Software and Systems Modeling manuscript No.
(will be inserted by the editor)

The KlaperSuite Framework for Model-Driven Reliability Analysis of Component-Based Systems

Andrea Ciancone[1], **Mauro Luigi Drago**[1], **Antonio Filieri**[1,2], **Vincenzo Grassi**[3], **Heiko Koziolek**[4], **Raffaela Mirandola**[1]

[1] Politecnico di Milano, Milano, Italy
{filieri,drago,mirandola}@elet.polimi.it

[2] Reliable Software Systems Group, University of Stuttgart, Stuttgart, Germany
antonio.filieri@informatik.uni-stuttgart.de

[3] Universita' di Roma "Tor Vergata", Roma, Italy
vgrassi@info.uniroma2.it

[4] Industrial Software Systems, ABB Corporate Research, Ladenburg, Germany
heiko.koziolek@de.abb.com

Received: date / Revised version: date

Abstract Automatic prediction tools play a key role in enabling the application of non-functional requirements analysis, to simplify the selection and the assembly of components for component-based software systems, and in reducing the need for strong mathematical skills for software designers. By exploiting the paradigm of Model-Driven Engineering (MDE), it is possible to automatically transform design models into analytical models, thus enabling formal property verification.

MDE is the core paradigm of the KlaperSuite framework presented in this paper, which exploits the KLAPER pivot language to fill the gap between design and analysis of component-based systems for reliability properties. KlaperSuite is a family of tools empowering designers with the ability to capture and analyze QoS views of their systems, by building a one-click bridge towards a number of established verification instruments. In this article we concentrate on the reliability prediction capabilities of KlaperSuite and we evaluate them with respect to several case studies from literature and industry.

Keywords: Model Driven Engineering, Reliability Analysis, Component Based Systems

1 Introduction

Non-functional properties of software are critical in a large number of everyday applications. The pervasiveness of software intensive components spreads from avionic control systems to financial transactions management, up to intersecting the web navigation of everyone. End users do not only expect software to provide its intended functionality, but also to be dependable, performant, and cost-effective. Meeting these expectations is becoming a core aspect of software development processes. For this reason, the central role of non-functional properties has to be accounted for since the early stages of design. The high level design choices, such as the architectural ones, set the basis to achieve both functional and non-functional goals, and need to be supported by methodologies and tools able to capture these two dimensions of the product at the same time. In the practice of

past years, first the entire system is built, then its Quality of Service (Quality of Service (QoS)) is measured and, when violations to its requirements are discovered, developers have to try to identify the most appropriate improvements.

This practice may lead to several drawbacks. Late discovery of non-functional requirements violations can be harmful for the success of the development process itself. Indeed, the impact of changes on development costs and on failure risks may be non negligible, if changes are applied when a complete implementation of a system already exists [67].

In recent years, several techniques for early assessment of quality attributes have been proposed in literature, typically based on very specific quality-related formalisms such as Queuing Networks (Queuing Networks (QNs)) [43], Petri Nets (Petri Nets (PNs)) [56], or Markov Models (Markov Models (MMs)). However, given their very specific purpose, such formalisms are not suitable for representing many design concerns, and often require a deep knowledge of the underlying theoretical foundations to understand the provided results, which not every software engineer has. Software systems are in fact rarely designed starting from mathematical models: designers usually think at different abstraction levels and use domain-specific concepts that better reflect the modeling intent. The wide-spread usage of languages such as the Unified Modeling Language (UML) [53], SysML [52], or domain specific languages [17, 8] to design component-based systems [68] provide some evidences.

To overcome this modeling gap between quality-specific formalisms and high-level design languages, *model-based quality prediction* approaches [6, 8, 73, 67, 2,

40,7,10] have been proposed in literature. The idea behind these approaches is to leverage Model Driven Engineering (Model-Driven Engineering (MDE)) techniques — such as model transformations [5,18,23,9]— to relieve the engineer from the burden of manually creating and maintaining quality-oriented formal models. Indeed the quality-oriented models can usually by automatically derived from design-oriented abstractions. An example is the Performance by Unified Analysis (PUMA) approach described by Woodside et al. in [73], which proposes the adoption of UML augmented with performance information through profiles to model architectures. These design-oriented abstractions can be then automatically converted into a Layered Queuing Network (LQN) model and the quality of the system consequently predicted. The Palladio Component Model (PCM) model described in [8] is another notable example of the same paradigm. The PCM approach proposes its own modeling language to represent component-based software systems in place of UML and provides a comprehensive model-driven toolchain to analyze several quality attributes — not only performance — of designed systems.

In this article we present KlaperSuite, our model-driven proposal to support early-stage analysis of non-functional attributes for component-based systems [68] which we already introduced in [14].

The core idea behind KlaperSuite is to exploit a pivot model [32] to bridge the gap between design and analysis models and provide a comprehensive toolchain for QoS assessment. The pivot model is KLAPER, a close-to-design formalism that captures also relevant information for QoS analyses. KLAPER can represent

design concepts, such as components, behaviors, or single operations, as well as a broad and extensible set of QoS annotations. A number of automatic transformations from KLAPER to analysis models is provided by the KlaperSuite, that is also in charge of running the specific analyzers and bringing their results back to the designers in a completely transparent way. KlaperSuite is enabled by an extendible plugin-based architecture, allowing QoS specialists to define new model transformations from KLAPER to other useful existing analysis tools.

From a designer perspective, it is possible to define a model transformation for her preferred modeling language (e.g. UML) to KLAPER, and then let the KlaperSuite run any of the available analyses with no need to deal with the semantics of the many underlying analysis models. This approach enhances the reuse of KLAPER-based analysis tools and make available all of them in a unified interface. KlaperSuite also provides support for direct definition of KLAPER models as first class artifacts. Indeed KLAPER embeds the most common design concepts and can be possibly used as first modeling language.

Finally, KlaperSuite provides access to all the automatically generated analytical models for further investigation by expert users.

The main contribution of this paper is two-fold. First we describe our comprehensive tool-chain based on the KLAPER language — which previously was only a stand-alone proposal and lacked the level of integration necessary for its usage in real scenarios — by concentrating especially on its reliability prediction features. Second, we validate our approach, and specifically its reliability prediction features, by applying KlaperSuite first to the analysis of cases taken from litera-

ture and then on an industrial problem. The results show that KlaperSuite provide results as accurate as the compared approaches taken from literature and that it is efficient enough to scale on large industrial-strength problems.

The rest of the article is organized as follows. Section 2 introduces the KLAPER language, its usage scenarios, and outlines the benefits of using a pivot language. Section 3 describes the KlaperSuite framework, pin-points its relationship with KLAPER, and outlines the supported QoS analysis tools. Section 4 instead describes in detail the tools provided by KlaperSuite to perform reliability analyses. In Section 5 we present our case studies to show the capabilities of our framework, while Sections 6 and 7 describe related work and future research directions, respectively.

2 The KLAPER Approach

The KlaperSuite framework is built upon the concepts provided by the KLAPER intermediate language and upon its model transformations and analysis capabilities. In this section we briefly describe KLAPER in order to introduce some useful concepts that will be used in the rest of this paper. First we pinpoint why intermediate languages are useful to ease QoS analysis tasks — both from the point of view of system designers and of engineers building frameworks to perform such analysis — and what distinguishes them from other modeling notations. Then we introduce the two main usage scenarios of KLAPER, i.e., either system designers directly use it to model systems or it is used as a hidden bridge to fill the

gap between design models and QoS analysis models. Finally we provide a brief description of the concepts provided by KLAPER.

2.1 Overview and Motivation

KLAPER is an intermediate modeling language [32] whose main goals are to separate system design models from quality-related models and to ease the translation among them. Design and quality-related models are very different kinds of abstractions, both from a syntactic perspective and from a semantic perspective. The former are more *user-oriented* and talk in terms of concepts closer to the engineering domain area. A well known example is UML [53] and the concepts it provides to specify the architecture of a system in the form of *class*, *component*, and *deployment* models. The latter concentrate instead on quality and are centered around the QoS analysis techniques to compute predictions. In this sense a notable example is represented by the Queuing Network formalism, which provides a user-friendly notation to describe how jobs and computations are performed by a system and flow through the various resources, but that can be hardly used to describe different *non quality-related* facets of a system.

When quality is an important aspect for a software system, both these kinds of abstractions are necessary. Given this aforementioned mismatch of information, the design models specifying for example the architecture of the system are not well-suited for QoS analyses, and complementary quality models are necessary to use the existing QoS prediction techniques available in literature. Software system design is however an iterative and interactive process in which models undergo

several refinement stages. If both architectural and quality-related models are used by system designers, keeping them consistent during the whole process is mandatory and to be effective it demands automation.

In this sense, mechanisms to link system design models with quality-related models are necessary. These mechanisms should be devised for both directions: from design model to quality-related models, to support automated investigation of QoS properties of possible design solutions, and in the opposite direction to bring analysis results back to design models (e.g. by automating the insertion in the design models of suggested modifications). We review in Section 6 proposed approaches covering these two issues. The focus of this paper is on mechanisms supporting the design model to quality-related model linking.

Different options are viable to achieve this goal. Either this linking/translation can be performed in a direct manner — system designs are translated into quality models in one single step — or the translation process can be split in two or more separate, and smaller, steps. Intermediate languages play a fundamental role in the latter case and, for what concerns KLAPER, the process of keeping consistent these two abstractions is split in two stages:

- **Extracting** from system design models all (and only) the information relevant for the analysis of QoS metrics and expressing it with an intermediate language (which in our case is KLAPER).
- **Generating** quality-related models from such information.

The KlaperSuite framework supports the former stage by the definition of the KLAPER language itself, whose linguistic constructs help in evidencing the

relevant information to be extracted. Then, KlaperSuite supports the latter stage by automating the transformation from a KLAPER model to a suitable analysis model. Using such a multi-stage approach has the advantage of reducing the semantic gap between consecutive stages, thus facilitating their definition and automation. Besides, adopting a model transformation approach based on an intermediate language has the positive side effect of alleviating the potential *"n-by-m"* problem, i.e., the problem of translating n heterogeneous design notations (that could be used by the different system architects or component providers) into m quality-related notations (each one supporting different quality metrics and prediction techniques). Indeed, by splitting the translation in two stages, the *"n-by-m"* problem is reduced to the definition of n transformations from the various design notations to the single intermediate language, and m transformations from the intermediate language to the different quality-related formalisms. Supporting new analysis techniques (or integrating into the framework an existing one) would then require only adding one new transformation from the intermediate representation to the specific quality-related formalism required by the new prediction technique; where the direct approach would have instead required writing n transformations. The same would hold if a new design language must be integrated. Instead of writing m new transformations to support all the available analysis formalisms, the two-staged approach requires only the development of one transformation to the intermediate representation.

2.2 Usage Scenarios

The main purpose of intermediate languages such as KLAPER is to ease the development and the usage of quality prediction frameworks by bridging the gap between design models and quality-related formalisms. Nonetheless, given their conciseness and expressiveness, intermediate languages can also be directly used by architects to design systems. Figure 1 depicts this fact in the specific case of KLAPER.

