `scenario` element defines the model class which, once instantiated, represents the overall system scenario.

8.4 MODEL GENERATION OVERVIEW

The third step of the workflow in Figure 8.1 is performed by the automated model generation algorithm, which is organized in two phases: *instances generation*, in which component instances are generated, and *instances composition*, in which the generated component instances are connected together.

The generation algorithm uses two data structures: a *queue* Q containing model classes that still need to be instantiated, and a *stack* T containing model instances that have been instantiated but whose interfaces still need to be connected. More in details, given a TMDL "Library" $\mathcal{L}$, and a TMDL "Scenario" $\mathcal{S}$, the steps to assemble the overall performability model are summarized in the following:

— *Instances generation* —

1) Based on the *root* element in $\mathcal{S}$, the root model class c is identified. The pair $\{c, 1\}$ is enqueued in Q.

2) The pair $\{c, m\}$ is dequeued from Q:
   a) instances of c are created based on multiplicity m;
   b) an index is assigned to each instance;
   c) each instance is pushed into the stack T.

3) If c is a composed class:
   a) For each model class c_i, referenced as submodel of c with multiplicity m_i, the pair $\{c_i, m_i\}$ is enqueued in Q.
   b) If the composition template corresponding to c requires additional submodels that have not been specified in the scenario, the corresponding default pairs $\{c_j, m_j\}$ are created, based on default values of template parameters, and enqueued in Q.

4) If Q is not empty then go back to Step 2. Otherwise stop: all the instances required to obtain a valid model have been created.

— *Instances composition* —

1) The instance i is removed from the stack T.

2) If i is a composed instance, then the interfaces of its submodels are connected based on the rules defined by the related model template.

3) If T is not empty, then go to Step 1. Otherwise the whole model generation process ends.

During the instances generation phase, for each model template links to the instances that it has generated are stored, so that pattern specified in different templates can be applied to the proper model instances.

8.4.1 *Prototype Realization*

It is commonly agreed [185] that the development of custom DSLs and the related model-driven workflows is a complex task that should be addressed with an *iterative* process. Useful feedback for the formalization of domain concepts is obtained by the definition and implementation of the language; feedback for the definition of the language is obtained by the definition and implementation of model-transformations; feedback on model-transformations is obtained by validating the produced artifacts.

According to this view, a prototype of the entire workflow has been realized, based on the Eclipse platform, and it will guide us through the validation and refinement of the entire approach. The TMDL meta-model has been defined as an Ecore model using EMF [70], while Xtext [189] has then been used to define a textual syntax, and to generate the editor and parser for the language.

Concerning the model composition and generation algorithm, it shall actually be composed of two phases: i) generation of required instances and assignment of their parameters; ii) composition of generated instances; and iii) the actual copy of template implementations to populate the generated structure. The first two steps have been realized using the ATL language [5], while the third, uses simple XSL Transformations (XSLT) [188] to manipulate XML models, i.e., to substitute parameters placeholders with the corresponding actual values obtained from the previous steps.

8.5 APPLICATION TO THE WORLD OPERA SYSTEM

In this section we describe the application of the proposed approach to the WO model described in Section 7.4.

8.5.1 *Library Specification*

As introduced before, the performability model for a World Opera show consists of 4 atomic templates: *Component*, *StreamCollector*, *StreamAdapter*, *StreamPlayer*. For simplicity, we assume here that components are subject to two failure modes: a "silent" failure mode, in which the component just stops working, and a "noisy" failure mode, in which the component produces noisy/incorrect output. Selected portions of the the TMDL "Library" specification for this system are shown in Listing 8.1. Ellipses (...) have been used to mark parts of the specifications which have been omitted.

Atomic templates are defined in lines 1–16; for each of them, an implementation of the model is referenced using the *body* attribute. The atomic template *component* (lines 1–9) has five parameters: *failrate*, which specifies the failure rate of the component, *spares*, which specifies the number of spares allowed for the component, *fprobnoisy*, which specifies the probability that the component fails with the noisy failure mode, *sw_delay* and *sw_prob*, which specify the delay and failure probability of switching to a spare component. The *component* atomic template exposes four interfaces. As discussed in Section 7.4, *working_state* provides the current working state of the individual component, while *num_components*, *num_failed_noisy*, and *num_failed_silent* are used to record, for components in the same group, the number of them that are currently working or failed.

The *streamcollector* template (lines 10–14) models the recording of a stream. It has no parameters, and its interfaces are *stream_out*, *num_components*, *num_failed_noisy*, and *num_failed_silent*. It should be noted that the *streamcollector* template is associated with the "s" (for stream) *prefix*. The index associated to instances of this template is related to the index of streams to be represented in the scenario. The *streamadapter* and *streamplayer* templates have a similar structure and they are not shown here. They are also associated with the "s" *prefix*.

Lines 18–52 depict the specification of some composition templates. The *repcomponent* template (lines 18–28) is a "replica" composition template for the *component* template model. This template specifies which interfaces should be connected together when composing multiple identical instances of the *component* template. This template covers the "Rep" nodes of Figure 7.3: "Rep1", "Rep2", "repo1", and "repo2". The template has one parameter, *num*, specifying the number of components to be replicated. The interfaces which are connected together are *num_components*, *num_failed_noisy*, and *num_failed_silent*, while *working_state* interfaces are not connected.

Lines 30–44 depict the specification of the *node_displayws*, corresponding to the composition of a display workstation with its "StreamAdapter" models, and with the models of the projectors that are under its control. This composition template covers the node "DisplayWS_with_Streams" of Figure 7.3. The template has one parameter, *streams*, describing which streams (in the form of

numerical indices) should be handled by the display workstation represented by the model. Submodels of this template are: one instance of the *component* template to represent the workstation ("ws"), a certain number of instances of the *component* template to represent the projectors ("proj"), and a certain number of *streamadapter* templates ("sa"). As shown in the listing, the multiplicity of the *streamadapter* templates is set based on the *streams* parameter. Three mergerules are defined: i) merge `working_state` in the *component* model with `component_state` in all the *streamadapter* models; ii) merge `stream_out` of the *streamadapter* models with the `stream_in` of *node_proj* models having the same indices (i.e., referring to the same stream), and iii) forward the `stream_in` of *streamadapter* models as interfaces of the composed model (they will be either connected with corresponding interfaces of the mixer, or forwarded up as interfaces of the whole stage).

The specification of the overall WO model corresponds to the *stageset* template (lines 46–52). As submodels it has a certain number of the *stage* template model. The *stage* template model, not shown here for the sake of brevity, corresponds to the top-level Join of Figure 7.3, i.e., the "Gateway_with_Streams" node in the top right part of the figure. Intuitively, each *stage* submodel has one interface for each stream in which the stage is involved. More in detail, for each stream that is acquired in the stage, the `stream_out` interface of the corresponding *streamadapter* model is forwarded as `outgoing_out`; similarly, for each stream that is received from another stage, the `stream_in` interface of the corresponding *streamadapter* model is forwarded as `incoming_in`.

The *mergebyname* specification in the *stageset* template model specifies that `outgoing_out` and `incoming_in` interfaces of *stage* models should be connected based on their indices having prefix "s". For example, if stage A has an interface whose instance name is "incoming_in_s3", and stage B has an interface whose instance name is "outgoing_out_s3" the two interfaces will be connected together.

Listing 8.1: *(Selected portions of the) TMDL "Library" specification for the World Opera system.*

```
1  atomic component {
2    body "Component.xml"
3    parameters {
4      failrate def 1.0E−4, spares def 0, fprobnoisy def 0,
5      sw_delay def 1, sw_fprob def 0.05
6    }
7    interfaces { num_components, num_failed_noisy,
8      num_failed_silent, working_state }
9  },
10 atomic streamcollector prefix "s" {
11   body "StreamCollector.xml"
12   interfaces { stream_out, num_components,
13     num_failed_noisy, num_failed_silent }
14 },
```

```
atomic streamadapter prefix "s" { ... },
atomic streamplayer prefix "s" { ... },

composition replica repcomponent {
  parameters { num def 1 }
  submodules {
    block c { component mult paramref { num } }
  }
  mergerules {
    mergeall num_components { "component.num_components" },
    mergeall num_failed_noisy { "component.num_failed_noisy" },
    mergeall num_failed_silent { "component.num_failed_silent" }
  }
},
...
composition node_displayws {
  parameters { streams def { 1 } }
  submodules {
    block ws { component mult 1 },
    block proj { node_proj },
    block sa { streamadapter mult paramref { "node_proj.streams" } }
  }
  mergerules {
    mergeall component_state {
      "component.working_state", "streamadapter.component_state" },
    mergebyname streams_out {
      "streamadapter.stream_out", "node_proj.stream_in" },
    forward streams_in { "streamadapter.stream_state_in" }
  }
},
...
composition stageset {
  submodules { block sg { stage } }
  mergerules {
    mergebyname inout prefixes "s" {
      "stage.incoming_in", "stage.outgoing_out" }
  }
}
```

8.5.2 *Specification of Scenarios*

Listing 8.2 shows a subset of the TMDL "Scenario" specification for a WO performance comprising three stages and five multimedia streams, listed in the following:

1. *a_orchestra* – audio of the orchestra;
2. *a_scene* – audio of actors;
3. *v_orchestra* – video of the orchestra;

4. *v_scene* – video of actors;

5. *v_director* – video of the orchestra director.

Streams 1,3, and 5 are captured in stage #1, while streams 2 and 4 are captured in stage #2. All the streams are reproduced in all the three stages. Stage #1, corresponds to the model depicted in Figure 7.3.

Listing 8.2 focuses on the specification of projectors of Stage #1, i.e., the highlighted part of Figure 7.3. Using the *component* template, a model class for a projector is created (lines 3–9), and values are specified for all its parameters, except *sw_fprob* and *spares*, for which the default value specified in the template is used.

Listing 8.2 also shows the definition of two different model classes based on the same template. In particular, classes *s1_proj_orchestra_director* and *s1_proj_scene* are created from the same *node_proj* model template; the two nodes "Projector_Orchestra_and_Director" and "Projector_Scene" in Figure 7.3 would be generated as instances of these two classes.

The specification of *s1_proj_orchestra_director* states that such composed class has an instance of the *projector* class as submodels, and that it handles streams 3 and 5. Submodels of kind "StreamAdapter" do not need to be specified: their multiplicity is derived from the *stream* parameter, in a similar way as for the *node_displayws* template (see Listing 8.1). Similarly, the specification of *s1_proj_scene* specifies that the corresponding projector should handle stream 4.

A class derived from the *node_displayws* is shown in the listing, specifying that the display workstation should process streams 3, 4, and 5. Also in this case, the "StreamPlayer" models do not need to be specified.

Listing 8.2: *(Selected portions of the) TMDL/Scenario specification for a WO performance composed of three stages and five streams.*

```
1  scenario { root wo_show
2    ...
3    class projector { usetemplate component
4      assignments {
5        "component.failrate" value 0.006,
6        "component.fprobnoisy" value 0.1,
7        "component.sw_delay" value 60.0,
8      }
9    },
10   class workstation { usetemplate component ... }
11   ...
12   class s1_proj_orchestra_director { usetemplate node_proj
13     assignments { "node_proj.streams" value { 3,5 } }
14     submodels { projector }
15   },
16   class s1_proj_scene { usetemplate node_proj
17     assignments { "node_proj.streams" value { 4 } }
18     submodels { projector }
```

```
19    },
20    ...
21    class s1_node_displayws { usetemplate node_displayws
22      assignments { "node_displayws.streams" value { 3,4,5 } }
23      submodels {
24        workstation, s1_proj_orchestra_director, s1_proj_scene
25      }
26    },
27    ...
28    class wo_show { usetemplate stageset ... }
29  }
```

One of the major advantages in using this approach, once the TMDL "Library" specification is established, is the ability to automatically obtain the model for different system scenarios by simply changing few lines of the TMDL "Scenario" specification. Listing 8.3 shows the modifications needed to perform the modification discussed in Section 7.5, i.e., introducing a new projector for stream 5, *v_director*. Using the TMDL approach, this modification only requires to add the specification of a new class based on the *node_proj* template (*s1_proj_director* class), having parameter *streams* set to {5}, and add it as a submodel of the class corresponding to the display workstation.

Listing 8.3: *Modified TMDL "Scenario" specification for adding a new projector dedicated to stream 5,* v_director, *to the model.*

```
1   class s1_proj_orchestra { usetemplate node_proj
2     assignments { "node_proj.streams" value { 3 } }
3     submodels { projector }
4   },
5   class s1_proj_director { usetemplate node_proj
6     assignments { "node_proj.streams" value { 5 } }
7     submodels { projector }
8   },
9   class s1_proj_scene { usetemplate node_proj
10    assignments { "node_proj.streams" value { 4 } }
11    submodels { projector }
12  },
13  ...
14  class s1_node_displayws { usetemplate node_displayws
15    assignments { "node_displayws.streams" value { 3,4,5 } }
16    submodels {
17      workstation, s1_proj_orchestra, s1_proj_director, s1_proj_scene
18    }
19  },
```

Let us now suppose that the architecture of Stage #1 changes, and that stream *v_scene* is now reproduced on two identical projectors. The few required modifications to the specification (with respect to Listing 8.2) are shown in Listing 8.4.

In this case we have only added the specification "replica repcomponent mult 2" to the *projector* submodel, meaning that it should be replicated twice, using the replica template *repcomponent*.

Listing 8.4: *Modified TMDL "Scenario" specification for a scenario where stream* v_scene *is reproduced on two identical projectors.*

```
class s1_proj_scene { usetemplate
node_proj assignments { "node_proj.streams" value { 4 } }
  submodels { projector replica repcomponent mult 2 }
},
```

Listing 8.5 considers the case in which the two projectors have instead different properties. In this case, two different classes *projector_a* and *projector_b* should be defined from the same template, having different parameters. Then, the two classes are set as submodels of the *s1_proj_scene* model class, which refers to stream *v_scene*.

Listing 8.5: *Modified TMDL "Scenario" specification for a scenario where stream* v_scene *is reproduced on two projectors having different properties.*

```
class projector_a { usetemplate
  assignments { "component.failrate" value 0.006, ... }
},
class projector_b { usetemplate component
  assignments { "component.failrate" value 0.001, ... }
},

class s1_proj_scene { usetemplate node_proj
  assignments { "node_proj.streams" value { 4 } }
  submodels { projector_a, projector_b }
},
```

Even these simple operations, if performed directly on the model, would require considerable effort for the modeler. Performing the same modification that is specified in Listing 8.5 would require the following steps: i) duplicating the atomic model for the "Projector", modifying the needed parameters, ii) adding an instance of the new "Projector" atomic model to the composed model of Figure 7.3, iii) properly connect the interfaces of the new model to the "Projector_Scene" Join. More complex modifications would require even more steps to be performed, in the worst case leading to modify the shared variables within all the Join nodes until the root of the overall model. In our approach, once the model library has been specified, such modifications are instead handled automatically by the model generation process.

8.6 APPLICATION TO THE HIDENETS SYSTEM

In order to verify the applicability of this approach to different kind of systems, we applied the TMDL specification approach to another library of template models, which were used in the past in the context of the HIDENETS project [22] to model a vehicular networks system in a highway scenario. The model, defined using the SANs formalism, is described in [23, 113], and it is not fully described here for the sake of brevity.

The use-case modeled in [23] evolves around a scene with an accident on a road, and a subsequent traffic jam. The considered network scenario is composed by a set of overlapping UMTS[2] mobile telecommunication cells covering a highway, and a set of mobile devices (embedded or inside cars and emergency vehicles) moving in the highway and requiring different network services, including "regular" services (e.g., voice calls, browsing), and "emergency" services (e.g., multimedia communication between the ambulance and the hospital).

The challenge in such use-case is being able to provide the emergency services in a congested environment (users in the traffic jam), while still providing an acceptable QoS to other users. For this purpose, the analysis focused on evaluating the impact of different parameters on a set of key reliability and performance metrics [23]. Also in this case, the TMDL approach would facilitate the evaluation of target metrics in different scenarios, e.g., different network services, different base stations layouts, different classes of users.

8.6.1 *Library Specification*

The performability model for the HIDENETS use case was based on 7 atomic SAN templates: *Phases*, *User*, *UserMobility*, *BaseStation*, *Service*, *ServiceManager*, *CellManager*. The scenario evaluated in [23] consisted of four base stations, two different kind of users (regular users and the ambulance), and five network services (3 regular services and 2 emergency services). The corresponding composed model, obtained from ~40 instances of the above template models, is depicted in Figure 8.4.

Selected portions of the the TMDL "Library" specification for this system are shown in Listing 8.6. Atomic templates are defined in lines 1–16; for each of them, an implementation of the model is referenced using the *body* attribute. The atomic template "basestation" (lines 1–10) has two parameters: *numservices*, which specifies how many services are supported by such basestation, and *load_max*, which specifies the maximum allowed load for that basestation. The *basestation* atomic template exposes several interfaces. Among them, two are shown in the listing: *LoadFactor* exposes the current load factor of the bases-

2 Universal Mobile Telecommunications System

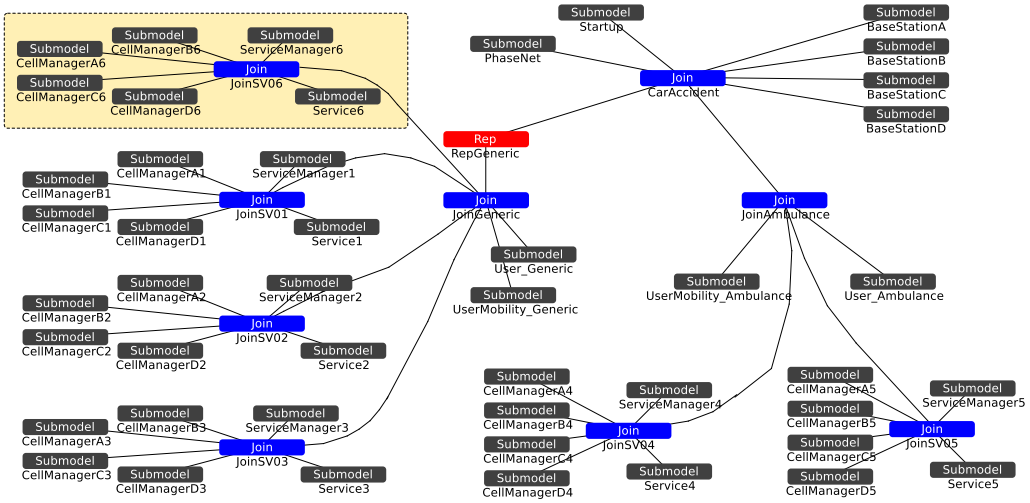


Figure 8.4: Composed SAN model for the HIDENETS scenario with 4 basestations and 5 services. Adding a 6th service requires adding (and properly connecting) the highlighted submodels.

tation; while *ChannelCount* is an array that, for each service, contains the number of channels allocated to that service on the basestation. Accordingly, the cardinality of the array is set to the *numservice* parameter.

The *cellmanager* template (lines 11–16) has the role of managing the delivery of a specific service to a specific basestation. Accordingly, it has a *load* parameter, which specifies the (average) load produced on the basestation by allocating one channel for such service.

Lines 17–34 depict the specification of a composition template, *servicenode*, which represents a service of the network. The template has two parameters: *load*, which specify the load produced by the allocation of one channel for that service, and *mycells*, which specifies on which basestations the service is implemented. The *servicenode* composition template composes one instance of a *service* component (line 23), one instance of a *servicemanager* component (line 24), and a certain number of *cellmanager* components (line 25). Since each *cellmanager* model manages the delivery of the service on a given basestation, the multiplicity of the *cellmanager* submodules is set to match the value of the *mycells* parameter. Line 27 shows an example usage of the *bind* construct, which allows parameters of submodules to be "linked" to parameters of the containing template. In this case the *load* parameter of the *cellmanager* submodules is bound to the *load* parameter of the *servicenode* composition template.

Lines 28–33 define the rules to follow to assemble the *servicenode* template based on its submodules. Line 29 specifies that the *ServChannels* interface of the *servicemanager* should be connected together with the *ServChannels* interface of all the *cellmanager* submodules. Interfaces *ChannelCount* and *LoadFactor* are forwarded from the *cellmanager* submodules.

Listing 8.6: *(Selected portions of the) TMDL "Library" specification for the HIDENETS system.*

```
1  atomic basestation { body "BaseStation.xml"
2    parameters {
3      numservices def 1,
4      load_max def 0.8
5    }
6    interfaces {
7      LoadFactor, ...
8      array ChannelCount mult paramref numservices
9    }
10 }, ...
11 atomic cellmanager { body "CellManager.xml"
12   parameters {
13     load def 0.01
14   }
15   interfaces { ... }
16 },
17 composition servicenode {
18   parameters {
19     my_cells def { 1 },
20     load def 0
21   }
22   submodules {
23     block s { service mult 1 },
24     block sm { servicemanager mult 1 },
25     block cm { cellmanager mult paramref mycells }
26   }
27   bindings { "cellmanager.load" to load }
28   mergerules {
29     mergeall ServChannels { "cellmanager.ServChannels", "servicemanager.ServChannels" }
30     forward ChannelCount { "cellmanager.ChannelCount" },
31     forward LoadFactor { "cellmanager.LoadFactor" }
32     ...
33   }
34 }
35 ...
```

8.6.2 *Specification of Scenarios*

Listing 8.7 shows a subset of the TMDL "Scenario" specification for the scenario modeled in [23, 113]. In particular, the definition of the different services and the different class of users is shown.

Using the *servicenode* template, five model classes are created, each one representing one of the five services of the scenario: *voice*, *browsing*, *filetransfer*, *emergency_data*, *emergency_video* (lines 4–13). Each of them is supported by all the four basestations (*mycells* parameter), but has different values for the load that it generates (*load*).

Listing 8.7 also shows the definition of two different class of users, both based on the *template_user* template (lines 14–23). The "normal" user is represented by the *normaluser* composed component; the scenario specifies that three components derived from the *servicenode* template should be used as subcomponents: *voice*, *browsing* and *filetransfer*. The "ambulance" user is represented by the *ambulance* component, which uses the same template, but uses *emergency_data*, and the *emergency_video* as submodels, since it has access to different network services provided by the infrastructure.