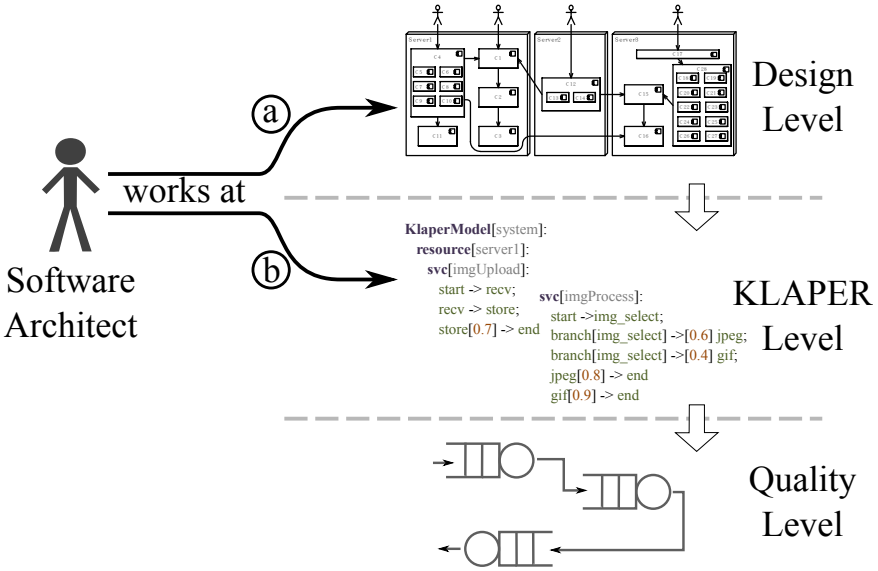


Fig. 1 KLAPER usage scenarios.

A software architect can possibly work at two different abstraction levels. In the first scenario (branch *a* in the Figure), architects design a system by using a design-oriented modeling notation such as UML [53] or SysML [52], intermediate KLAPER models are hidden from their perspective, automatically kept consistent

with the upper level models, and used to generate quality models and predict the system QoS (see also [32]). This is the standard scenario and the preferred way to leverage the possibilities offered by intermediate languages. The other viable option (branch *b* in the Figure) consists instead in working directly at the KLAPER level to model both the architecture of the system and its quality attributes [31]; the transformation facilities will then be used only to generate the quality models to predict the system QoS. This option is not advisable when new software systems are being developed, but it is useful in situations in which no design models of the system exist — for example when dealing with legacy applications — but architects want to assess the impact of some changes on the exhibited quality without having to reverse engineer a complete design model of the system being maintained.

2.3 The KLAPER Meta-model

KLAPER provides a set of modeling concepts to express in a compact and elegant manner both architectural and quality-related information for component-based software systems, by abstracting away at the same time all the irrelevant details. Figure 2 outlines the KLAPER meta-model and the relationship among the provided concepts[1]. As we will clarify later in Section 3, our language (and the associated framework) is mainly intended for supporting stochastic quality prediction techniques and, specifically, performance/reliability stochastic analyses for

[1] Some details (e.g., the attributes of the meta-classes) have been omitted for clarity and space reasons.

component-based systems. It is thus no surprise that the KLAPER meta-model reflects this fact and that it is centered around components and stochastic properties. In the rest of this section we provide a brief overview of how system designs can be represented with KLAPER. A detailed explanation of the language is available in [32].

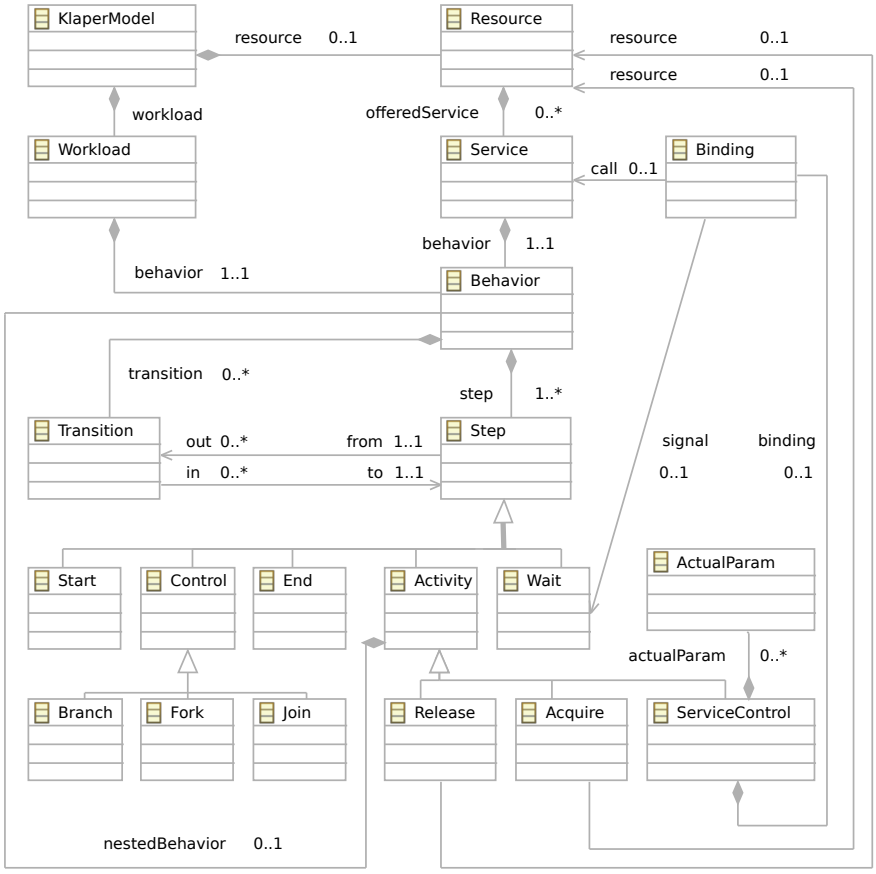


Fig. 2 The KLAPER meta-model.

In terms of KLAPER, a system is an assembly of *Resource*s modeling either software components or hardware components — such as a processor, a disk, or

a communication link — and offering/requiring one or more *Service*s. *Resource*s and *Service*s are the basic building blocks of the language and provide different properties to specify both functional and non-functional aspects. The *scheduling policy* and the *multiplicity* for the *Resource* meta-class are examples of such properties, which determine the specific algorithm used to complete service requests and the number of concurrent requests that can be served in parallel, respectively.

A *Service* models a piece of functionality and it may be characterized by specifying its *formal parameter*s which will be instantiated with actual values during service invocation. Formal parameters (and the corresponding actual parameters) provide a convenient abstraction of the real service parameters for analysis purposes, and are especially useful to support parametric QoS analyses [32]. For example, the functionality of a component responsible for processing a list of objects (for example a list of invoices) may be abstractly represented with a service accepting as formal input parameter an integer-valued random variable, whose probability distribution determines the likelihood of being invoked with lists of different sizes.

KLAPER also allows for the specification of how components work internally (*reactive* behavior) by attaching a behavioral specification to each *Service* and of how the system is used (*proactive* behavior) by attaching a behavioral specification to a *Workload*, which models the demand for the software system requested by external entities such as users. We recall that KLAPER is mainly concerned with QoS and, as a matter of fact, the behavior of a service is described in an abstract manner and from a stochastic perspective. In detail, a *Behavior* is a directed

graph of *Step*s, each one modeling an atomic piece of computation that may take time to complete and that may fail before its completion. A *Step* is intended to be a computational abstraction, that could encompass several lines of code of a real component. *Step*s can also be further described by specifying the *internalExecTime*, the *internalFailTime* and the *internalFailProb* attributes in order to give a probabilistic characterization of important quality-related aspects of execution. We point out here that the performance/reliability attributes associated to each *Step* only refer to the internal characteristics of the computation stage. They do not take into account the possible delays or failures caused by the invocation of other services required during the computation. These two aspect will then be composed together in the model analysis phase, to get the overall system performance and/or reliability.

Different kinds of *Step*s are supported by KLAPER. *Control Steps (Branch, Fork, Join)* can be used to regulate the flow of control from *Step* to *Step* in a probabilistic manner. Specifically, the *Branch* step models the conditional selection among different alternatives, and *Fork* and *Join* steps model parallel execution of activities. Loop control structures can be modeled in KLAPER in two different ways: either as guard-controlled loops, with a *Branch* step probabilistically modeling the guard that controls the exit from the loop, or by using the *Repetition* attribute of *Activities* and *Steps*.

ServiceCall steps can be used to model invocation of external services. A *ServiceCall* only specifies the existence of a relationship among required and offered services, the actual recipient of the call is captured separately through a *Binding*.

This distinction allows for a clear separation between the model of the components and the model of their composition. A set of bindings can be regarded as a self-contained specification of an assembly and, since the *ServiceCall* concept can also be used to model the access to platform level services from components, a set of bindings can model as well the deployment of the application components on the underlying platform.

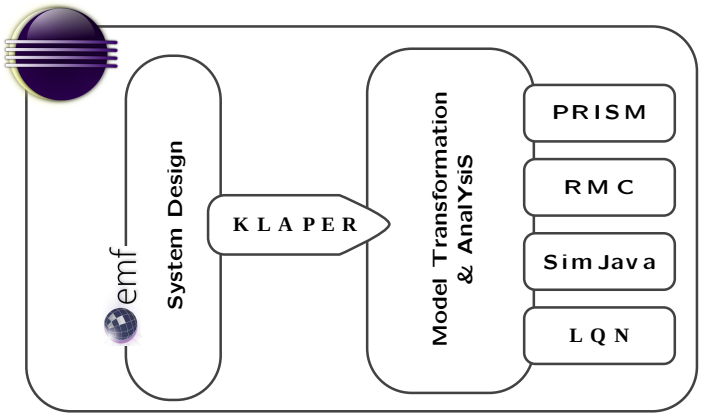
3 The KlaperSuite Framework

KlaperSuite is an integrated framework built upon the KLAPER intermediate language and upon its transformation and analysis features. The aim of KLAPER is to provide a foundation of concepts, meta-models and techniques to easily build tools, possibly based on existing quality prediction methodologies, to support the early assessment of the QoS for software systems. The goal of the KlaperSuite framework is instead to integrate those existing tools based on KLAPER into a user-friendly development environment.

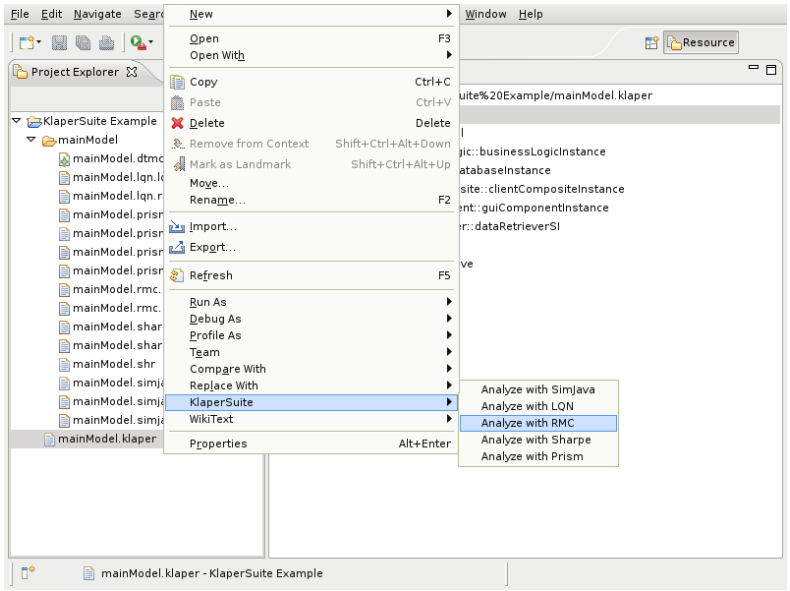
Providing an integrated workbench, where obtaining predictions for the QoS of the software system being designed is a one-click experience, is critical for a widespread adoption of a quality prediction methodology. We believe that the more a development environment seamlessly integrates QoS analysis techniques, the more practitioners will adopt those techniques and the more they will build reliable and performing software. As a consequence of this fact, KlaperSuite[2] has

[2] The framework can be downloaded from `http://home.dei.polimi.it/filieri/tools2011`

been built upon the Eclipse IDE [69], a de-facto standard both in academia and in industrial settings, in order to provide a unified interface and a familiar environment.