Listing 8.7: *(Selected portions of the) TMDL/Scenario specification for the scenario evaluated in [23, 113].*

```
1  scenario {
2    root caraccident
3    ...
4      class voice { usetemplate servicenode
5        assignments {
6          "servicenode.my_cells" value { 1,2,3,4 },
7          "servicenode.load" value 0.01357
8        }
9      },
10     class browsing { usetemplate servicenode ... }
11     class filetransfer { usetemplate servicenode ... }
12     class emergency_data { usetemplate servicenode ...
13     class emergency_video { usetemplate servicenode ... }
14     class normaluser { usetemplate user
15       submodels {
16         voice mult { 1 }, browsing mult { 2 }, filetransfer mult { 3 }
17       }
18     }, ...
19     class ambulance { usetemplate user
20       submodels {
21         emergency_data mult { 4 }, emergency_video mult { 5 }
22       }
23     },
24 } }
```

Listing 8.8 shows the modifications needed in order to add an "Instant Messaging" service for normal users. In order to add the new service, it is sufficient to create a new *messaging* model class (from the *servicenode* template) and add it as submodel of the *normaluser* class.

Listing 8.8: *Modifications required to the TMDL "Scenario" specification in order to add a 6th service "Instant Messaging" to the scenario.*

```
1      class messaging { usetemplate servicenode
2      assignments {
3          "servicejoin.my_cells" value { 1,2,3,4 },
4          "servicejoin.load" value 0.001357
5        }
6      }, ...
```

```
7     class normaluser { usetemplate template_user
8       submodels {
9         voice mult { 1 }, browsing mult { 2 }, filetransfer mult { 3 }, messaging mult { 6 }
10      }
11    },
```

Performing the same modification by hand would require the modeler to add 6 atomic models, and properly connect them to each other, and to the already existing portion of the model, possibly also requiring modifications to existing connections. With this approach, connections are specified in the library, and are performed automatically by the model generation and composition algorithm. Applying the TMDL approach appears therefore to be useful in very different kind of systems.

8.7 TOWARDS A SYSTEM OF SYSTEMS APPROACH

We conclude this chapter with an outlook on possible applications of the proposed approach, with a focus on exploiting its synergies with the other contributions of this dissertation.

While the approach introduced in this chapter can be useful to support the modeling of large and dynamic systems under different configurations, so far we have still assumed *static* template models libraries, created by hand a-priori, based on the knowledge of the system that is under the analysis.

A further shift in system complexity has been however recently introduced by the System of Systems (SoS) paradigm, in which etherogeneous existing computer systems, called Constituent Systems (CS) are integrated in order to provide synergistic services and more efficient economic processes. This integration is occurring in many areas denoted by different names, including web services, sensor networks, critical infrastructures; the core issue in all these systems is the same, i.e., the integration of autonomous systems to solve a given problem.

One of the peculiarities of such systems consists in their dynamicity and etherogeneity: over the years, new functionalities are added to these systems, existing functionalities have to be modified in order to meet the demands of an evolving society, and new stand-alone systems are being interconnected and integrated. Model-based dependability evaluation in such a dynamic and evolving context is an unprecedented challenge. Moreover, system evaluation should be performed continuously, when new components are introduced into (or removed from) the environment.

While neither the DEP-UML nor the TMDL approaches alone would be completely sufficient in such context, we believe that their combined application can provide a useful contribution towards a System of Systems (SoS) evaluation framework. The TMDL approach provides a contribution in managing

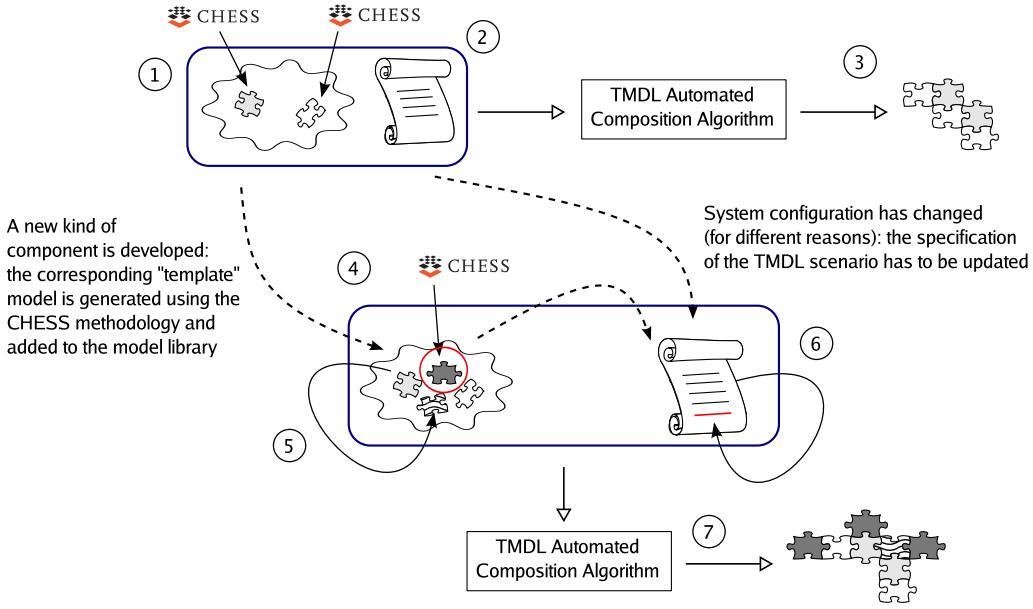


Figure 8.5: The application of approaches presented in this thesis for a System of Systems evaluation approach.

dynamicity in system configurations, since evaluation models for different configurations could be automatically assembled based on libraries of template models. On the other hand, the DEP-UML approach can be employed to actually populate such model libraries, obtaining a stochastic model for dependability analysis as an artifact of the system development process.

This view is further illustrated in Figure 8.5, and described in the following. We suppose that two new elements of the SoS architecture are developed using the CHESS methodology; their evaluation models are automatically derived and placed in the model library (#1). The system configuration is then described by means of the TMDL language (#2), and an overall SoS analysis model is generated (#3). After some time, a new component is developed, and can potentially become part of the SoS; its evaluation model is therefore generated and stored in the model-library (#4). The introduction of a new element in the model library may require to add other "templates", in order to compose the newly introduced model with existing ones (#5). After some time, the system configuration changes, possibly including the newly developed component. The corresponding TMDL specification is then updated to reflect the change, possibly referring to the newly added template (#6). The new SoS evaluation model is then automatically assembled (#7) and evaluated. The above steps are of course repeated during the entire life of the SoS, for a continuous evaluation of its the target metrics.

CONCLUSION AND OUTLOOK

Models plays a primary role in dependability and performability assessment of modern computing systems. Modeling, over other evaluation techniques, has the key advantage of not exercising a real instance of the system, which may be costly, dangerous, or simply unfeasible (e.g., because the system is still under design). As a fault-forecasting technique, model-based evaluation can be used to estimate the degree to which a given design provides the required dependability attributes, thus allowing system architects to understand and learn about specific aspects of the system, to detect possible design weak points or bottlenecks, to perform early validation of dependability requirements, or to suggest solutions for future releases or modifications. More in general, models are employed in the evaluation of the QoS provided by the system, under the form of dependability, performance, or performability metrics.

Modern computing systems have become very different from what they used to be in the past: their scale is growing, they are becoming massively distributed, interconnected, and evolving. Moreover, a shift towards the use of off-the-shelf components is becoming evident in several domains. Such increase in complexity makes model-based assessment a difficult and time-consuming task. Moreover, models need to be maintained and updated throughout the entire system design process, reflecting architectural changes, variations in system or component configurations, and possibly integrating additional information produced by other evaluation approaches.

In the last years, the development of systems has increasingly adopted the CBD and MDE philosophies as a way to reduce the complexity in system design. CBD refers to the established practice of building a system out of reusable "black-box" components, while MDE refers to the systematic use of models as primary artefacts throughout the engineering lifecycle. Languages like UML, BPEL, AADL, etc., allow not only a reasonable unambiguous specification of design but also serve as the input for subsequent development steps like code generation, formal verification, and testing.

In the last decade, MDE approaches have been also extensively employed for the analysis of extra-functional system properties. To this purpose, language extensions were introduced and utilized to capture the required extra-functional concepts. Deriving dependability analysis models from the engineering models that are created during the development process has the advantage that — besides the required model extensions — there is no need to learn and use specific formalisms, and modelling efforts can therefore be saved. Although a lot of work has been (and is being) developed on MDE techniques for dependability analysis, most of the approaches have been defined as extensions

to a "general" system development process, often leaving the actual process unspecified. Similarly, supporting tools are typically detached from the design environment, and assume to receive as input a model satisfying certain constraints. While in principle such approach allows not to be bound to specific development methodologies, in practice it introduces a gap between the design of the functional system model, its enrichment with dependability information, and the subsequent analysis. This is one of the effects of UML complexity and generality, which typically lead system designers to adopt only a subset of the language; even worse, in different contexts different subsets are typically adopted. As a result, once the functional model of the system has been designed, there is no guarantee that a given UML-based analysis technique can be applied.

The work in this thesis defined a MDE approach for quantitative dependability analysis by taking into account a concrete system development process, and the associated component model. The reference methodology is the one defined within the CHESS project, which focuses on the development of critical real-time embedded systems, spanning the railway, telecommunications, space, and automotive domains. The CHESS methodology is realized by means of the CHESS ML modeling language, a UML profile which reuses portions of the UML, MARTE, and SysML standards, and adds further elements to support its component model.

Based on the CHESS methodology and language, a set of extensions to support the specification and analysis of dependability properties have been defined, with a focus on quantitative dependability analysis. The process leading to the definition of such extensions included several phases: i) collection of language requirements, based on requirements of industrial partners, common practices, and other sources; ii) definition of a conceptual model of required concepts; iii) analysis of existing languages; iv) actual definition of UML extensions. The resulting language, DEP-UML, supports the specification of dependability properties of components, multiple failure modes, propagation, maintenance activities, fault-tolerant structures, and promotes an incremental modeling approach, in which the system is designed in subsequent refinement steps.

After defining DEP-UML, the thesis designed a model-transformation algorithm for the automated generation of Stochastic Petri Nets (SPNs) from models enriched with DEP-UML properties. The algorithm adopts an intermediate model, and actually defines two model-transformations: the first generates an intermediate representation from DEP-UML models; the second generates a SPNs model from the obtained intermediate representation. The proposed approach has been concretely realized as a plugin for the Eclipse platform, and it has been integrated with the overall CHESS framework, which also includes a diagram editor and constraints checking mechanisms. From a practical point of view, having the analysis tool capable of direct interactions with the system design environment: i) ensures and enforces the correctness of the

model provided as input to the transformation algorithm, and ii) enables back-annotation of obtained results in the design model. While the tool is integrated in the CHESS framework, it has been designed with the aim to be reused in other contexts; as such, its architecture actually relies on multiple intermediate model-transformation steps.

The introduced approach (and tool) is then applied to two extensive case studies. The first one takes into account a multimedia processing workstation with high availability requirements, and aims at evaluating the effectiveness of a software rejuvenation policy. The functional model of the system has been designed according to the CHESS methodology, and then evaluated using the proposed approach. A second case study of a fire detection system demonstrates the application of DEP-UML to the analysis of hardware architectures. This case study took into account two hierarchical levels of the system, and highlighted how the methodology can be applied in incremental refinement steps, with limited modifications on the already specified architecture.

The last part of the thesis builds on the key observation that, while for embedded systems it is often possible to follow and control the whole design and development process, the same does not hold for other classes of systems and infrastructures. In particular, large-scale complex systems don't fit well in the paradigm proposed by the CHESS project, and alternative approaches are therefore needed.

The thesis then analyzed a common technique used in the domain of Stochastic Petri Nets for modeling large-scale complex systems; such technique, which has some similarities with object-oriented approaches, relies on the definition of "template" models, which are then instantiated multiple times with different parameters and composed by place superposition. The main gaps that were identified in its application reside in i) the definition of patterns for assembling instances of template models, which are typically defined only informally, and ii) the actual assembly of the overall model, which requires complex and time-consuming tasks. The modification of the system model is therefore cumbersome, and changes in system configurations, or different system scenarios are difficult to take into account.

To address this problem, an MDE workflow for the automated composition of complex performability models has then been defined. The workflow is based on the Template Models Description Language (TMDL), a domain-specific language that allows libraries of template models to be specified (including composition patterns), and then used. A key point of the approach is that the model generation algorithm is the same for any TMDL library, and it is defined and implemented only once, since semantic information on the transformation is moved to the model library. After having introduced such workflow, the thesis detailed the TMDL language, and described a prototype realization of the workflow within the Eclipse platform. Te use of the TMDL language for the specification of template models libraries and is then illustrated through its

application to two different examples. The application of such approach, still in a prototypal shape, could improve the modeling of large-scale, complex, and dynamic systems. Actually, a modification in system configuration would only require small changes to the TMDL specification, and the triggering of the model-generation algorithm.

The thesis than concluded with an outlook on the possible combined application of the proposed approaches, with a focus on exploiting their synergies. In particular, a further shift in system complexity has been introduced by the System of Systems (SoS) paradigm, in which etherogeneous existing computer systems, called Constituent Systems (CS) are integrated in order to provide synergistic services and more efficient economic processes. This integration is occurring in many areas denoted by different names, including web services, sensor networks, critical infrastructures; the core issue in all these systems is the same, i.e., the integration of autonomous systems to solve a given problem.

While neither the DEP-UML nor the TMDL approaches alone would be completely sufficient in such context, we believe that their combined application can provide a useful contribution towards a System of Systems (SoS) evaluation framework. The TMDL approach (introduced in Chapters 7–8) provides a contribution in managing dynamicity in system configurations, since evaluation models for different configurations could be automatically assembled based on libraries of template models. On the other hand, the DEP-UML approach (introduced in Chapters 3–6) can be employed to actually populate such model libraries, since stochastic models for dependability analysis are obtained as an artifact of the system development process.

The field for the application of MDE techniques for dependability is therefore growing; future work should be aimed at supporting model-based evaluation at different system scales, and at integrating models and results into a System of Systems evaluation framework.

BIBLIOGRAPHY

[1] V. Adve, R. Bagrodia, J. Browne, E. Deelman, A. Dube, E. Houstis, J. Rice, R. Sakellariou, D. Sundaram-Stukel, P. Teller, and M. Vernon. "PO-EMS: end-to-end performance design of large parallel adaptive computational systems." In: *IEEE Transactions on Software Engineering* 26.11 (2000), pp. 1027–1048 — *cited on page* 15.

[2] E. Amparore and S. Donatelli. "Model checking CSL^{TA} with Deterministic and Stochastic Petri Nets." In: *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN'10 (Chicago, Illinois, USA, June 28–July 1, 2010). IEEE, 2010, pp. 605–614 — *cited on page* 14.

[3] T. Araki and T. Kasami. "Some decision problems related to the reachability problem for Petri nets." In: *Theoretical Computer Science* 3.1 (1977), pp. 85–104 — *cited on page* 85.

[4] U. Aßmann. *Invasive Software Composition*. Springer, 2003 — *cited on page* 150.

[5] *Atlas Transformation Language (ATL)*. URL: `http://www.eclipse.org/atl/` (visited on 12/28/2013) — *cited on pages* 105, 107, 156.

[6] A. Avižienis, J.-C. Laprie, B. Randel, and C. Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing." In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33 — *cited on pages* iii, 1–4, 38, 41–42, 59.

[7] S. Baarir, M. Beccuti, D. Cerotti, M. De Pierro, S. Donatelli, and G. Franceschinis. "The GreatSPN Tool: Recent Enhancements." In: *ACM SIGMETRICS Performance Evaluation Review* 36.4 (Mar. 2009), pp. 4–9 — *cited on page* 14.

[8] C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. "Performance evaluation and model checking join forces." In: *Communications of the ACM* 53.9 (Sept. 2010), pp. 76–85 — *cited on pages* 12, 14.

[9] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. "Model-based performance prediction in software development: a survey." In: *IEEE Transactions on Software Engineering* 30.5 (2004), pp. 295–310 — *cited on page* 24.

[10] F. Bause, P. Buchholz, and P. Kemper. "A Toolbox for Functional and Quantitative Analysis of DEDS." In: *Computer Performance Evaluation*. Ed. by R. Puigjaner, N. Savino, and B. Serra. Vol. 1469. LNCS. Springer, 1998, pp. 356–359 — *cited on page* 15.

[11] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, and W. Stewart. "The PEPS Software Tool." In: *Computer Performance Evaluation. Modelling Techniques and Tools*. Ed. by P. Kemper and W. H. Sanders. Vol. 2794. LNCS. Springer, 2003, pp. 98–115 — *cited on page* 15.

[12] C. Beounes, M. Aguera, J. Arlat, S. Bachmann, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J. Moreira de Souza, D. Powell, and P. Spiesser. "SURF-2: A program for dependability evaluation of complex hardware and software systems." In: *23rd International Symposium on Fault-Tolerant Computing, Digest of Papers*. FTCS-23 (Toulouse, France, June 22–24, 1993). IEEE, 1993, pp. 668–673 — *cited on page* 14.

[13] S. Bernardi and S. Donatelli. "Stochastic Petri nets and inheritance for dependability modelling." In: *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing*. PRDC'04 (Papeete, Tahiti, French Polynesia, Mar. 3–5, 2004). IEEE, 2004, pp. 363–372 — *cited on page* 142.

[14] S. Bernardi, S. Donatelli, and J. Merseguer. "From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models." In: *Proceedings of the 3rd International Workshop on Software and Performance*. WOSP'02 (Rome, Italy). ACM, 2002, pp. 35–45 — *cited on page* 25.

[15] S. Bernardi, J. Merseguer, and D. C. Petriu. "Dependability modeling and analysis of software systems specified with UML." In: *ACM Computing Surveys* 45.1 (Nov. 2012), 2:1–2:48 — *cited on page* 24.

[16] S. Bernardi, J. Merseguer, and D. C. Petriu. "A dependability profile within MARTE." In: *Software and Systems Modeling* 10.3 (2011), pp. 313–336 — *cited on pages* 25, 49.

[17] S. Bernardi and D. C. Petriu. "Comparing two UML Profiles for Non-functional Requirement Annotations: the SPT and QoS profiles." In: *UML Modeling Languages and Applications. UML 2004 Satellite Activities*. UML 2004 Satellite Activities (Lisbon, Portugal, Oct. 10–15, 2004). Vol. 3297. LNCS. Springer, 2004 — *cited on page* 47.

[18] A. Bobbio and K. Trivedi. "An Aggregation Technique for the Transient Analysis of Stiff Markov Chains." In: *IEEE Transactions on Computers* C-35.9 (Sept. 1986), pp. 803–814 — *cited on page* 14.

[19] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. John Wiley & Sons, 2006 — *cited on pages* 8, 16.

[20] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and I. Mura. "Dependability modeling and evaluation of multiple-phased systems using DEEM." In: *IEEE Transactions on Reliability* 53.4 (Dec. 2004), pp. 509–522 — *cited on pages* 14, 102.

[21] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. "Dependability Analysis in the Early Phases of UML Based System Design." In: *Journal of Computer Systems Science and Engineering* 16.5 (2001), pp. 265–275 — *cited on pages* 25, 37, 50, 55.

[22] A. Bondavalli, O. Hamouda, M. Kaâniche, P. Lollini, I. Majzik, and H.-P. Schwefel. "The HIDENETS Holistic Approach for the Analysis of Large Critical Mobile Systems." In: *IEEE Transactions on Mobile Computing* 10.6 (June 2011), pp. 783–796 — *cited on pages* 25, 111, 139, 163.

[23] A. Bondavalli, P. Lollini, and L. Montecchi. "QoS Perceived by Users of Ubiquitous UMTS: Compositional Models and Thorough Analysis." In: *Journal of Software* 4.7 (Sept. 2009) — *cited on pages* 142, 163, 165–166, 203.

[24] A. Bondavalli, I. Majzik, and I. Mura. "Automatic Dependability Analysis for Supporting Design Decisions in UML." In: *Proceedings of the 4th IEEE High Assurance System Engineering Symposium)*. HASE'99 (Washington D. C., USA, Nov. 17–19, 1999). IEEE, 1999, pp. 64–71 — *cited on page* 50.

[25] A. Bondavalli, P. Lollini, I. Majzik, and L. Montecchi. "Modelling and Model-Based Assessment." In: *Resilience Assessment and Evaluation of Computing Systems*. Ed. by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel. Springer, July 2012, pp. 153–165 — *cited on page* 15.

[26] A. Bovenzi, S. Russo, F. Brancati, and A. Bondavalli. "Towards identifying OS-level anomalies to detect application software failures." In: *Proceedings of the 1st IEEE International Workshop on Measurements and Networking*. M&N'11 (Anacapri, Italy). 2011, pp. 71–76 — *cited on page* 111.

[27] A. W. Brown and K. C. Wallnau. "The current state of CBSE." In: *IEEE Software* 15.5 (September/October 1998), pp. 37–46 — *cited on pages* 17–18.

[28] R. M. L. R. Carmo, L. R. Carvalho, E. Souza e Silva, M. C. Diniz, and R. R. Muntz. "TANGRAM-II: A performability modeling environment tool." In: *Computer Performance Evaluation Modelling Techniques and Tools*. Ed. by R. Marie, B. Plateau, M. Calzarossa, and G. Rubino. Vol. 1245. LNCS. Springer, 1997, pp. 6–18 — *cited on page* 15.

[29] G. Casale, R. R. Muntz, and G. Serazzi. "Special issue on tools for computer performance modeling and reliability analysis." In: *ACM SIGMETRICS Performance Evaluation Review* 36.4 (Mar. 2009), pp. 2–3 — *cited on page* 14.

[30] A. Ceccarelli, J. Grønbæk, L. Montecchi, H.-P. Schwefel, and A. Bondavalli. "Towards a Framework for Self-Adaptive Reliable Network Services in Highly-Uncertain Environments." In: *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. ISORCW'10 (Carmona, Spain, May 4–7, 2010). IEEE, 2010, pp. 184–193 — *cited on page* 140.

[31] CENELEC EN 50126:1999-09. *Railway applications – The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*. 1999 — *cited on pages* 43, 125.