(a) KlaperSuite architecture.



(b) The KlaperSuite user experience.

Fig. 3 High level view of KlaperSuite.

Figure 3 shows a global view of what KlaperSuite is by outlining the relationship with KLAPER and the supported QoS analysis techniques in Figure 3(a), as well as what the user-experience looks like in Figure 3(b). The current implementation of KlaperSuite aims at providing a comprehensive family of tools to execute common QoS analysis tasks, ranging from prediction of reliability to performance. For what concerns reliability, KlaperSuite currently supports both a PRISM based analysis tool and a Recursive Markov Chain (RMC) based tool. This paper specifically focuses on the evaluation of the reliability prediction capabilities of KlaperSuite, hence we give an extensive overview of how these analyses work in Section 4, while we defer to Appendix A for details about the model transformations that allow to carry out the analysis. For what concerns performance, KLAPER already provides both an LQN based prediction tool and the *SimJava*[3]-based simulator (a description of these tools can be found in [14,60]). We are currently in the stage of refactoring and fully integrating these analyzers into KlaperSuite; their usability is currently limited.

All the tools embedded in the framework are fully automated, they require at most the setting of few configuration parameters in order to be run, and their execution follows the two-staged execution pattern we previously outlined in Section 2.1. First design models are automatically transformed into a KLAPER-based intermediate representation, then such intermediate models are transformed into an appropriate notation (specific for each kind of analysis) compatible with the prediction tool to be used. By doing so, engineers that want to integrate new anal-

[3] http://www.icsa.inf.ed.ac.uk/research/groups/hase/simjava/

ysis tools into KlaperSuite or that want to support new design notations can take advantage of the existing transformations with a consequent reduction of the development effort.

The whole suite is implemented as a plugin-based architecture within the Eclipse IDE. Extensions are possible by plugging additional analysis modules and the corresponding QVT model transformations.

The results of the analysis are then gathered from the underlying analysis tool and saved with a plain text format in the workspace.

Given the variety of formats supported by the underlying analysis tools, the different needs of each engineering domain and of each system architect, the current implementation of KlaperSuite does not provide any default parser or rich interface to show the results. However, we are currently working in this direction in order to implement parsers and rich user interfaces for the currently supported quality prediction tools.

4 Models and Tools for Reliability Analysis

This article focuses on the reliability prediction capabilities of KlaperSuite and on their validation. Thus we concentrate on them in this section and we provide an in depth description of the methodologies and tools supported by our framework.

In this paper we will refer to reliability, in a broad sense, as the probability of satisfying an assigned task [13]. Such a definition is also referred to as “reliability on demand” [28] and is particularly suitable for service oriented architecture, where a service, once invoked, has a certain probability to be successfully exe-

cuted. There are a number of probabilistic models for software reliability [36]. Among them, architecture-based approaches are frequently based on Discrete-Time Markov Chain (DTMC) models of the software's behavior [37]. KlaperSuite reliability analysis instruments fall in this category.

DTMCs can be roughly seen as finite state-transition automata where each state s_i has a certain probability p_{ij} to reach state s_j. As for probability theory, for each state s_i it holds that $\sum_j p_{ij} = 1$. The states of a DTMC are used to represent relevant states of the execution of a software system. For example, in KLAPER a state may represent an internal activity or the invocation of a *Service*. In a DTMC-based reliability analysis, it is common to extend the model of the system with a set of states that represent meta-conditions of the execution, that is, they correspond neither to internal activities nor to external invocations, but they rather correspond to failures or success. These *meta-states* are typically related to permanent conditions of the system, and thus their counterpart in the domain of DTMCs is represented by *absorbing states*, i.e., any state s_i such that $p_{ii} = 1$ is said to be absorbing, with the immediate meaning that state s_i, once reached, cannot be left.

Given a DTMC model of the software behavior, reliability can be then rephrased as the probability of reaching a convenient success state. KLAPER allows for the description of service oriented architectures that can be automatically translated into DTMC models. Each service behavior is defined in a structured-programming fashion, by composing *activities* through sequence, branch, loop, and fork-join control structures. Each activity can be defined as a black-box operation characterized by its own failure probability (P_f), or as a white-box with an associated

internal behavior (see Section 2). Moreover, each branch is labeled with the probability of taking one or other possible alternatives.

Such structured behaviors can be translated into a corresponding DTMC in a natural way by introducing a DTMC state to represent the execution of each atomic step, and then connecting them coherently with control structures [22]. We give in Appendix A.1 details about the actual rules we adopt to implement this translation. In particular, the rule adopted to implement the translation of Fork and Join structures into a DTMC is based on the assumption that parallel activities fail independently of each other. In this respect, we point out that this assumption, whose aim is to help in getting a tractable model, is at the basis of other state-of-the-art software architecture reliability models (see, for example, [71]). Dealing explicitly with the case of failure dependencies among parallel components would require a more general modeling approach. How to devise it is the subject of ongoing research [45].

Two complementary absorbing states (*end* and *failure*) represent success and failure respectively, and each node is connected to the *failure* state with its own failure probability by rescaling the other transition by a factor $1 - P_f$. Intuitively, execution moves to the next state if and only if the execution of the current step does not fail. Reaching the *end* step of a KLAPER behavior means that the execution has been successfully completed. The execution of software modeled in KLAPER starts by executing the users' behavior as described in the workloads for the system. As a matter of fact, reliability analysis is tailored to the expected usage profile.

To conclude this introduction, we point out that the invocation of an external service corresponds to making a transition toward its *start* state and then reconnecting its *end* state to the next step after the invocation. This however leads to some modeling issues in the case of recursive invocation. DTMCs do not allow for recursive invocation, but require to unroll the sequence of calls to a finite depth. We identified two different solutions to this issue. The first consists in limiting the recursion to a depth where further invocations do not significantly affect the reliability estimation, that is, the probability of further invoking the service is low enough to make its impact on software reliability negligible (given the desired accuracy for the prediction). The second solution consists instead in adopting a superclass of DTMCs named Recursive Markov Chains (RMCs) [21]. These stochastic models extend DTMCs by allowing a state to be recursively connected to another RMC, thus they allow the explicit representation of recursion and provide a suitable mathematical framework to compute the probability of reaching the *success* state.

In the rest of this section we present the two reliability analysis tools supported by KlaperSuite. The first is based on the probabilistic model checker PRISM [35], while the second is based on an implementation of the algorithms to analyze RMCs described in [21].

4.1 The PRISM-based Tool

KlaperSuite is able to automatically transform a KLAPER model into a form compatible for consumption by PRISM. The input for PRISM consists of a DTMC

model and a property to be verified upon it. In our case, such a property represents the probability of reaching the *success* state. PRISM performs its computations by using iterative numerical algorithms [35]. These algorithms allow for computing the requested probability with a desired (finite) accuracy, i.e., the difference between the real probability and the results computed by PRISM will be lower than the given threshold. KlaperSuite requires by default a maximum error of 10^{-12}.

The translation of KLAPER models into PRISM inputs allows also for further analyses on the software behavior, besides the overall reliability on demand. PRISM in fact supports a wide range of properties to be verified apart from reliability (for example it is possible to compute the failure probability given that the process reached a certain state, or the probability of completing a run without passing through a certain state).

Finally, PRISM provides also process algebra constructs to simulate function invocations through the concept of *module*. Modules are DTMCs whose execution can be synchronized to simulate the unrolling of function call chains. Such unrolling is pursued iteratively until the analysis result converges to the desired accuracy. The drawback of function calls unrolling lies in the exponential state-space explosion. Our experience showed that even small KLAPER models may not be tractable through PRISM in presence of recursive invocations.

This issue motivated our research for a more efficient way to deal with recursion, namely Recursive Markov Chains (RMCs), which we present later in this Section.

4.1.1 The Transformation The automatic transformation from KLAPER to PRISM inputs is realized in two steps. The first step consists in a model-to-model transformation from KLAPER to an intermediate meta-model that reproduces the structure of a PRISM model. This transformation is implemented in QVT-Operational, the imperative model-to-model transformation language standardized by the OMG [33]. The second step is a model-to-text transformation implemented in Xpand2 [20], which generates the textual input files used by the PRISM tool for the analysis. We defer to Appendix A.1 and Appendix A.2 for details about the first and second step of this transformation, respectively.

4.2 The RMC-based Tool

As we mentioned before, a Recursive Markov Chain (RMC) can be seen as a collection of finite-state Markov Chains with the ability to invoke each other, possibly in a recursive way. They have been introduced in [21]. RMCs can be efficiently analyzed, by means of non-linear equation systems, to compute reachability properties. Thus the probability of reaching the success state can be formalized in this framework.

Referring to [21], KLAPER behaviors can be classified as *1-exit* control flows. *1-exit* control flows are control flows which have only one single *end* state, a property that allows the verification of any reachability property in P-time. Despite the theoretical worst case complexity, in most practical cases RMC analysis can be performed in reasonably short time (see Section 5). In KlaperSuite we support both the basic solution algorithm for RMC and the Newton variant, that exploit

the popular Newton method to reduce the number of iterations needed to reach the fixed-point solution. Both the algorithms are described in details in [21].

If time complexity is an advantage of RMCs, by contrast they are suitable only for reachability analysis and hence, within KlaperSuite, only for the evaluation of the overall reliability on demand. As a matter of fact, the analysis of more complex properties — such as the ones we mentioned for PRISM — cannot be, in general, easily verified with these models.

Compared to PRISM, the RMC-based analysis can handle very large models with recursive invocations. The accuracy of results is arbitrary also for this tool and set by KlaperSuite to the default 10^{-12} value.

4.2.1 The Transformation The first step of the transformation from KLAPER to RMC is the same model-to-model transformation we use for PRISM. From the intermediate PRISM-tailored model, KlaperSuite extrapolates a system of equations that is directly solved by our Java implementation of the iterative algorithms described in [21], without any need for external tools. We provide both the basic fixed-point algorithm and its optimization via the Newton method to speed up convergence (pros and cons of basic and Newton methods are discussed in [21]). We defer to Appendix A.1 and Appendix A.3 for details about the first and second step of this transformation, respectively.

5 Empirical Validation

In this section we describe the results of the empirical validation we conducted for the reliability analysis features of KlaperSuite. The validation comprises four case

studies — the first three have been extracted from the existing literature while the last one has been derived from an industrial system — and has been conducted as follows. The complete KLAPER models of each test case, and the corresponding analysis reports, can be downloaded from `http://home.dei.polimi.it/filieri/2011sosym`.

The three literature-based case studies have been extracted from [13,26] and from [74,41,46]. Section 5.1 describes them in detail and outlines the results of the empirical validation. For the first two case studies [13,26], the available literature does not provide complete design models, but only their formalization through DTMCs models. Our experiments have thus been conducted by reverse-engineering from those DTMC models the corresponding KLAPER models, and by proceeding according to the usage scenario *a* described in Section 2.2. For the third case study — the Business Reporting System (BRS) case study [74,41,46] — high level design models were instead available, we directly used them for the experiment, and we proceeded according to the usage scenario *b* described in Section 2.2. The purpose of evaluating KlaperSuite via these case studies is two-fold: first we intend to validate the mathematical infrastructure of KlaperSuite by comparing our predictions with the results described in the original literature, second we intend to verify the efficiency of the various reliability prediction tools embedded in KlaperSuite. With respect to the first goal, all the analyses of problems for literature matched the results provided in the original papers, up to the accuracy there reported.

Concerning efficiency and scalability, we have also conducted a extensive set of experiments where randomly generated large KLAPER models have been tested in order to stress KlaperSuite analysis features. The results of these experiments are reported in Section 5.3.

Finally, the industrial case study described in Section 5.2 has been instead conducted in collaboration with ABB and is based on the corresponding case study developed for the Q-ImPrESS project[4]. Q-ImPrESS provides a model-driven methodology and an Eclipse-based development environment supporting the design of complex software systems and the analysis of their reliability, performance and maintainability attributes at an architectural level. Q-ImPrESS proposes the adoption of the Service Architecture Meta-Model (SAMM) [59], a new abstract design notation to describe both the structure of the system with a component-based paradigm and the QoS of each constituent. QoS is then estimated by deriving prediction models from SAMM design models, and by using tools (such as KLAPER for reliability properties) to concretely perform the estimations. Since also KlaperSuite leverages the KLAPER intermediate language, it has been relatively easy to apply the same case study used for Q-ImPrESS also for our validation. However, in this case the aim of the experiments we performed is different from the literature-based case studies. Here we intend to show that KlaperSuite also scales in industrial settings and that the predictions computed by our framework are sound and compatible with real measurements.

[4] `http://www.q-impress.eu`

5.1 The Case Studies Extracted from Literature

Every case study described here has been modeled in KlaperSuite and analyzed with both the PRISM and the RMC based tools. For each case study we give results concerning both the space and time complexity of the resulting Markov model. For the former aspect, we give in Appendix A.1) details about the translation process from KLAPER to Markov model, that show that we should not expect large increase in the size of the obtained Markov model with respect to the original KLAPER model (our experiments confirm this point). For the latter aspect, the two solvers exhibit in general different performances, depending on the characteristics of the models being analyzed. For example, the way the modeled components are involved in the control flow (i.e., either by using recursion or by using loops) is an example of such characteristics. In particular, when recursion is heavily present in the KLAPER model, our experiments show that managing it with process algebra in PRISM could lead to poor performance if compared with the RMC-based analyzer.

Defining which characteristics of the control flow can be used to drive the selection of the analysis tool is still a matter of investigation. However, in Section 5.3 we describe the results we have obtained with our random models generator and we give some glimpses on the role of model topology and dimensions on the performance of the solution tool.

5.1.1 User-centered Reliability The work described in [13] is among the first works using DTMC to perform reliability analyses, and is the first aiming at for-

malizing the role of the users to characterize the behavior of software systems. The role of the users, i.e., a probabilistic profile of the incoming requests, has been modeled via KLAPER *Workload*s. The control flow of the application has been instead derived from the DTMC described in the original case study, which implicitly defines it. The DTMC process has been inferred statistically from the number of invocations from one component to another. With only this information, the control flow cannot be easily restated in a structured programming form — since DTMCs can represent also loops not directly expressible via structured programming — nor we can assume any recursion. As a matter of fact, for the case study we defined a KLAPER-based model leading to the DTMC described in the original work. The obtained KLAPER model consists of 1 resource, 1 service, 22 transitions, and 10 internal failure probabilities. The reliability obtained with KlaperSuite is 0.8299408117043016 versus the 0.8299 given in the paper.

Table 1 shows the results we obtained by running this case study. In particular, it shows and compares the performance of the two analyzers (PRISM and RMC).

Table 1 KlaperSuite analysis of [13]'s model.

Prism model states	20
Prism model transitions	46
Prism execution time	2574 ms
RMC number of equations	16
RMC execution time	1771 ms
RMC execution time with Newton method	27 ms

In this case the performance of the PRISM-based tools and of the basic implementation of the RMC-based tool have comparable performance (though PRISM takes more time because of the external process invocation). The use of Newton method to analyze the same problem leads to faster convergence of the algorithm with significant improvement in computation time.

5.1.2 The GCC Case Study In [26] the authors propose an empirical analysis of the Gnu Compiler Collection (GCC) C compiler version 3.2.3. The analysis framework used in [26] requires a component-based model of the software system being analyzed (similarly to KlaperSuite), and the model has been extracted from the source code. In the original work, the authors instrument the compiler with a profiler and run a large test suite. The information from the compiler has then been used to derive a component-based model of the software, to determine the software architecture, and to describe the failure behavior of each component. The original experiments identified 85 failures with a regression test executing the test suite from a more recent version of the compiler (3.3.3). As we did for the previous case study, a KLAPER-based model has been defined in order to lead to a DTMC that matches the one in [26].

The obtained KLAPER model consists of 1 resource, 1 service, 22 transitions, and 13 internal failure probabilities. The reliability obtained with KlaperSuite is 0.9997626025641249 versus the 0.9997 given in the paper.

Table 2 shows the results we obtained by running this case study. The RMC-based analysis has been interrupted after 20 minutes since, as explained in [21], a basic fixed-point solver may suffer of too slow convergence. Running the Newton

method on the same model instance provided instead the result in hundreds of milliseconds.

Table 2 KlaperSuite analysis of [26]'s model.

Prism model states	23
Prism model transitions	118
Prism execution time	2553 ms
RMC number of equations	19
RMC execution time	$> 20min$
RMC execution time with Newton method	101 ms

5.1.3 The BRS Case Study The last literature-based case study concerns the analysis of a Business Reporting System (BRS) with KlaperSuite. The BRS system has been used in several former papers [74,41,46] as a case study for performance and reliability prediction of component-based software systems.

The BRS is a web-based system, which provides managers with access to a number of key performance indicators of their enterprise. It allows monitoring the current values as well as generating detailed reports aggregating formerly recorded data. Figure 4 depicts the SAMM-based [59] model of the system with several selected *Behaviors*. The BRS consists of five components deployed on four servers and includes nine *Behavior* specifications in total. The internal failure probabilities for the steps in the behaviors have been estimated based on the complexity of the involved calculations and on experience with similar systems (see Section 6 for a discussion on this topic).

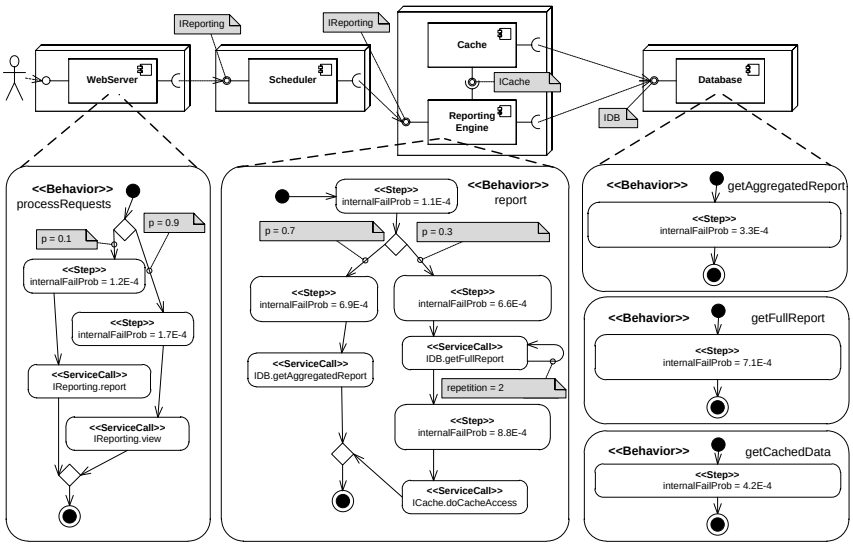


Fig. 4 The SAMM architectural model of the Business Reporting System (BRS).

The KLAPER model automatically generated from the SAMM model by KlaperSuite consists of 7 resources, 66 steps, 51 transitions, and 11 internal failure probabilities. For space reasons we do not illustrate it here. After the first transformation step, KlaperSuite transforms the intermediate model into a DTMC within half a second on a regular PC. The result is a DTMC model with 200 states, and 403 transitions.

KlaperSuite estimate the reliability of the BRS system in 0.9986512518492634, the corresponding estimation computed by the PCM toolchain is 0.99865, for the same version of the system.

Table 3 shows the computation time of reliability analysis with PRISM and RMC. Notice that in this case the execution time of the Newton solver for RMC is significantly longer than the execution with the basic algorithm. The BRS KLAPER model, despite the number of equations, presents a few loops in its control flow

(actually, only some loops with *Repetition* attribute equal to two). This allows to make the equation solver quickly converge to a fixed point. Indeed, in complete absence of loops, known values take at most as many iterations of the basic solver as the number of equations to be propagated, leading to the solution. Each iteration of the basic solver takes a short time to be accomplished. On the other hand, each iteration of the Newton solver requires extra time to compute the next step of the iterative algorithm. Such an extra time is not compensated by the reduction in the number of iterations, leading to the poor performance of the solver.

Table 3 KlaperSuite analysis of the BRS model.

Prism model states	200
Prism model transitions	403
Prism execution time	2762 ms
RMC number of equations	170
RMC execution time	32 ms
RMC execution time with Newton method	3507 ms

We also used KlaperSuite to perform a sensitivity analysis, i.e., the identification of the components mostly contributing to the system reliability [13]. We imposed small variations to single components internal failure probabilities and analyzed the corresponding variation on the overall reliability on demand of the system. We point out that we limit to these parameters the sensitivity analysis, as the uncertainty in the model mainly stems from them, rather than from other parameters like the transition probabilities. The latter indeed are part of the descrip-

tion of the model structure, which resembles the system structure and is assumed to be given and not subject to significant changes.

Figure 5 shows that the curves for *"AcceptView"* and *"PrepareViewing"* have the highest slopes, thus the system reliability is most sensitive to them. The system is less sensitive to the other actions, i.e., improving their reliability has only a minor effect on the overall system reliability on demand. The actions related to viewing have a higher impact, because of the much higher volume of invocations they receive.

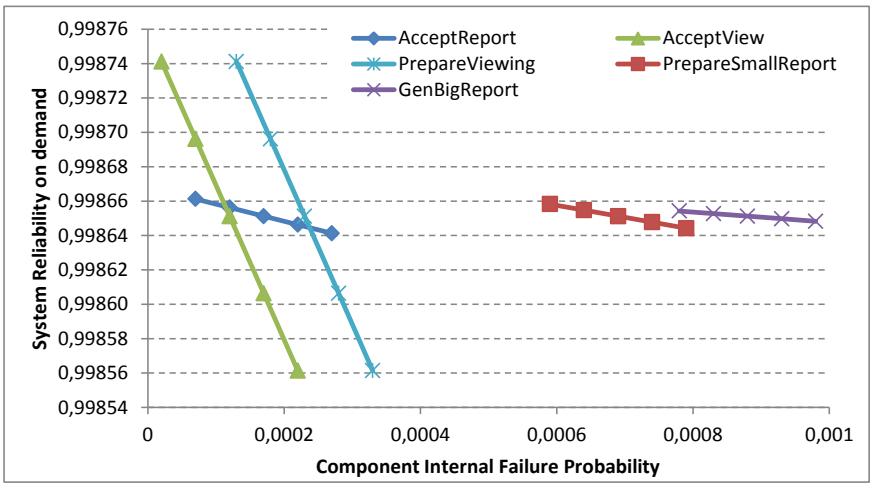


Fig. 5 Reliability Predictions for the Business Reporting System.