[32] R. Chapman. "Correctness by Construction: A Manifesto for High Integrity Software." In: *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software*. SCS'05 (Sydney, Australia, Aug. 19–20, 2005). Australian Computer Society, Inc., 2006, pp. 43–46 — *cited on page* 31.

[33] M. Chaudron, I. Crnkovic, and H. van Vliet. "Component-based software engineering." In: *Software Engineering: Principles and Practice*. Ed. by H. van Vliet. Wiley, 2008. Chap. 18 — *cited on page* 19.

[34] D. Chen, S. Dharmaraja, D. Chen, L. Li, K. Trivedi, R. Some, and A. Nikora. "Reliability and availability analysis for the JPL Remote Exploration and Experimentation System." In: *Proceedings of the International Conference on Dependable Systems and Networks*. DSN'02 (Bethesda, MD, USA, June 23–26, 2002). IEEE, 2002, pp. 337–342 — *cited on page* 13.

[35] *CHESS: "Composition with guarantees for High-integrity Embedded Software components aSsembly"*. ARTEMIS-2008-1-100022. URL: `http://www.chess-project.org/` — *cited on pages* iv, 29, 36, 97.

[36] CHESS Project. *Analysis and Evaluation Solutions*. Deliverable 2.2. Dec. 2010 — *cited on pages* 31–33, 52, 102.

[37] CHESS Project. *CHESS Modelling Language and Editor*. Deliverable 2.1. Mar. 2010 — *cited on page* 35.

[38] CHESS Project. *CHESS Toolset User Guide*. Version 3.2. URL: `http://www.math.unipd.it/~azovi/CHESS/CHESS_3.2/` (visited on 12/28/2013) — *cited on pages* 33, 36, 116.

[39] CHESS Project. *Dependability and security properties and analysis methods*. Deliverable 3.1. Mar. 2010 — *cited on page* 37.

[40] CHESS Project. *Multi-concern Component Methodology (MCM) and Toolset*. Deliverable 2.3.2. Jan. 2012 — *cited on page* 31.

[41] CHESS Project. *State-Based extra functional properties and State-Based Analysis*. Video Demonstrations, Set 2. URL: `http://www.chess-project.org/page/videos-1` (visited on 12/28/2013) — *cited on pages* 97, 111.

[42] CHESS Project. *Transformations and analysis support to dependability*. Deliverable 3.2.2. Dec. 2011 — *cited on pages* 70, 111.

[43] S. Chiaradonna, P. Lollini, and F. Di Giandomenico. "On a Modeling Framework for the Analysis of Interdependencies in Electric Power Systems." In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN'07 (Edinburgh, UK, June 25–28, 2007). IEEE, 2007, pp. 185–195 — *cited on page* 141.

[44] G. Ciardo, R. German, and C. Lindemann. "A characterization of the stochastic process underlying a stochastic Petri net." In: *IEEE Transactions on Software Engineering* 20.7 (1994), pp. 506–515 — *cited on pages* 11–12, 69, 84.

[45] G. Ciardo and A. Miner. "SMART: simulation and Markovian analyzer for reliability and timing." In: *Proceedings of IEEE International Computer Performance and Dependability Symposium*. IPDS'96 (Urbana-Champaign, IL, USA, Sept. 4–6, 1996). IEEE, 1996, p. 60 — *cited on page* 15.

[46] G. Ciardo. "Petri nets with marking-dependent arc cardinality: Properties and analysis." In: *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*. ICATPN'94 (Zaragoza, Spain, June 20–24, 1994). Ed. by R. Valette. Vol. 815. LNCS. Springer, 1994, pp. 179–198 — *cited on page* 85.

[47] G. Ciardo, A. Blakemore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi. "Automated Generation and Analysis of Markov Reward Models Using Stochastic Reward Nets." In: *Linear Algebra, Markov Chains, and Queueing Models*. Ed. by C. D. Meyer and R. J. Plemmons. Vol. 48. The IMA Volumes in Mathematics and its Applications. Springer, 1993, pp. 145–191 — *cited on pages* 12, 84.

[48] G. Ciardo and A. S. Miner. "Efficient reachability set generation and storage using decision diagrams." In: *Proceedings of the 20th International Conference on Applications and Theory of Petri Nets*. ICATPN'99 (Williamsburg, Virginia, USA, June 21–25, 1999). Springer, 1999, pp. 6–25 — *cited on page* 13.

[49] A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, T. Vardanega, and A. Zovi. "CHESS: a Model-Driven Engineering Tool Environment for Aiding the Development of Complex Industrial Systems." In: *Proceedings of the 27th International Conference on Automated Software Engineering*. ASE'12 (Essen, Germany, Sept. 3–7, 2012). ACM, Sept. 2012 — *cited on page* 52.

[50] F. Ciccozzi, A. Cicchetti, and M. Sjodin. "Towards a Round-Trip Support for Model-Driven Engineering of Embedded Systems." In: *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. SEAA'11 (Oulu, Finland). IEEE, 2011, pp. 200–208 — *cited on page* 30.

[51] A. Clark, S. Gilmore, J. Hillston, and M. Tribastone. "Stochastic Process Algebras." In: *Formal Methods for Performance Evaluation*. Ed. by M. Bernardo and J. Hillston. Vol. 4486. LNCS. Springer, 2007, pp. 132–179 — *cited on page* 12.

[52] P. C. Clements. "From Subroutines to Subsystems: Component-Based Software Development." In: *The American Programmer* 8.11 (1995) — *cited on pages* 17–18.

[53] V. Cortellessa and A. Pompei. "Towards a UML profile for QoS: a contribution in the reliability domain." In: *SIGSOFT Software Engineering Notes* 29.1 (2004), pp. 197–206 — *cited on page* 25.

[54] V. Cortellessa, H. Singh, and B. Cukic. "Early reliability assessment of UML based software models." In: *Proceedings of the 3rd International Workshop on Software and Performance*. WOSP'02 (Rome, Italy). New York, NY, USA: ACM, 2002, pp. 302–309 — *cited on page* 24.

[55] T. Courtney, S. Gaonkar, K. Keefe, E. W. D. Rozier, and W. H. Sanders. "Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models." In: *39th IEEE/IFIP International Conference on Dependable Systems Networks*. DSN'09 (Estoril, Portugal, June 29–July 2, 2009). IEEE, 2009, pp. 353–358 — *cited on pages* 15, 142, 147.

[56] I. Crnkovic, M. Chaudron, and S. Larsson. "Component-Based Development Process and Component Lifecycle." In: *Proceedings of the International Conference on Software Engineering Advances*. ICSEA06 (Tahiti, French Polynesia, Oct. 29–Nov. 3, 2006). IEEE, 2006, pp. 44–44 — *cited on page* 19.

[57] P. Cuenot, P. Frey, R. Johansson, H. Lönn, Y. Papadopoulos, M.-O. Reiser, A. Sandberg, D. Servat, R. Tavakoli Kolagari, M. Törngren, and M. Weber. "The EAST-ADL Architecture Description Language for Automotive Embedded Software." In: *Model-Based Engineering of Embedded Real-Time Systems*. Ed. by H. Giese, G. Karsai, E. Lee, B. Rumpe, and B. Schätz. Vol. 6100. LNCS. Springer, 2011, pp. 297–307 — *cited on pages* 48, 58.

[58] K. Czarnecki and S. Helsen. "Classification of Model Transformation Approaches." In: *Proceedings of the 18th Annual SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'03 (Oct. 26–30, 2003). Anaheim, CA, USA: ACM, 2003 — *cited on pages* iii, 24.

[59] A. Daidone, F. D. Giandomenico, A. Bondavalli, and S. Chiaradonna. "Hidden Markov Models as a Support for Diagnosis: Formalization of the Problem and Synthesis of the Solution." In: *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*. SRDS'06 (Leeds, UK, Oct. 2–4, 2006). 2006, pp. 245–256 — *cited on page* 6.

[60] M. Dal Cin, G. Huszerl, and K. Kosmidis. "Evaluation of Safety-Critical Systems based on Guarded Statecharts." In: *Proceedings of the Fourth IEEE International Symposium on High Assurance Systems Engineering*. HASE'99 (Washington D. C., USA, Nov. 17–19, 1999). IEEE, 1999 — *cited on page* 25.

[61] A. D'Ambrogio, G. Iazeolla, and R. Mirandola. "A method for the prediction of software reliability." In: *Proceedings of the 6th IASTED Software Engineering and Applications Conference*. SEA'02 (Cambridge, MA, USA, Nov. 4–6, 2002). 2002 — *cited on page* 24.

[62] S. Datla and N. Gidijala. "Parallelizing Motion JPEG 2000 with CUDA." In: *Proceedings of the 2nd International Conference on Computer and Electrical Engineering*. ICCEE'09 (Dubai, Dec. 28–30, 2009). Vol. 1. 2009, pp. 630–634 — *cited on page* 112.

[63] D. D. Deavours and W. H. Sanders. ""On-the-fly" solution techniques for stochastic Petri nets and extensions." In: *IEEE Transactions on Software Engineering* 24.10 (Oct. 1998), pp. 889–902 — *cited on page* 13.

[64] D. D. Deavours and W. H. Sanders. "An efficient disk-based tool for solving large Markov models." In: *Performance Evaluation* 33.1 (1998), pp. 67–84 — *cited on page* 13.

[65] E. W. Dijkstra. "On the role of scientific thought." In: *Selected Writings on Computing: A Personal Perspective*. Ed. by E. W. Dijkstra. Springer, 1982, pp. 60–66 — *cited on pages* 17, 32, 140.

[66] L. Dingping, Z. Kaitao, and Y. Qiqi. "Application of Data Stream Outlier Mining Techniques in Steam Generator Safety Early Warning System of Nuclear Power Plant." In: *Proceedings of the 5th International Conference on Measuring Technology and Mechatronics Automation*. ICMTMA'13 (Hong Kong, Jan. 16–17, 2013). IEEE, 2013, pp. 287–290 — *cited on page* 111.

[67] S. Distefano and A. Puliafito. "Dependability Evaluation with Dynamic Reliability Block Diagrams and Dynamic Fault Trees." In: *IEEE Transactions on Dependable and Secure Computing* 99.2 (2008) — *cited on page* 7.

[68] S. Donatelli, S. Haddad, and J. Sproston. "Model Checking Timed and Stochastic Properties with CSL^{TA}." In: *IEEE Transactions on Software Engineering* 35.2 (2009), pp. 224–240 — *cited on page* 14.

[69] S. Donatelli. "Superposed Generalized Stochastic Petri Nets: Definition and efficient solution." In: *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*. ICATPN'94 (Zaragoza, Spain, June 20–24, 1994). Ed. by R. Valette. Vol. 815. LNCS. Springer, 1994, pp. 258–277 — *cited on page* 140.

[70] *Eclipse Modeling Framework (EMF)*. URL: `http://www.eclipse.org/modeling/emf/` (visited on 12/28/2013) — *cited on pages* 97, 156.

[71] P. Feiler and A. Rugina. *Dependability Modeling with the Architecture Analysis & Design Language (AADL)*. Technical Report CMU/SEI-2007-TN-043. Software Engineering Institute, Carnegie Mellon, July 2007 — *cited on page* 49.