5.2 The Industrial-based ABB Case Study

This case study analyzes a large-scale industrial Process Control System (PCS) from ABB [42]. The goal here is to demonstrate that KlaperSuite can also deal with large and real models. Furthermore, we also want to show that our framework

is able to find the components mostly contributing to the system reliability so that effective improvement measures could be identified and applied.

The ABB PCS system is used in different industry branches, such as power generation, chemical processes, or material handling. It provides access to a number of sensors and actuators built into an industrial process for supervision. Human operators get a graphical visualization of the most important process values and can interact with the system (e.g., stopping pumps or opening valves).

Figure 6 shows a high-level overview of the system as modeled using the Q-ImPrESS tools. It consists of 28 components, which are deployed on three servers. There are more than 30 services modeled for this system, which we cannot show here for space reasons. The internal failure probabilities for the Steps of the services had been determined in a former case study [42] using software reliability growth models based on an analysis of the PCS' bug tracking system.

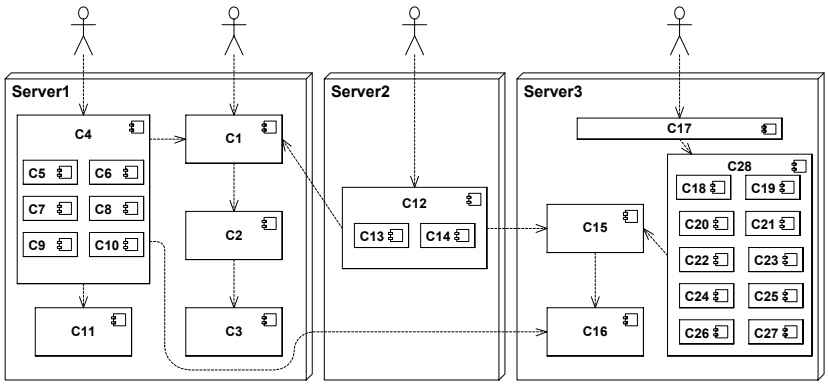


Fig. 6 ABB PCS system

The automatically generated KLAPER model of the system consists of 34 resources, 217 steps, 167 transitions, and 8 internal failure probabilities. We omit showing details of this model here for space reasons. KlaperSuite mapped the model to a DTMC model with 275 states, and 261 transitions. The execution time for the PCS case is reported in Table 4. Table 4 reports the computation time of

Table 4 KlaperSuite analysis of the ABB PCS model.

Prism model states	275
Prism model transitions	261
Prism execution time	NA
RMC number of equations	552
RMC execution time	118 ms
RMC execution time with Newton method	104933 ms

reliability analysis with PRISM and RMC. In this case PRISM fails to build the model for a memory space problem, while the execution time of the RMC solver with the Newton method takes quite a long time. This case is similar to BRS one: each iteration of the Newton method is very expensive. Although the number of iterations is much lower than the basic case, this does not suffice to compensate the cumulative cost of the Newton method extra computations.

We performed a sensitivity analysis of the model to identify the most critical services in the system. Figure 7 depicts how the system reliability changes if we change the failure probability of four exemplary steps. The figure leaves out

the concrete system reliability values as well as the actual names of the steps for confidentiality reasons.

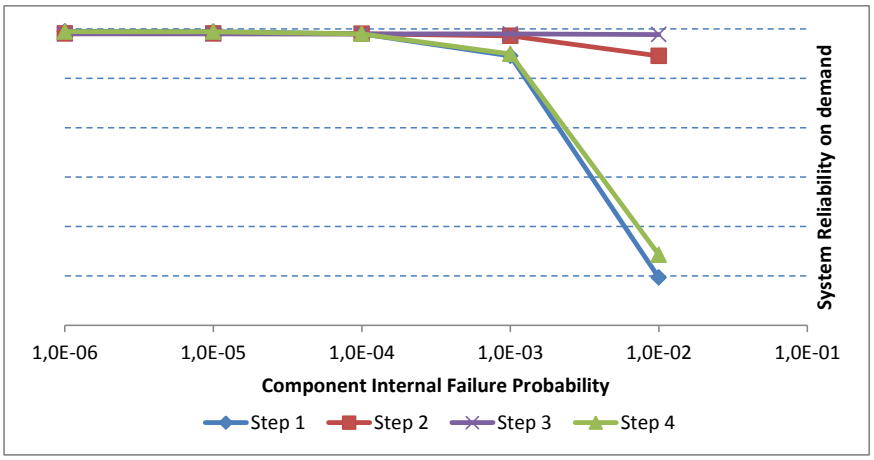


Fig. 7 Reliability Predictions for the ABB PCS

It can be seen that the steps 1 and 4 have the highest impact on the overall system reliability on demand, which considerably decreases with a decrease of their internal failure probabilities. Steps 2 and 3 only have a minor impact on the overall system reliability. Reliability improvement measures should then focus on the software piece responsible of steps 1 and 4, for example by conducting more testing or implementing fault tolerance mechanisms.

5.3 Scalability Tests

In this section we show the results of scalability tests conducted on the Klaper-Suite.

As already discussed in Section 4 and empirically shown in the industrial case study (cf. Section 5.2), PRISM is not suitable for the analysis of recursive models, but for simple cases where the unrollment of function calls quickly makes the approximation accurate enough. For this reason, we analyzed non recursive models comparing PRISM and RMC execution time. Due to the absence of recursive calls, RMC-Newton has been omitted from the comparison because its computational overhead is not justified in absence of recursive calls and its execution time exceeded by orders of magnitude that of the other solvers in our tests. The reader interested in an empirical comparison of the execution times of the RMC algorithm and its Newton variant can refer to [21].

All the experiments have been conducted on an Intel Pentium D 3 GHz with 2Gb of RAM, Debian 6.0.5 32bit.

All the KLAPER models have been randomly generated. The random generator allows to set the number of services, and, for each service behavior, the number of activities, the number of branches, and the number of service invocations. For each observation we took 10 samples and reported their average value.

In the first test-suite we analyzed a model containing a variable number of *small services*. Each service behavior is composed by 5 activities and 1 branch. Each service invokes 2 ± 1 other services.

Figure 8 shows the result of this comparison. On the x-axis we reported the total number of activities, that is the number of services (from 2 to 26) times 5 activities per service. Notice that the y-axis is in logarithmic scale to evidence the difference in order of magnitudes of the two solvers. Indeed, RMC-Base outper-

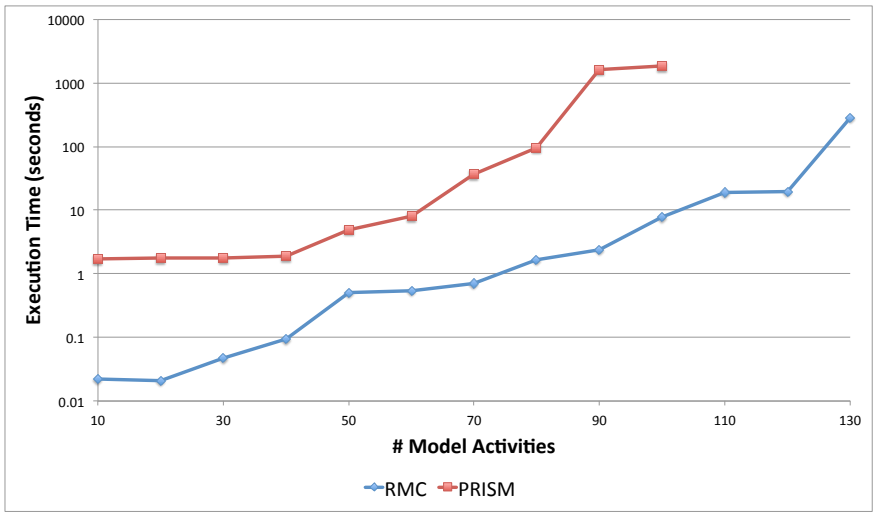


Fig. 8 Tools comparison: small-size services.

formed PRISM with a relative speed-up up to three order of magnitude for the largest models. For models with 22 services or more, PRISM took more than 2 hours and has been interrupted.

In the second test suite we considered medium-sized services, each containing from 25 to 45 activities. Each service behavior contains at least one branch node and 2 ± 1 invocations of external services. In Figure 9 we reported on the x-axis the total number of activities, obtained by multiplying the number of services (from 10 to 12) times the number of activities per service (from 25 to 45).

The two solvers showed in this case a comparable performance for the largest instances, but PRISM outperformed RMC for smaller cases.

From Figures 8 and 9 we can deduce some information about the factors that mostly affect the scalability of the two tools, namely number of services and number of activities per service.

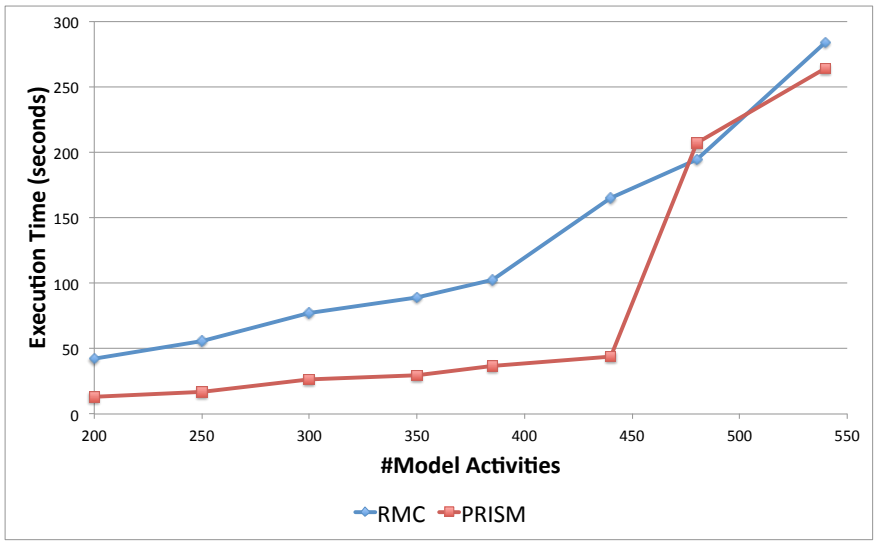


Fig. 9 Tools comparison: medium-size services

For PRISM analysis, each service is mapped, by the model transformation procedure, to a module (cf. Section 4.1). Before starting the actual analysis, the model-checker needs to build the actual state space of the model, that is the cartesian product of the local state space of each module. This phase is time consuming and mostly depends on the number of services. For this reason in Figure 8 PRISM execution time grows so quickly. The quick growth in RMC performance is instead due to the number of service invocations. Indeed, each of these introduces a non-linear equation in the system to be solved, slowing down the convergence [21]. Since there are 2 ± 1 invocations per service, the number of non linear equations grows linearly with the number of services in this setting.