[72] R. Filippini and A. Bondavalli. "Modeling and Analysis of a Scheduled Maintenance System: a DSPN Approach." In: *The Computer Journal, BCS* 47.6 (2004), pp. 634–650 — *cited on page* 50.

[73] G. Franceschinis, M. Gribaudo, M. Iacono, N. Mazzocca, and V. Vittorini. "DrawNET++: Model Objects to Support Performance Analysis and Simulation of Systems." In: *In Proceedings of the 12th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*. TOOLS'02 (London, UK, Apr. 14–17, 2002). Ed. by T. Field, P. Harrison, J. Bradley, and U. Harder. Vol. 2324. LNCS. Springer, 2002, pp. 55–60 — *cited on page* 15.

[74] R. Fricks, C. Hirel, S. Wells, and K. Trivedi. "The development of an integrated modeling environment." In: *Proceedings of the World Congress on Systems Simulation*. WCSS'97 (Singapore, Sept. 1997). 1997, pp. 471–476 — *cited on page* 15.

[75] B. Gallina, M. Javed, F. U. Muram, and S. Punnekkat. "A Model-Driven Dependability Analysis Method for Component-Based Architectures." In: *Proceedings of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications*. SEAA'12 (Çeşme, Turkey, Sept. 5–8, 2012). IEEE, 2012, pp. 233–240 — *cited on page* 52.

[76] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995 — *cited on page* 17.

[77] S. Gaonkar, K. Keefe, R. Lamprecht, E. Rozier, P. Kemper, and W. H. Sanders. "Performance and dependability modeling with Möbius." In: *ACM SIGMETRICS Performance Evaluation Review* 36.4 (Mar. 2009), pp. 16–21 — *cited on page* 101.

[78] L. Gönczy, S. Chiaradonna, F. Di Giandomenico, A. Pataricza, A. Bondavalli, and T. Bartha. "Dependability Evaluation of Web Service-Based Processes." In: *Proceedings of the 3rd European Performance Engineering Workshop*. EPEW'06 (Budapest, Hungary, June 21–22, 2006). Ed. by A. Horváth and M. Telek. Vol. 4054. LNCS. Springer, 2006, pp. 166–180 — *cited on page* 26.

[79] K. Goseva-Popstojanova, A. Hassan, A. Guedem, W. Abdelmoez, D. E. M. Nassar, H. Ammar, and A. Mili. "Architectural-Level Risk Analysis Using UML." In: *IEEE Transactions on Software Engineering* 29.10 (Oct. 2003), pp. 946–960 — *cited on page* 25.

[80] G. Goth. "Beware the March of this IDE: Eclipse is overshadowing other tool technologies." In: *IEEE Software* 22.4 (July-August 2005), pp. 108–111 — *cited on pages* 36, 97, 102, 109.

[81] V. Grassi, R. Mirandola, and A. Sabetta. "Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach." In: *Journal of Systems and Software* 80.4 (2007), pp. 528–558 — *cited on pages* 25, 69, 100.

[82] S. Gérard, C. Dumoulin, P. Tessier, and B. Selic. "Papyrus: A UML2 Tool for Domain-Specific Language Modeling." In: *Model-Based Engineering of Embedded Real-Time Systems. International Dagstuhl Workshop, Revised Selected Papers*. (Dagstuhl Castle, Germany, Nov. 4–9, 2007). Ed. by H. Giese, G. Karsai, E. Lee, B. Rumpe, and B. Schätz. Vol. 6100. LNCS. Springer, 2011, pp. 361–368 — *cited on pages* 36, 97.

[83] P. G. Harrison and B. Strulo. "SPADES - a Process Algebra for Discrete Event Simulation." In: *Journal of Logic and Computation* 10.1 (Jan. 2000), pp. 3–42 — *cited on page* 12.

[84] B. R. Haverkort, H. Hermanns, and J.-P. Katoen. "On the Use of Model Checking Techniques for Quantitative Dependability Evaluation." In: *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*. SRDS'00 (Nürnberg, Germany, Oct. 16–18, 2000). IEEE, 2000, pp. 228–237 — *cited on page* 14.

[85] B. R. Haverkort and I. G. Niemegeers. "Performability Modelling Tools and Techniques." In: *Performance Evaluation* 25.1 (1996), pp. 17–40 — *cited on page* 14.

[86] A. Hegedus, G. Bergmann, I. Ráth, and D. Várró. "Back-annotation of Simulation Traces with Change-Driven Model Transformations." In: *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods*. SEFM'10 (Pisa, Italy, Sept. 13–18, 2010). IEEE, 2010, pp. 145–155 — *cited on page* 30.

[87] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. "On Language-Independent Model Modularisation." In: *Transactions on Aspect-Oriented Software Development VI. Special Issue on Aspects and Model-Driven Engineering*. Ed. by S. Katz, H. Ossher, R. France, and J.-M. Jézéquel. Springer, 2009, pp. 39–82 — *cited on pages* 150–151.

[88] H. Hermanns and M. Rettelbach. "Syntax, Semantics, Equivalences, and Axioms for MTIPP." In: *Proceedings of the 2nd Workshop on Process Algebras and Performance Modelling*. PAPM'94 (Erlangen, Germany, July 1994). 1994, pp. 71–87 — *cited on page* 12.

[89] *HIDE: "High Level Integrated Design Enviroment for Dependability"*. 1998 — *cited on page* 37.

[90] L. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Trèves. "The Petri Net Markup Language and ISO/IEC 15909-2." In: *Proceedings of the 10th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*. CPN'09 (Aarhus, Denmark, Oct. 19–21, 2009). 2009 — *cited on pages* 100, 103, 149.

[91] J. Hillston. "A Compositional Approach to Performance Modeling." PhD thesis. Cambridge University Press, 1995 — *cited on pages* 12–13.

[92] G. Horton, V. G. Kulkarni, D. M. Nicol, and K. S. Trivedi. "Fluid stochastic Petri nets: Theory, applications, and solution techniques." In: *European Journal of Operational Research* 105.1 (Feb. 1998), pp. 184–201 — *cited on page* 13.

[93] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. "Software rejuvenation: analysis, module and applications." In: *25th International Symposium on Fault-Tolerant Computing, Digest of Papers*. FTCS-25 (Pasadena, CA, USA, June 27–30, 1995). IEEE, 1995, pp. 381–390 — *cited on pages* 65, 113.

[94] H.Zhao, D.Song, and Y.Dong. "Design and Implementation of AADL Model Safety Assessment Tool." In: *Proceedings of the 12th International Conference on Quality Software*. QSIC'12 (Xi'an, China, Aug. 27–29, 2002). IEEE, 2012, pp. 251–257 — *cited on page* 26.

[95] *IEC 60125: "Fault tree analysis (FTA)"*. Second Edition. Dec. 2006 — *cited on page* 8.

[96] *IEC 60812: "Analysis techniques for system reliability – Procedure for failure mode and effects analysis (FMEA)"*. Second Edition. Jan. 2006 — *cited on page* 39.

[97] *ISO/IEC 15909-1: "Software and Systems Engineering – High-level Petri nets – Part 1: Concepts, definitions and graphical notation"*. 2004 — *cited on page* 103.

[98] *ISO/IEC 15909-2: "Software and Systems Engineering – High-level Petri nets – Part 2: Transfer format"*. 2011 — *cited on page* 103.

[99] *ISO/IEC Standard for Systems and Software Engineering - Recommended Practice for Architectural Description of Software-Intensive Systems*. ISO/IEC 42010, IEEE Std 1471-2000. 2007 — *cited on page* 32.

[100] J. Jürjens. "UMLsec: Extending UML for Secure Systems Development." In: *Proceedings of the 5th International Conference on The Unified Modeling Language*. UML'02 (Dresden, Germany, Sept. 30–Oct. 4, 2002). Springer, 2002, pp. 412–425 — *cited on page* 25.

[101] M. Kaâniche, P. Lollini, A. Bondavalli, and K. Kanoun. "Modeling the resilience of large and evolving systems." In: *International Journal of Performability Engineering* 4.2 (Apr. 2008), pp. 153–168 — *cited on page* 13.

[102] K. Kanoun and M. Ortalo-Borrel. "Fault-tolerant system dependability-explicit modeling of hardware and software component-interactions." In: *IEEE Transactions on Reliability* 49.4 (2000), pp. 363–376 — *cited on page* 141.

[103] J. Kemeny and J. Snell. *Finite Markov Chains*. D. Van Nostrand Company, Inc., 1960 — *cited on page* 13.

[104] C. Kobryn. "UML 2001: A Standardization Odyssey." In: *Communications of the ACM* 42.10 (Oct. 1999), pp. 29–37 — *cited on page* 21.

[105] M. Kovács, P. Lollini, I. Majzik, and A. Bondavalli. "An integrated framework for the dependability evaluation of distributed mobile applications." In: *Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*. (Newcastle upon Tyne, UK, Nov. 17–19, 2008). ACM, 2008, pp. 29–38 — *cited on page* 25.

[106] M. Kuntz, M. Siegle, and E. Werner. "Symbolic Performance and Dependability Evaluation with the Tool CASPA." In: *Applying Formal Methods: Testing, Performance, and M/E-Commerce. Proceedings of FORTE 2004 Workshops*. FORTE'04 (Toledo, Spain, Oct. 1–2, 2004). Vol. 3236. LNCS. Springer, 2004, pp. 293–307 — *cited on page* 15.

[107] M. Kwiatkowska, G. Norman, and D. Parker. "PRISM: Probabilistic Model Checking for Performance and Reliability Analysis." In: *ACM SIGMETRICS Performance Evaluation Review* 36.4 (2009), pp. 40–45 — *cited on page* 15.

[108] E. Lamboray, A. Zollinger, O. G. Staadt, and M. Gross. "Interactive Multimedia Streams in Distributed Applications." In: *Computer & Graphics* 27.5 (2003) — *cited on page* 112.

[109] L. Lednicki, A. Petricic, and M. Zagar. "A Component-Based Technology for Hardware and Software Components." In: *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications*. SEAA'09 (Patras, Greece, Aug. 27–29, 2009). IEEE, Aug. 2009 — *cited on page* 19.

[110] C. Lindemann, A. Reuys, and A. Thummler. "The DSPNexpress 2000 performance and dependability modeling environment." In: *29th Annual International Symposium on Fault-Tolerant Computing, Digest of Papers*. FTCS-29 (Madison, WI, USA, June 15–18, 1999). 1999, pp. 228–231 — *cited on page* 14.

[111] Y. Liu, X. Wang, and L. Zhao. "Scalable Video Streaming in Wireless Mesh Networks for Education." In: *International Journal of Distance Education Technology* 9.1 (Jan. 2011), pp. 1–20 — *cited on page* 111.

[112] P. Lollini, A. Bondavalli, and F. di Giandomenico. "A Decomposition-Based Modeling Framework for Complex Systems." In: *IEEE Transactions on Reliability* 58.1 (Mar. 2009), pp. 20–33 — *cited on page* 13.

[113] P. Lollini, L. Montecchi, and A. Bondavalli. *On the evaluation of HIDENETS use-cases having phased behavior*. Technical Report RCL-071201. Università degli Studi di Firenze, Dipartimento di Sistemi e Informatica, Dec. 2007 — *cited on pages* 142, 163, 165–166, 203.

[114] P. Lollini. "On the Modeling and Solution of Complex Systems: From Two Domain-Specific Case-Studies Towards the Definition of a More General Framework." PhD thesis. Università degli Studi di Firenze, Dottorato in Informatica e Applicazioni (XVIII ciclo), Dec. 2005 — *cited on page* 13.

[115] M. Magyar and I. Majzik. "Modular Construction of Dependability Models from System Architecture Models: A Tool-Supported Approach." In: *Proceedings of the 6th International Conference on the Quantitative Evaluation of Systems*. QEST'09 (Budapest, Hungary, Sept. 13–16, 2009). 2009, pp. 95–96 — *cited on page* 26.

[116] V. Mainkar and K. Trivedi. "Sufficient conditions for existence of a fixed point in stochastic reward net-based iterative models." In: *IEEE Transactions on Software Engineering* 22.9 (Sept. 1996), pp. 640–653 — *cited on page* 13.

[117] I. Majzik, A. Pataricza, and A. Bondavalli. "Stochastic Dependability Analysis of System Architecture Based on UML Models." In: *Architecting Dependable Systems*. Ed. by R. De Lemos, C. Gacek, and A. Romanovsky. Vol. 2677. LNCS. Springer, 2003, pp. 219–244 — *cited on pages* 25, 69.

[118] I. Malavolta, H. Muccini, P. Pelliccione, and D. Tamburri. "Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies." In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 119–140 — *cited on page* 50.

[119] M. Malhotra. "An efficient stiffness-insensitive method for transient analysis of Markov availability models." In: *IEEE Transactions on Reliability* 45.3 (1996), pp. 426–428 — *cited on page* 14.

[120] M. Malhotra and K. Trivedi. "Power-hierarchy of dependability-model types." In: *IEEE Transactions on Reliability* 43.3 (1994), pp. 493–502 — *cited on page* 8.

[121] N. Maruyama, A. Nukada, and S. Matsuoka. "A high-performance fault-tolerant software framework for memory on commodity GPUs." In: *Proceedings of the IEEE International Symposium on Parallel Distributed Processing*. IPDPS'10 (Apr. 19–23, 2010). IEEE, 2010, pp. 1–12 — *cited on page* 112.

[122] C. Meng, T. Wang, W. Chou, S. Luan, Y. Zhang, and Z. Tian. "Remote surgery case: robot-assisted teleneurosurgery." In: *Proceedings of the IEEE International Conference on Robotics and Automation*. ICRA'04 (New Orleans, LA, USA, Apr. 26–May 1, 2004). Vol. 1. IEEE, 2004, pp. 819–823 — *cited on page* 111.

[123] J. Merseguer, J. Campos, S. Bernardi, and S. Donatelli. "A compositional semantics for UML state machines aimed at performance evaluation." In: *Proceedings of the 6th International Workshop on Discrete Event Systems*. WODES'02 (Zaragoza, Spain, Oct. 2–4, 2002). 2002, pp. 295–302 — *cited on page* 25.

[124] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997 — *cited on pages* 17–18.

[125] J. F. Meyer. "On Evaluating the Performability of Degradable Computing Systems." In: *IEEE Transactions on Computers* C-29.8 (Aug. 1980), pp. 720–731 — *cited on pages* iii, 2.

[126] Microsoft. *COM: Component Object Model Technologies*. URL: `http://www.microsoft.com/com/default.mspx` (visited on 12/28/2013) — *cited on pages* 19, 195.

[127] *Möbius Manual*. Version 2.4, Rev. 1. University of Illinois at Urbana-Champaign, PERFORM Group. Dec. 2012 — *cited on page* 142.

[128] A. Möller, J. Fröberg, and M. Nolin. "Industrial Requirements on Component Technologies for Embedded Systems." In: *Proceedings of the 7th International Symposium on Component-Based Software Engineering*. Ed. by I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. Wallnau. Vol. 3054. LNCS. Springer, May 24–25, 2004, pp. 146–161 — *cited on page* 19.

[129] M. K. Molloy. "Performance Analysis Using Stochastic Petri Nets." In: *IEEE Transactions on Computers* 31.9 (Sept. 1982), pp. 913–917 — *cited on page* 11.

[130] L. Montecchi, P. Lollini, and A. Bondavalli. *An Intermediate Dependability Model for state-based dependability analysis*. Technical Report RCL101115. Università degli Studi di Firenze, Resilient Computing Lab, Jan. 2011 — *cited on pages* 69–70.

[131] L. Montecchi, P. Lollini, and A. Bondavalli. "Towards a MDE Transformation Workflow for Dependability Analysis." In: *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems*. ICECCS'11 (Las Vegas, NV, USA, Apr. 27–29, 2011). IEEE, 2011, pp. 157–166 — *cited on pages* 69, 75.

[132] L. Montecchi, N. Nostro, N. Veeraraghavan, R. Vitenberg, H. Meling, and A. Bondavalli. *Stochastic Activity Networks model for the evaluation of the World Opera system*. Technical Report RCL-131001. Università degli Studi di Firenze, Resilient Computing Lab, Oct. 2013 — *cited on pages* 144–145.

[133] L. Montecchi. *Module 05: State-based Dependability Analysis*. CHESS Project Training Video. URL: `http://www.chess-project.org/page/training` (visited on 12/28/2013) — *cited on pages* 97, 111.

[134] L. Montecchi, P. Lollini, and A. Bondavalli. "Dependability Concerns in Model-Driven Engineering." In: *IEEE International Symposium on Object/-Component/Service-Oriented Real-Time Distributed Computing Workshops*. WORNUS'11 (Newport Beach, CA, USA, Mar. 28–31, 2011). IEEE, 2011, pp. 254–263 — *cited on pages* 39, 70, 73, 123.

[135] A. van Moorsel and Y. Huang. "Reusable software components for performability tools, and their utilization for web-based configuration tools." In: *Proceedings of the 10th International Conference in Computer Performance Evaluation: Modeling Techniques and Tools*. TOOLS'98 (Palma de Mallorca, Spain, Sept. 14–18, 1998). Ed. by R. Puigjaner, N. N. Savino, and B. Serra. Vol. 1469. LNCS. 1998, pp. 37–50 — *cited on page* 15.

[136] A. van Moorsel and W. H. Sanders. "Adaptive Uniformization." In: *ORSA Communications in Statistics: Stochastic Models* 10.3 (Aug. 1994), pp. 619–648 — *cited on page* 14.

[137] M. Moretto. "Progettazione, realizzazione ed utilizzo di un generatore di simulatori per sistemi a fasi multiple." Italian. *English title*: "Design, realization, and application of a generator of simulators for multiple-phase systems". Master's thesis. Università degli Studi di Pisa, Corso di Laurea in Ingegneria Informatica, Dec. 2004 — *cited on page* 102.

[138] J. K. Muppala, M. Malhotra, and K. S. Trivedi. "Stiffness-tolerant methods for transient analysis of stiff Markov chains." In: *Microelectronics and Reliability* 34 (1994), pp. 1825–1841 — *cited on page* 14.

[139] T. Murata. "Petri nets: Properties, analysis and applications." In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580 — *cited on page* 9.

[140] M. Nelli, A. Bondavalli, and L. Simoncini. "Dependability Modelling and Analysis of Complex Control Systems: an Application to Railway Interlocking." In: *Proceedings of the 2nd European Dependable Computing Conference*. EDCC-2 (Taormina, Italy, Oct. 2–4, 1996). 1996, pp. 93–110 — *cited on page* 13.

[141] D. M. Nicol, W. H. Sanders, and K. S. Trivedi. "Model-based evaluation: from dependability to security." In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan.-March 2004), pp. 48–65 — *cited on pages* 6–7, 13.

[142] W. D. Obal II and W. H. Sanders. "State-space support for path-based reward variables." In: *Proceedings of the 1998 International Computer Performance and Dependability Symposium*. IPDS'98 (Durham, NC, USA, Sept. 7–9, 1998). 1998, pp. 228–237 — *cited on page* 14.

[143] Object Management Group. *A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Version 1.1*. OMG Document {formal/2011-06-02}. June 2011 — *cited on pages* 23, 47, 50, 54.

[144] Object Management Group. *Assuring Dependability of Consumer Devices – Request For Information*. OMG Document {sysa/2011-12-02}. Dec. 2011 — *cited on page* 24.

[145] Object Management Group. *Common Object Request Broker Architecture (CORBA)®*. URL: `http://www.omg.org/spec/CORBA/` (visited on 12/28/2013) — *cited on pages* 19, 195.

[146] Object Management Group. *Dependability Assurance Framework For Safety-Sensitive Consumer Devices – Request For Proposal*. OMG Document {sysa/2013-03-13}. Mar. 2013 — *cited on page* 24.

[147] Object Management Group. *MDA Guide Version 1.0.1*. OMG Document {omg/03-06-01}. June 2003 — *cited on page* 30.

[148] Object Management Group. *OMG Systems Modeling Language (OMG SysML), Version 1.3*. OMG Document {formal/2012-06-01}. June 2012 — *cited on pages* 23, 32, 48.

[149] Object Management Group. *OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1*. OMG Document {formal/2011-08-05}. Aug. 2011 — *cited on page* 21.

[150] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*. OMG Document {formal/2011-08-06}. Aug. 2011 — *cited on pages* 21–22, 37, 58, 75.

[151] Object Management Group. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (OMG QoS&FT), Version 1.1*. OMG Document {formal/2008-04-05}. Apr. 2008 — *cited on pages* 23, 46.

[152] Object Management Group. *UML Profile for Schedulability, Performance, and Time Specification (OMG SPT), Version 1.1*. OMG Document {formal/05-01-02}. Jan. 2005 — *cited on pages* 23, 47.

[153] G. J. Pai and J. B. Dugan. "Automatic synthesis of dynamic fault trees from UML system models." In: *Proceedings of the 13th International Symposium on Software Reliability Engineering*. ISSRE'02 (Annapolis, MD, USA, Nov. 12–15, 2002). 2002, pp. 243–254 — *cited on page* 24.

[154] M. Panunzio and T. Vardanega. "A Component Model for On-board Software Applications." In: *36th EUROMICRO Conference on Software Engineering and Advanced Applications*. SEAA'10 (Lille, France, Sept. 1–3, 2010). 2010, pp. 57–64 — *cited on page* 33.

[155] M. Panunzio and T. Vardanega. "Pitfalls and misconceptions in component-oriented approaches for real-time embedded systems: lessons learned and solutions." In: *Proceedings of the 3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*. CRTS'10 (San Diego, CA, USA, Nov. 30, 2010). 2010 — *cited on page* 26.

[156] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules." In: *Communications of the ACM* 15.12 (Dec. 1972), pp. 1053–1058 — *cited on page* 17.

[157] C. A. Petri. "Kommunikation mit Automaten." German. PhD thesis. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962 — *cited on page* 9.

[158] B. Plateau and K. Atif. "Stochastic automata network for modeling parallel systems." In: *IEEE Transactions on Software Engineering* 17.10 (Oct. 1991), pp. 1093–1108 — *cited on pages* 12, 140.

[159] B. Plateau. "On the stochastic structure of parallelism and synchronization models for distributed algorithms." In: *Proceedings of the 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS'85 (Austin, TX, USA, Aug. 26–29, 1985). New York, NY, USA: ACM, 1985, pp. 147–154 — *cited on page* 13.