On the other hand, the size of the different behaviors (i.e. the number of activities they are composed by) has a smaller impact on the execution time of both the solvers, if compared to the number of services. In Figure 9, the performance

of the two tools is comparable when the number of services is small, and coherent with the previous experiment. Indeed, PRISM execution time increases significantly when more than 10 services are introduced, because of the impact of the state space building. For a smaller number of services, PRISM state construction and the number of non linearities in the equation system of RMC play a minor role; in this setting, most of the time of both PRISM and RMC is spent in solving the respective systems of equation. This make the optimized numerical routines of PRISM outperform our Java implementation of the basic solution algorithm for RMC [21]. On the other hand, the solver used by PRISM is defined for linear systems only, while RMC, though slower on linear ones, can deal also with the non linearities introduced in case of recursive calls [21].

Further evidence of this conjecture comes from Figure 10, where only 2 services have been defined with a number of activities from 25 to 100 for each of the two. In this settings the impact of model construction for PRISM is negligible with respect to the time required by the numerical procedures, making PRISM more efficient than RMC.

5.4 Summary of the Validation

In this section we modeled and analyzed with KlaperSuite a set of case studies extracted from literature, from an industrial setting and from randomly generated models. The validation showed that KlaperSuite is able to deal with the complexity of each case study, although the adopted prediction tools performed differently.

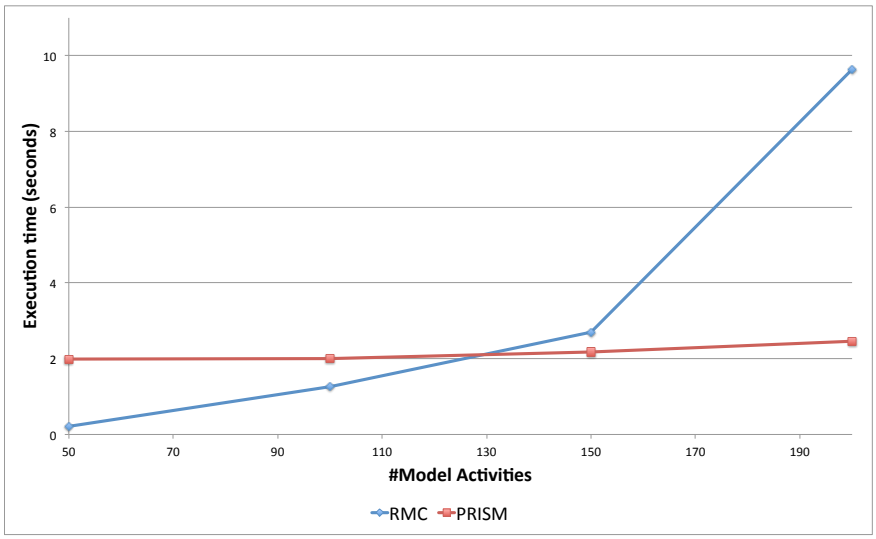


Fig. 10 Tools comparison: large-size services

Indeed, the obtained results show that for each specific scenario a tool may outperform the others.

The three solvers introduced in the previous sections differ for the amount of information provided and the suitability for specific problems.

In general, the use of PRISM provides, besides the estimation of the expected system reliability, the complete PRISM input. Such an input could be a good source of information for expert users that could possibly run further analyses of their systems. Indeed, apart from computing the overall reliability, the model-checker can be employed for finer grain analysis, e.g. for computing the probability of hitting a failure after a certain operation has been performed. Access to this finer grain analysis is out of the scope of KlaperSuite and requires some mathematical skills from the user. On the other hand, the generated PRISM models are not tailored to KlaperSuite analysis and embed all the available information about the

artifact. Hence, KlaperSuite can be exploited also to translate KLAPER models to PRISM for further investigation.

If the KLAPER models do not contain any recursive function call, the use of PRISM or RMC provided comparable performance on our low-power machine if the number of services is between 10 and 20. The gap between PRISM and RMC appears more evident in case the model contains a few large services. In such a case, the optimized algorithms of PRISM outperform our Java based solver for RMC. A currently unexploited alternative is to use an external tool, such as Matlab, to solve the set of equations generated for RMC. This could provide significant improvements thanks to optimized algorithms, topology based heuristics, and out-of-the-shelf parallelization capabilities.

In presence of recursive calls, PRISM tries to unroll the recursive calls until the required analysis accuracy is met. Such a process, besides being time-consuming, may result in out of memory exceptions due to the need to store the large state space derived from the unrolling. In such cases, a RMC approach is recommended.

The choice between RMC base and the Newton variant has to be considered as a trade-off. Indeed, the base implementation requires more iterations to converge, but each single iteration is extremely efficient, being no more than the evaluation of n arithmetic expressions (where n is the number of derived equations). On the other hand, the Newton variant requires less iterations, but each of them is time consuming, since it requires to evaluate n^2 derivatives. The effectiveness of the Newton method is significant when the chain of recursive calls can become quite long. In all the other cases the effort required in each iteration overcomes the ben-

efit of performing a smaller number of them. Empirical evidence of this claim has been provided in Section 5.3; a theoretical complexity analysis of the two variants can instead be found in [21].

As a final remark, from the industrial case study of Section 5.2 we would like to underline that design models, being abstraction of the implemented system, may capture relevant architectural aspects in relatively small KLAPER instances. Indeed, the large scale Process Control System from ABB, resulted in 28 components and 30 services, for a total of about 250 activities. The model presents also an high level of interdependency among the components, captured by several recursive invocations. Though the analysis is not feasible with the PRISM-based solver, such a complexity is within the computation capabilities of the RMC solvers included in the current version of KlaperSuite.

6 Related Work

In the last years, the need for including early quality prediction in the software development process has been widely recognized. In particular, there has been an increasing interest in model transformation methodologies for the generation of analysis-oriented target models (including performance and reliability models) starting from design-oriented source models, possibly augmented with suitable annotations. Several proposals have been presented concerning the generation of performance or reliability analysis models, some of them suggest a *direct* model generation while others suggest the use of *intermediate* models. In this section we briefly summarize these approaches and some other topics closely related to these

transformations, such as the already cited *feedback provisioning* step and the open problem of the *model parameter estimation*.

Direct Reliability Prediction Methods Reliability prediction methods for software architectures have been surveyed in several papers [29,24,37]. Almost all of these approaches use a DTMC or CTMC model to describe the control flow between components and respective solver tools to conduct reliability predictions. Some approaches require the software architect to directly work with the Markov-model notation (e.g., [64,66,72,25]), which might discourage practitioners because of the semantical gap to the architectural models they commonly use (e.g., in UML).

Therefore, several approaches (e.g., [16,27,76,62,58]) have proposed the use of a high-level modeling notation (e.g., UML sequence and deployment diagrams), annotated with the necessary reliability data (e.g., component transition probabilities and internal failure probabilities). Transformation tools map these high-level models into Markov chains, so that standard solvers can execute reliability predictions with the transformation output. The benefit is that developers can reuse their existing UML model and only need to provide additional reliability annotations. Furthermore, the mathematical details of the prediction methods can be encapsulated into tools and thus be transparent to the developer.

This class of approaches can be further divided into scenario-based approaches (e.g., [62,76,58]) and UML-based approaches [16,27].

From the scenario-based approaches, Yacoub et al. [76] manually create component dependency graphs out of sequence diagrams, which are then processed by tools. Rodrigues et al. [62] sketch a transformation from message sequence charts to DTMCs and also propose an implementation [63]. However, the tools for both approaches are not publicly available. Popic et al. [58] extend the ECRA tool for reliability analysis, so that it accepts UML use case, sequence, and deployment diagrams.

From the UML-based approaches Cortellessa et al. [16] propose a mapping from UML diagrams into Markov models, but also provide no tool support. Goseva et al. [27] use UML sequence diagrams and Marko models in their approach and mention that the implementation of a tool would be straightforward. However, up to today such a tool has not been provided.

In contrast to other approaches, KlaperSuite is fully implemented and available for third-party testing. Moreover. the approaches described above focus exclusively on reliability prediction, whereas KLAPER also support performance predictions.

Reliability Prediction through Intermediate Models The large gap between design-oriented and analysis-oriented models can make direct transformations like the ones summarized above quite complicated. A different way to deal with transformation complexity is to pass through an *intermediate* model (the *"kernel"*) by pruning the information from the design model that is not needed to execute the desired analyses, but still retaining the needed one. One of the first proposals along

this direction (albeit in a different context from non-functional requirements analysis of component-based systems) can be found in [39], where the kernel language is called a ”pivot metamodel”.

Among the transformation approaches that make use of intermediate models, Petriu et al. [57] proposed the CSM (Core Scenario Model). CSM is a MOF compliant kernel meta-model, specifically related to performance analysis. Transformation from UML to CSM and from it to different performance models are provided. Gu et al. [34] proposed, in a similar way, their own intermediate meta-model to transform UML model with performance annotations to performance modeling formalisms.

With respect to the kernel languages of [34, 57], KLAPER is intended to serve also for reliability and, possibly, trade-off analysis between performance and reliability. KLAPER is specifically targeted to component-based systems and it has been applied for the analysis of performance and reliability using Queuing Networks and Markov Models [32] and experienced with the CoCoME case study [61, 31]. Extensions of KLAPER have also been proposed to analyze self-adaptive [30] and reactive [55] systems. In these works the KLAPER models have been designed manually, without using any automated transformation tool.

Feedback Provisioning As pointed out is Section 2.1, besides considering the linking from design-oriented to quality-oriented models, also the opposite direction should be considered, to give automated support to bring analysis results back to design models. This research area has received some attention in the last years and is called *feedback provisioning*. Its goal is to give support to possibly non-

experienced engineers and guide them in the selection of an appropriate design solution when issues concerning quality attributes are detected by means of analysis tools.

The kind of feedback to provide and the way to provide it depend on the adopted methodology, and some (partly automated) approaches have been already proposed in literature. Examples are

- *rule-based* approaches: they rely on a set of domain specific predefined rules to identify potential quality-related problems and to suggest modifications to the system models. These approaches, however, present several drawbacks: human intervention is required, every approach defines its own language to specify rules, and rules propose solutions only for simple issues and at the level of quality prediction models (i.e., manual intervention is required to translate the suggested changes to the abstraction level of design models) [54, 75, 15, 48].
- *meta-heuristic* approaches: they leverage specific algorithms to explore the alternatives space and to propose different complete system solutions satisfying certain quality criteria. The algorithms used by these approaches, however, limit the set of supported quality attributes, the set of supported exploration directions, and are tailored to specific modeling environments [47, 11, 3].
- *Design Space Exploration (DSE)* frameworks: they work similarly to meta-heuristic approaches, but the alternatives space is explored by encoding the problem as a Constraint Satisfaction Problem (CSP). Although DSE approaches are extremely efficient, they suffer from the same kind of problems outlined for meta-heuristic techniques [65, 38, 51].

– *model-driven* approaches: they rely on the possibility of exploring different design alternatives and feeding back the results to software designers exploiting the capabilities of quality driven model transformations [19,50].

Model Parameter estimation Another open issue in the field of quality-driven model generation concerns the performance and reliability model parameters estimation [67,24,12,42]. Several methods have been defined and applied. They are mainly based on estimations derived from measurements on the running software or on estimations derived from similar applications or finally from educated guesses based on the experience of the software engineers [12,67]. In particular, in [42] different data collection methods for the estimation of failure probabilities are described. They can be based on the use of code metrics (such as lines of code), reliability growth models [44], fault injection techniques [26] and/or statistical testing [49]. Each of them presents pros and cons and the problem of failure parameter estimation is still a matter of debate and investigation.