[160] M. Rabah and K. Kanoun. "Performability evaluation of multipurpose multiprocessor systems: the "separation of concerns" approach." In: *IEEE Transactions on Computers* 52.2 (2003), pp. 223–236 — *cited on page* 141.

[161] S. M. Rinaldi, J. P. Peerenboom, and T. K. Kelly. "Identifying, understanding, and analyzing critical infrastructure interdependencies." In: *IEEE Control Systems Magazine* 21.6 (Dec. 2001), pp. 11–25 — *cited on page* 139.

[162] A.-E. Rugina, K. Kanoun, and M. Kaâniche. "The ADAPT Tool: From AADL Architectural Models to Stochastic Petri Nets through Model Transformation." In: *Proceedings of the 7th European Dependable Computing Conference*. EDCC'08 (Kaunas, Lithuania, May 7–9, 2008). 2008, pp. 85–90 — *cited on page* 26.

[163] A.-E. Rugina, K. Kanoun, and M. Kaâniche. "A system dependability modeling framework using AADL and GSPNs." In: *Architecting Dependable Systems IV*. Ed. by R. de Lemos, C. Gacek, and A. Romanovsky. Vol. 4615. Springer, 2007, pp. 14–38 — *cited on page* 24.

[164] R. Sahner, K. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems. An Example-Based Approach Using the SHARPE Software Package*. Kluwer Academic Publishers, 1996 — *cited on page* 13.

[165] W. H. Sanders. "Integrated frameworks for multi-level and multi-formalism modeling." In: *Proceedings of the 8th International Workshop on Petri Nets and Performance Models*. PNPM'99 (Zaragoza, Spain, Sept. 6–10, 1999). IEEE, 1999, pp. 2–9 — *cited on page* 14.

[166] W. H. Sanders and J. F. Meyer. "Reduced base model construction methods for stochastic activity networks." In: *IEEE Journal on Selected Areas in Communications* 9.1 (Jan. 1991), pp. 25–36 — *cited on pages* 140, 142, 147.

[167] W. H. Sanders and J. F. Meyer. "Stochastic activity networks: formal definitions and concepts." In: *Lectures on Formal Methods and Performance Analysis*. Ed. by E. Brinksma, H. Hermanns, and J.-P. Katoen. Vol. 2090. Springer, 2002, pp. 315–343 — *cited on pages* 12, 142, 147.

[168] D. C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering." In: *Computer* 39.2 (2006), pp. 25–31 — *cited on pages* iii, 19–20.

[169] M. Serafini, P. Lollini, and A. Bondavalli. "Modeling on-line tests in safety-critical systems." In: Safety and Reliability for Managing Risk. London: Taylor & Francis Group, 2006 — *cited on pages* 141–142.

[170] C. U. Smith, C. M. Lladó, and R. Puigjaner. "Performance Model Interchange Format (PMIF 2): A Comprehensive Approach to Queueing Network Model Interoperability." In: *Performance Evaluation* 67.7 (July 2010), pp. 548–568 — *cited on page* 100.

[171] Society of Automotive Engineers. *Architecture Analysis & Design Language (AADL)*. SAE Standards: AS5506. Nov. 2004 — *cited on page* 49.

[172] Society of Automotive Engineers. *Architecture Analysis & Design Language (AADL) Annex Volume 1*. SAE Standards: AS5506/1. June 2006 — *cited on pages* 24, 49, 58.

[173] M. Sridharan, S. Ramasubramanian, and A. K. Somani. "HIMAP: Architecture, Features, and Hierarchical Model Specification Techniques." In: *Proceedings of the 10th International Conference in Computer Performance Evaluation: Modeling Techniques and Tools*. TOOLS'98 (Palma de Mallorca, Spain, Sept. 14–18, 1998). Ed. by R. Puigjaner, N. N. Savino, and B. Serra. Vol. 1469. LNCS. Springer, 1998, pp. 348–351 — *cited on page* 15.

[174] M. Stamatelatos, W. Vesely, J. Dugan, J. Fragola, J. Minarick, and J. Railsback. *Fault Tree Handbook with Aerospace Applications*. Version 1.1. NASA, Aug. 2002 — *cited on pages* iii, 7.

[175] Sun Microsystems. *JavaBeans™*. Version 1.01. Aug. 1997 — *cited on page* 19.

[176] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Second Edition. Addison-Wesley Professional, 2002 — *cited on page* 17.

[177] C.-W. Ten, C.-C. Liu, and M. Govindarasu. "Vulnerability Assessment of Cybersecurity for SCADA Systems Using Attack Trees." In: *Power Engineering Society General Meeting*. (Tampa, FL, USA, June 24–28, 2007). IEEE, 2007, pp. 1–8 — *cited on page* 8.

[178] M. Tribastone, A. Duguid, and S. Gilmore. "The PEPA eclipse plugin." In: *ACM SIGMETRICS Performance Evaluation Review* 36.4 (Mar. 2009), pp. 28–33 — *cited on page* 15.

[179] K. Trivedi. "SHARPE 2002: Symbolic Hierarchical Automated Reliability and Performance Evaluator." In: *Proceedings of the International Conference on Dependable Systems and Networks*. DSN'02 (Bethesda, MD, USA, June 23–26, 2002). 2002, p. 544 — *cited on page* 15.

[180] U.S. Department of Defense. *Military handbook – Reliability prediction of electronic equipment*. MIL-HDBK-217F. Jan. 1990 — *cited on page* 129.

[181] U.S. Department of Defense. *Systems Engineering Guide for Systems of Systems*. Version 1.0. Aug. 2008 — *cited on page* iv.

[182] N. Veeraragavan, L. Montecchi, N. Nostro, A. Bondavalli, R. Vitenberg, and H. Meling. "Understanding the Quality of Experience in Modern Distributed Interactive Multimedia Applications in Presence of Failures: Metrics and Analysis." In: *Proceedings of the 28th ACM Symposium on Applied Computing*. SAC'13 (Coimbra, Portugal, Mar. 18–22, 2013). DADS Track. ACM, 2013 — *cited on pages* 141, 144–145.

[183] N. Veeraragavan, R. Vitenberg, and H. Meling. "Reliability Modeling and Analysis of Modern Distributed Interactive Multimedia Applications: A Case Study of a Distributed Opera Performance." In: *Proceedings of the 12th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*. Vol. 7272. LNCS. Springer Berlin Heidelberg, 2012, pp. 185–193 — *cited on pages* 142–143.

[184] V. Vittorini, M. Iacono, N. Mazzocca, and G. Franceschinis. "The OsMoSys approach to multi-formalism modeling of systems." In: *Software and Systems Modeling* 3.1 (2004), pp. 68–81 — *cited on pages* 15, 147.

[185] M. Völter. "MD* Best Practices." In: *Journal of Object Technology* 8.6 (Sept. 2009), pp. 79–102 — *cited on page* 156.

[186] M. Walter, C. Trinitis, and W. Karl. "OpenSESAME: an intuitive dependability modeling environment supporting inter-component dependencies." In: *Proceedings of the 3rd Pacific Rim International Symposium on Dependable Computing*. PRDC'01 (Seoul, Korea, Dec. 17–19, 2001). 2001, pp. 76–83 — *cited on page* 26.

[187] *Web Services Business Process Execution Language (WS-BPEL)*. Version 2.0. Organization for the Advancement of Structured Information Standards (OASIS), Apr. 2007 — *cited on page* 26.

[188] *XSL Transformations (XSLT), Version 2.0*. W3C Recommendation. Jan. 2007 — *cited on page* 156.

[189] *Xtext – Language Development Made Easy!* URL: `http://www.eclipse.org/Xtext/` (visited on 12/28/2013) — *cited on page* 156.

[190] A. Zarras, P. Vassiliadis, and V. Issarny. "Model-Driven Dependability Analysis of WebServices." In: *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE. Proceedings of the OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004, Part II*. Ed. by R. Meersman and Z. Tari. Vol. 3291. LNCS. Springer, 2004, pp. 1608–1625 — *cited on page* 26.

[191] A. Zimmermann. "Modeling and evaluation of stochastic Petri nets with TimeNET 4.1." In: *Proceedings of the 6th International Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS'12 (Cargèse, France, Oct. 9–12, 2012). 2012, pp. 54–63 — *cited on page* 14.

APPENDICES

A

ACRONYMS

AADL	Architecture Analysis & Design Language
ARTEMIS	Advanced Research & Technology for EMbedded Intelligence and Systems
ARTEMIS-JU	ARTEMIS Joint Undertaking
ASIL	Automotive Safety Integrity Level
ATL	ATLAS Transformation Language
BPEL	Business Process Execution Language
CBD	Component-Based Development
CBSE	Component-Based Software Engineering
CI	Critical Infrastructure
COM	Component Object Model [126]
CORBA	Common Object Request Broker Architecture [145]
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CS	Constituent Systems
CSD	Composite Structure Diagram
CSL	Continuous Stochastic Logic
CTMC	Continuous-Time Markov Chain
CUDA	Compute Unified Device Architecture
DAM	Dependability Analysis Modeling
DIMA	Distributed Interactive Multimedia Application
DSL	Domain Specific Language
DSPN	Deterministic and Stochastic Petri Net
DTMC	Discrete-Time Markov Chain
FMECA	Failure Mode, Effects and Criticality Analysis
FPTC	Failure Propagation and Transformation Calculus
FT	Fault Tree
FTA	Fault Tree Analysis
GCM	Generic Component Model

GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
GRM	Generic Resource Modeling
GSPN	Generalized Stochastic Petri Net
HIDENETS	HIghly DEpendable NETworks and Services
HRM	Hardware Resource Modeling
IT	Information Technology
KLAPER	Kernel LAnguage for PErformance and Reliability analysis
MC	Markov Chain
MARTE	Modeling and Analysis of Real-Time Embedded systems
MDA	Model-Driven Architecture
MDD	Multivalued Decision Diagram
MDE	Model-Driven Engineering
MTIPP	Markovian Timed Processes for Performance Evaluation
MTTF	Mean Time To Failure
MTTHE	Mean Time To Hazardous Event
NFSD	No Fire/Smoke Detected
OCL	Object Constraint Language
OMG	Object Management Group
OOP	Object Oriented Programming
PEPA	Performance Evaluation Process Algebra
PIM	Platform Independent Model
PMIF	Performance Model Interchange Format
PN	Petri Net
PNML	Petri Nets Markup Language
PSM	Platform Specific Model
P/T	Place/Transition
QN	Queuing Networks
QoS	Quality of Service
QoS&FT	Quality of Service & Fault Tolerance
RBD	Reliability Block Diagram
RFI	Request For Information
RFP	Request For Proposal

RG	Reliability Graph
SAN	Stochastic Activity Network
SIL	Safety Integrity Level
SysML	Systems Modeling Language
SoS	System of Systems
SPN	Stochastic Petri Net
SPT	Schedulability, Performance, and Time
SRN	Stochastic Reward Net
TMDL	Template Models Description Language
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
VSL	Value Specification Language
V&V	Verification & Validation
WCET	Worst-Case Execution Time
WO	World Opera
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformations

B

LIST OF GRAPHICS

FIGURES

TABLES

LISTINGS