Automated environments Despite the existence of several methods (both direct and based on intermediate models) that apply automated transformations to generate analysis-oriented models, still few integrated environments exist that include a family of tools empowering designers with the ability to capture and analyze the performance and reliability figures of their systems. PUMA [73], for example, provides a Model-Driven Engineering framework that adopts CSM as intermediate language to predict performance via Layered Queuing Networks or Stochastic Petri Nets starting from UML models augmented with MARTE profile compliant

annotations. Another notable example of a model-driven engineering framework to support the development of analysis transformations for non-functional properties is given by Palladio-Bench [1]. This framework is centered around Palladio, a newly developed component model [8], and integrates modeling, simulation/analysis, and result viewing in a single software tool. Both performance and reliability analysis are supported and can be analyzed based on the same Palladio model.

KlaperSuite lies in this research area and provides a fully automated and integrated environment including a family of tools empowering designers with the ability to capture and analyze the performance and reliability figures of their systems. The possibility of using different verification tools together with a simulation-based analysis tool could make KlaperSuite a valuable instrument for predicting software qualities during the development process.

7 Conclusions

In this article we presented KlaperSuite, a MDE-enabled toolchain to support analysis of non-functional attributes for component-based system since early stages of development. KlaperSuite reduces the gap between design and analysis models exploiting the pivot language KLAPER, that allows to represent both common design concepts and quality annotations in a unified model.

KlaperSuite includes a set of automatic model transformations from KLAPER to stochastic analysis models, that can be solved within the suite by means of es-

tablished solvers, such as the probabilistic model-checker PRISM. The results of analysis are captured by the suite and presented to the designer in a fully transparent way.

The main benefit of KlaperSuite is to enable the access to a comprehensive QoS analysis suite by implementing a single transformation from the preferred design model toward KLAPER. Hence, designers are empowered with the ability to analyze their systems, with established analysis instruments, in a seamless and integrated environment, without the burden to deal with each single instrument by hand.

In this article we specifically focused on the validation of the reliability analysis features of KlaperSuite and we presented an in-depth description of how the analyses are performed. To show the effectiveness of KlaperSuite, we validated our approach by using a set of literature-based and industrial case studies. The industrial case came from the model of a large scale process control system and showed the ability of the reliability analyzers to deal with real-life problems. Furthermore, we analyzed a set of randomly generated problems to show the scalability of the suite and to compare different analysis backends.

We are currently working on extending KlaperSuite to deal with more QoS properties. Specifically, we are integrating LQN-based verification for time performance analysis and a simulator for KLAPER models, based on *SimJava*. We are also planning to implement model transformations from higher level design languages (first of all certain subsets of UML) to KLAPER, for the sake of making KlaperSuite easier to integrate in established development settings.

A The KLAPER transformations

This appendix describes all the automatic model transformations defined in KlaperSuite to generate a reliability analysis model starting from a KLAPER model. First the model transformations are introduced, explaining their contexts and relations, then each transformation is detailed in a separate section.

As we pointed out in Section 4, KlaperSuite supports two *back-end*s to carry out reliability analyses: one relies on stochastic model checking and PRISM, the other is based on the RMC methodology. Each analysis *back-end* has its own reliability model, but their generation has in common a first model transformation that maps an input KLAPER model to an output Markov Model (MM). KlaperSuite then provides two model transformations from the intermediate MM to the *back-end* specific model that will be used to perform the analysis. This two-stages transformation approach is illustrated in Figure 11.

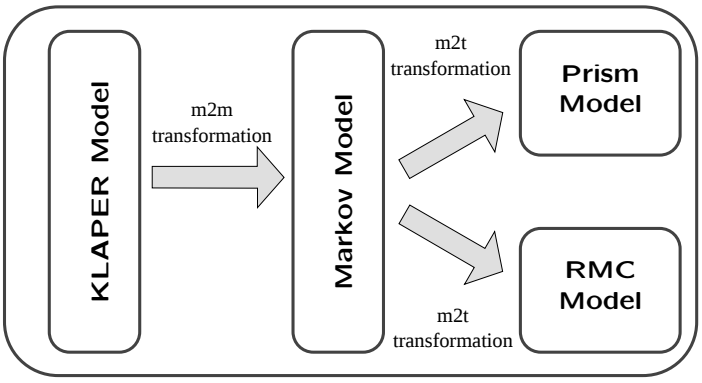


Fig. 11 The KlaperSuite transformations.

When PRISM is used to perform the analysis, a model-to-text transformation takes care of generating a textual representation of the Markov Model compatible with the PRISM model checker input format; in case of RMC, starting from the Markov Model, a system of equations — whose solution provides the reliability predictions for the system — is extrapolated.

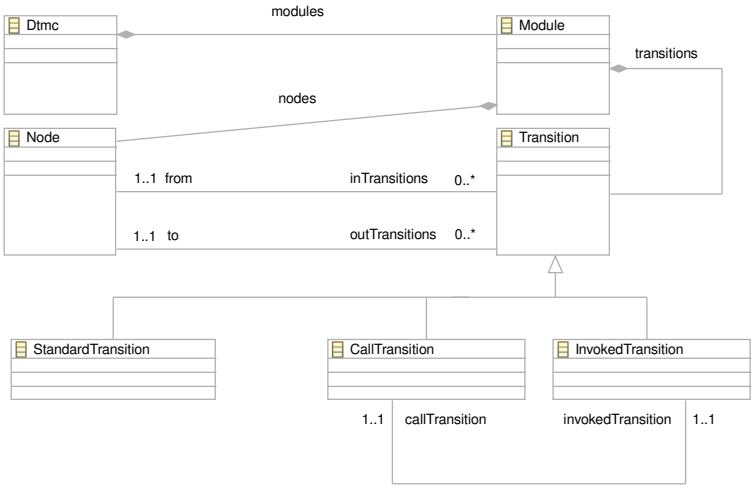


Fig. 12 The Markov Model meta-model.

Figure 12 outlines the structure of the meta-model which we use to represent intermediate Markov Models during the whole transformation process. This meta-model contains the concepts that may be found into Discrete-Time Markov Chains, with some tweaks to take advantage of some PRISM features such as *Module*s. In practice, we represent the different behaviors of a system via *Module*s; each *Module* is then internally described by means of *Node*s and *Transition*s.

A *Node* has several attributes defining characteristics such as its type (*basic*, *start*, *end*, and *failure*). *Start* nodes represent the initial state of a *Module*, *end* nodes and *failure* nodes represent instead standard stop and failure stop states for a *Module*, respectively. *Basic* nodes are used for all the other nodes in Markov Models. Each kind of *Node* must satisfy a well-defined set of validity constraints. *Basic* nodes must have at least one inbound transition and one outbound transition. *Start* nodes must have an outbound transition and no inbound transitions. *End* nodes and *failure* nodes must have at least one inbound transition without any outbound transition. In practice, a *Module* is a directed graph with exactly one *start* node and one *end* node. This can be granted by checking that, for every *Module* and for every *Node*, except for *failure* nodes, a path exists between the *start* node and the considered node and between the considered node and the *end* node.

Finally, we distinguish among two kinds of transitions: *Standard* transitions and *Call* transitions. In both cases, a constraint specifying that the connected *Node*s belong to the same module must be satisfied. *Call* transitions must in addition specify the module that will be *called* when the transition fires.

In the next sections we detail the model transformations used by KlaperSuite. Section A.1 describes the transformation from KLAPER to MM, while Sections A.2 and A.3 present the MM to PRISM and the MM to RMC transformations, respectively.

A.1 From KLAPER to Markov Models

The model-to-model transformation from KLAPER to Markov Models is implemented in QVT-Operational, the imperative model-to-model transformation language standardized by the OMG [33]. Briefly, a QVT-Operational transformation is composed by a set of functions that define operationally how a set of source elements is mapped to a set of target elements.

In this transformation, we map each KLAPER *Service* to a MM *Module*, while each KLAPER *Behavior* is mapped to a set of MM *Node*s and *Transition*s, representing the *Step*s and the *Transition*s contained in the *Behavior*. KLAPER *Workload*s are mapped in the same manner, but in this case the *Node*s and *Transition*s correspond to the *Workload Behavior*.

A mapping function is defined for each type of KLAPER *Step*. For some *Step*s (*Start*, *End*, *Branch*, *Join*, and *ServiceControl*) a corresponding MM *Node* exists, they are thus mapped one-to-one and connected according to the *Transition*s in the KLAPER model. Other *Step*s such as *Activit*ies and *Fork*s require a more articulated mapping, as explained in the following.

A KLAPER *Activity* step could define a nested behavior — describing its internal behavior — and a *repetition* attribute — indicating the number of times the activity will be repeated in case of failure —. For this reason, in general, this step is mapped to a set of MM *Node*s and *Transition*s. More precisely, the *Activity* internal behavior is mapped to a set of MM *Node*s and *Transition*s according to its structure, and this mapping is replicated as many times as it is indicated by the value of the *Repetition* attribute. This leads to a linear increase (proportional to the

Repetition value) of the number of MM nodes with respect to corresponding step in the original KLAPER model. In this respect, we also note that, as remarked in Section 2.3, besides using the *Repetition* attribute to model loops, KLAPER also supports the probabilistic modeling of guard-controlled loops, by a suitable use of the *Branch* step. This kind of modeling naturally matches cyclic structures of MM nodes, and thus leads to no increase in the number of MM nodes with respect to corresponding steps in the original KLAPER model.

Fork is the other kind of step requiring a special handling. A model may include several *Fork* control steps from which different execution flows are spawned that are then executed in parallel until a *Join* element is reached. The semantics is similar to the semantics of the corresponding concepts for UML. From the perspective of reliability, the global reliability of a *Fork-Join* block corresponds to the probability that all the actions performed in the parallel execution flows do not experience any failure. More formally, we may define the global *Fork-Join* block reliability as $\prod_j r_j$, where r_j is the reliability of the j-th spawned flow. As already said in Section 4, this formula holds under the assumption of independent failures among spawned flows. This formula can be represented in MMs by serializing the parallel execution flows, i.e., *Fork-Join* blocks are transformed by mapping each action execution in the spawned execution flows in a sequence of *Node*s and *Transition*s, which are then chained together in a possible sequence.

In general, a KLAPER *Transition* is mapped to the corresponding MM *StandardTransition*, except for outbound *Transition*s from a *ServiceControl* step. If

this is the case, KLAPER *Transition*s are mapped to *CallTransition*s linked to the called *Module*.

A.2 From Markov Models to PRISM files

This transformation is a model-to-text transformation implemented in Xpand2, a template language created by openArchitectureware [70], which is integrated in Eclipse Modeling Project. This transformation generates the textual representation of the PRISM model to be provided as input to the model checker. It produces two files, the PRISM model file and the PRISM properties file. The generation of the PRISM model file is straightforward. It contains the same information of the intermediate model in a textual representation. The PRISM properties file contains instead the definition of the system reliability properties which will be checked by PRISM in the form of the PCTL* expressions, as detailed in section 4.1.

PRISM is based on the Reactive Modules formalism described by Alur et al. in [4]. As a consequence, commands to regulate the state transitions among modules must be defined. In detail, each *Module* in the MM representation is transformed into the corresponding PRISM concept, and suitable commands to specify how the modules interleave are generated. For example, in the case of *CallTransition*s, suitable commands are generated to synchronize the interleaving between the caller module and the called module.

A.3 From Markov Models to RMCs

The model-to-text transformation from MMs to RMC is implemented in Java and generates a system of equations to compute the expected reliability. The system reliability can be calculated with arbitrary precision from the generated set of equations, as detailed in Section 4.2.

In the transformation, a MM *Node* is transformed into an equation defining the *Node* reliability, which in turn depends on the reliability of the connected *outTransition*s and of the related *Node*s. In particular, the reliability of the i-th *Node* is defined as $\sum_j p_{ij} * r_j$, where r_j is the reliability of the j-th *Node* and p_{ij} is the *outTransition* probability of the transition connecting the j-th *Node* to the i-th *Node*. The equation of the ending *Node*s is fixed to have reliability equal to one, while the equation of failure *Node*s is instead fixed to have reliability equal to zero.

References

1. Palladio: The software architecture simulator. Project website `http://www.palladio-simulator.com/`.
2. *WOSP : Proceedings of the International Workshop on Software and Performance*. ACM Press, 1998-2010.
3. Aldeida Aleti, Stefan Bjornander, Lars Grunske, and Indika Meedeniya. Archeopterix: An extendable tool for architecture optimization of aadl models. In *MOMPES*. IEEE, 2009.
4. Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

5. Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.
6. Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.
7. Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based performance prediction with the palladio component model. In *WOSP*, pages 54–65. ACM, 2007.
8. Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.
9. Jean Bézivin, Alfonso Pierantonio, Antonio Vallecillo, and Jeff Gray. Guest editorial to the special section on model transformation. *Software and System Modeling*, 8(3):303–304, 2009.
10. Tomas Bures, Jan Carlson, Ivica Crnkovic, Séverine Sentilles, and Aneta Vulgarakis. Procom - the progress component model reference manual, version 1.0. Technical Report MHD-MRTC-230/2008-1-SE, Malardalen University, June 2008.
11. Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *GECCO*. ACM, 2005.
12. Leslie Cheung, Roshanak Roshandel, Nenad Medvidovic, and Leana Golubchik. Early prediction of software component reliability. In *ICSE*, pages 111–120. ACM, 2008.
13. Roger C. Cheung. A user-oriented software reliability model. *IEEE Transactions on Software Engineering*, 6(2):118–125, 1980.
14. Andrea Ciancone, Antonio Filieri, Mauro Luigi Drago, Raffaela Mirandola, and Vincenzo Grassi. Klapersuite: An integrated model-driven environment for reliability and

performance analysis of component-based systems. In *TOOLS*, volume 6705 of *LNCS*, pages 99–114. Springer, 2011.

15. Vittorio Cortellessa, Anne Martens, Ralf Reussner, and Catia Trubiani. A process to effectively identify "guilty" performance antipatterns. In *FASE*, 2010.
16. Vittorio Cortellessa, Harshinder Singh, and Bojan Cukic. Early reliability assessment of uml based software models. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 302–309, New York, NY, USA, 2002. ACM.
17. Ivica Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., 2002.
18. Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.
19. Mauro Luigi Drago, Carlo Ghezzi, and Raffaela Mirandola. Towards quality driven exploration of model transformation spaces. In *MoDELS*, pages 2–16. ACM, 2011.
20. S. Efftinge and C. Kadura. Xpand language reference, 2006.
21. Kousha Etessami and Mihalis Yannakakis. Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. In *STACS*, volume 3404 of *LNCS*, pages 340–352. Springer, 2005.
22. Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Aspects of Computing*, pages 1–24.
23. Robert B. France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE*, pages 37–54. IEEE Computer Society, 2007.
24. Swapna S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE Transactions on Dependable Secure Computing*, 4(1):32–40, 2007.

25. Swapna S. Gokhale and Kishor S. Trivedi. Reliability prediction and sensitivity analysis based on software architecture. In *Proc. International Symposium on Software Reliability Engineering (ISSRE'02)*, pages 64–78. IEEE Computer Society, 2002.

26. Katerina Goseva-Popstojanova, Margaret Hamill, and Ranganath Perugupalli. Large empirical case study of architecture-based software reliability. In *ISSRE*, pages 43–52. IEEE Computer Society, 2005.

27. Katerina Goseva-Popstojanova, Ahmed Hassan, Ajith Guedem, Walid Abdelmoez, Diaa Eldin M. Nassar, Hany Ammar, and Ali Mili. Architectural-level risk analysis using uml. *IEEE Trans. on Softw. Eng.*, 29(10):946–960, October 2003.

28. Katerina Goseva-Popstojanova and Kishor S Trivedi. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2-3):179–204, 2001.

29. Katerina Goseva-Popstojanova and Kishor S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Perf. Eval.*, 45(2-3):179–204, 2001.

30. Vincenzo Grassi, Raffaela Mirandola, and Enrico Randazzo. Model-driven assessment of qos-aware self-adaptation. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 201–222. Springer, 2009.

31. Vincenzo Grassi, Raffaela Mirandola, Enrico Randazzo, and Antonino Sabetta. Klaper: An intermediate language for model-driven predictive analysis of performance and reliability. In *CoCoME*, volume 5153 of *LNCS*, pages 327–356, 2007.

32. Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *Journal of Systems and Software*, 80(4):528–558, 2007.

33. Object Management Group. Qvt 1.0 specification. `http://www.omg.org/spec/QVT/1.0/`, 2008.

34. Gordon P. Gu and Dorina C. Petriu. From uml to lqn by xml algebra-based model transformations. In *WOSP*, pages 99–110. ACM, 2005.

35. Andrew Hinton, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Prism: A tool for automatic verification of probabilistic systems. In *TACAS*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

36. Joseph R. Horgan and Aditya P. Mathur. *Handbook of software reliability engineering*, chapter Software testing and reliability, pages 531–565. McGraw-Hill, 1996.

37. Anne Immonen and Eila Niemelä. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7:49–65, 2008.

38. Ethan K. Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and Thomas Santen. Components, platforms and possibilities: Towards generic automation for mda. In *EMSOFT*. ACM, 2010.

39. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

40. Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, 2010.

41. Heiko Koziolek and Ralf Reussner. A model-transformation from the palladio component model to layered queueing networks. In *SIPEW*, volume 5119 of *LNCS*, pages 58–78. Springer, 2008.

42. Heiko Koziolek, Bastian Schlich, and Carlos Bilich. A large-scale industrial case study on architecture-based software reliability analysis. In *ISSRE*, pages 279–288. IEEE Computer Society, 2010.

43. Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice Hall, 1984.

44. Michael R. Lyu, editor. *Handbook of software reliability engineering*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

45. Michael R. Lyu. Software reliability engineering: A roadmap. In *FOSE*, pages 153–170, 2007.

46. Anne Martens, Heiko Koziolek, Steffen Becker, and Ralf Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *WOSP/SIPEW*, pages 105–116. ACM Press, 2010.

47. Anne Martens, Heiko Koziolek, Steffen Becker, and Ralf Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *WOSP/SIPEW*. ACM, 2010.

48. J. D. McGregor, F. Bachmann, L. Bass, P. Bianco, and M. Klein. Using arche in the classroom: One experience. Technical Report SEI-2007-TN-001, CMU, 2007.

49. Keith W. Miller, Larry J. Morell, Robert E. Noonan, Stephen K. Park, David M. Nicol, Branson W. Murrill, and Jeffrey M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Trans. Software Eng.*, 18(1):33–43, 1992.

50. Raffaela Mirandola and Catia Trubiani. A deep investigation for qos-based feedback at design time and runtime. In *ICECCS*, pages 2–16. IEEE, 2011.

51. Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-based design-space exploration and model synthesis. In *EMSOFT*. Springer, 2003.

52. Object Management Group (OMG). System modeling language 1.2. `http://www.omgsysml.org/`, 2010.

53. Object Management Group (OMG). Unified modeling language (uml) 2.3, superstructure. `http://www.omg.org/spec/UML/2.3`, 2010.

54. Trevor Parsons. A framework for detecting performance design and deployment antipatterns in component based enterprise systems. In *DSM*. ACM, 2005.

55. Diego Perez-Palacin, Raffaela Mirandola, José Merseguer, and Vincenzo Grassi. Qos-based model driven assessment of adaptive reactive systems. In *ICST Workshops*, pages 299–308. IEEE Computer Society, 2010.

56. Carl A. Petri. *Kommunikation mit Automaten.* PhD thesis, University of Bonn, 1962.

57. Dorin B. Petriu and C. Murray Woodside. An intermediate metamodel with scenarios and resources for generating performance models from uml designs. *Software and System Modeling*, 6(2):163–184, 2007.

58. Petar Popic, Dejan Desovski, Walid Abdelmoez, and Bojan Cukic. Error propagation in the reliability analysis of component based systems. In *Proc. 16th IEEE Int. Symp. on Software Reliability Engineering (ISSRE'05)*, pages 53–62, Washington, DC, USA, 2005. IEEE Computer Society.

59. Q-ImPrESS Consortium. The Q-ImPrESS project. Project website `http://www.q-impress.eu`, 2010.

60. Enrico Randazzo. *A Model-Based Approach to Performance and Reliability Prediction*. PhD thesis, Universitá degli Studi di Roma - Tor Vergata, 2010.

61. Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models [result from the Dagstuhl research seminar for CoCoME, August 1-3, 2007]*, volume 5153 of *LNCS*. Springer, 2008.

62. Genaína Nunes Rodrigues, David S. Rosenblum, and Sebastián Uchitel. Using scenarios to predict the reliability of concurrent component-based software systems. In *FASE*, volume 3442 of *LNCS*, pages 111–126. Springer, 2005.

63. Genaína Nunes Rodrigues, David S. Rosenblum, and Jonas Wolf. Reliability analysis of concurrent systems using LTSA. In *ICSE*, pages 63–64. IEEE Computer Society, 2007.

64. N. Sato and K. S. Trivedi. Accurate and efficient stochastic reliability analysis of composite services using their compact markov reward model representations. In *Proc. IEEE Int. Conf. on Services Computing (SCC'07)*, pages 114–121. IEEE Computer Society, 2007.

65. Tripti Saxena and Gabor Karsai. Mde-based approach for generalizing design space exploration. In *MoDELS*. Springer, 2010.

66. Vibhu Saujanya Sharma and Kishor S. Trivedi. Quantifying software performance, reliability and security: An architecture-based approach. *Journal on Systems and Software*, 80:493–509, August 2007.

67. Connie U. Smith and Lloyd G. Williams. *Performance and Scalability of Distributed Software Architectures: an SPE Approach*. Addison Wesley, 2002.

68. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

69. The Eclipse Foundation. Eclipse. Project website `http://www.eclipse.org`, 2010.

70. The openArchitectureware Consortium. openArchitectureware. Project website `http://www.openarchitectureware.org/`.

71. Wen-Li Wang, Dai Pan, and Mei-Hwa Chen. Architecture-based software reliability modeling. *Journal of Systems and Software*, 79(1):132–146, 2006.

72. Wen-Li Wang, Dai Pan, and Mei-Hwa Chen. Architecture-based software reliability modeling. *Journal of Systems and Software*, 79(1):132–146, January 2006.

73. Murray Woodside, Dorina C. Petriu, Dorin B. Petriu, Hui Shen, Toqeer Israr, and Jose Merseguer. Performance by unified model analysis (PUMA). In *WOSP*, pages 1–12. ACM Press, 2005.

74. Xiuping Wu and Murray Woodside. Performance modeling from software components. In *WOSP*, pages 290–301. ACM Press, 2004.

75. Jing Xu. Rule-based automatic software performance diagnosis and improvement. In *WOSP*. ACM, 2008.

76. Sherif M. Yacoub, Bojan Cukic, and Hany H. Ammar. A scenario-based reliability analysis approach for component-based software. *IEEE Transactions on Reliability*, 53(4):465–480, 2004.