Politecnico di Milano
Dipartimento di Elettronica e Informazione

DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

Model-Based Verification and Adaptation of Software Systems @Runtime

Doctoral Dissertation of:
Antonio Filieri

Advisor:
Prof. Carlo Ghezzi

Tutor:
Prof. Gianpaolo Cugola

Supervisor of the Doctoral Program:
Prof. Barbara Pernici

2012 - XXV

Abstract

The pervasiveness and flexibility of modern software systems and the uncertainty, unpredictability, and variability of their execution environments are strengthening the quest for systems that can self-adapt to their surrounding context.

At run time, systems are required to monitor the environment, identify possible violations of the requirements, and plan a suitable reaction.

Traditional techniques for automatic requirements verification are conceived for design time use and can hardly satisfy the time constraints imposed by run time analysis. Planning as well suffers from similar limitations, being most of the known approaches static and only effective in specific domains.

In this thesis several results from stochastic processes analysis, semantic interpretation of formal languages, and control theory are used to ground the definition of novel verification and adaptation methodologies devised for run time use, with a particular focus on non functional requirements such as reliability, performance, or cost. Particular attention has been paid to the generality and the formal assessment of the effectiveness of these methodologies, which have proved to be efficient and dependable in a wide application scope.

The contributions of this research can be summarized in the following list:

- a methodology for *run time efficient probabilistic model-checking*, that define a design time partial evaluation procedure that significantly simplifies and speeds up the verification at run time.

- an approach to *bring sensitivity analysis at run time* to estimate the impact of each monitored environmental condition online.

- a methodology for *syntactic-semantic incremental analysis*, that can be used to define incremental verification procedures of both functional and non-functional requirements.

- a methodology for *software control through Markov models*, providing for the automatic generation of broad-scope controllers able to adapt tunable software at run time in order to make it continuously satisfy its non functional requirements.

- a *reliability driven dynamic-binding* strategy grounded on control theory that provides formal assurance of scalability, robustness, and efficiency.

Sommario

Pervasività e flessibilità dei moderni sistemi software-intensive, assieme ad incertezza, non predicibilità e variabilità delle infrastrutture di calcolo, pongono l'accento sulla necessità di sistemi in grado di adattarsi autonomamente all'ambiente che li circonda.

Durante la loro esecuzione, tali sistemi devono essere in grado di monitorare l'ambiente, identificare possibili violazioni dei loro requisiti e pianificare reazioni opportune.

Le tecniche tradizionali per verificare il soddisfacimento dei requisiti sono tipicamente concepite per l'utilizzo in fase di progettazione e sviluppo e difficilmente riescono a soddisfare i vincoli temporali imposti dall'analisi a runtime. Gli algoritmi di planning soffrono spesso di limitazioni analoghe, essendo la maggior parte di essi pensati per contesti statici o per domini specifici.

In questa tesi, concetti propri dell'analisi dei processi stocastici, dell'interpretazione semantica dei linguaggi formali e della teoria del controllo costituiscono il fondamento di alcune metodologie innovative per la verifica e l'adattamento di sistemi software a runtime. La ricerca si focalizza prevalentemente sui requisiti non funzionali quali, ad esempio, affidabilità, performance o costo. Un'attenzione particolare è stata rivolta alla generalità delle procedure proposte e alla valutazione formale della loro efficacia, dimostrandone anche sperimentalmente efficienza e affidabilità in un ampio spettro applicativo.

I principali contributi di questa ricerca sono sintetizzabili nei seguenti punti:

- una metodologia per model-checking probabilistico a runtime che, basata su una valutazione parziale del problema durante la fase di design, è in grado di semplificare drasticamente la verifica dei requisiti a seguito di cambiamenti nell'ambiente o nel sistema.

- un approccio per effettuare un'analisi di sensitività a runtime allo scopo di stimare in tempo reale l'impatto di ciascuna delle condizioni ambientali monitorate durante l'esecuzione.

- una metodologia per l'analisi sintattico-semantica incrementale di artefatti software utile a definire procedure di verifica sia di requisiti funzionali che non funzionali.

- una metodologia per il controllo di sistemi software il cui control flow sia astraibile tramite processi Markoviani e che include la generazione automatica di controllori in grado di adattare il comportamento del sistema durante la sua esecuzione perché continui a soddisfare i suoi requisiti.
- un meccanismo di dynamic-binding adattativo mirato a garantire la convergenza dell'affidabilità del sistema al suo valore obiettivo. Tale meccanismo è fondato su risultati di teoria del controllo che ne garantiscono formalmente scalabilità, robustezza ed efficienza.

Contents

Part I

Overture

Introduction 1

Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away.

Antoine de Saint-Exupery

Self-Adaptive Software

Software is the backbone of modern society. Its structure, its development process, and the expectation people place on it have quickly changed since the dawn of Software Engineering [155], claiming for new directions and new perspectives.

Traditional software engineering processes, starting from the popular *waterfall* model, introduced in the 70s by Winston W. Royce, were mostly focused on how to discipline software development. Their inventors argued that careful requirements analysis could improve quality and avoid costly changes [20]. Avoiding changes was one of the central goals during early years of software engineering, in a time when organizations were monolithic, development centralized, deployment infrastructure well-known and mostly unchangeable, and application domains limited to critical systems, military, and big companies.

Taking a peek at the landscape of software in the past decade, almost none of these assumptions is still in place. Software development, provisioning, and maintenance is decentralized; systems are designed by assembling components developed and operated by third parties; binding between interfaces and implementations is delayed at run time; infrastructure is often *in the cloud* and may change as quickly as a minute; mobile devices are ubiquitous in everyday life, providing continuous interaction with billions of different users; networks are pervasive and heavily shape software execution.

Today software must change. If it could still be enough in many cases to design applications *for change* [162], in the near future software will be more and more required to continuously and autonomously adapt in response to unpredictable changes in its environment and goals.

In particular, self-adaptation is a key driver to deal with three challenges of modern software development [36]:

- **Volatility** of requirements, as consequence of the fast transformations of companies and customers
- **Uncertainty** about the effective operative conditions, hard to guess with accuracy at design time
- **Variability** in the behavior of the interacting environment: infrastructure, third party components, and customers.

A second difference between traditional software engineering and its current evolution concerns the role of *non-functional* requirements, such as reliability, performance, energy consumption, and cost. Customers require the continuous assurance of the agreed quality levels, despite the unpredictable changes the software undergoes. Most of non-functional requirements impose the satisfaction of specific *quantitative properties* [131], that have to be continuously verified in order to trigger a convenient adaptation process whenever a requirement is violated.

Verification @Runtime. Continuous verification of quantitative non-functional properties for self-adaptive systems is the first main focus of this thesis. Many current researches deal with the identification of "unhealthy" conditions that make the software violate its requirements. However, most of them are based on traditional verification techniques, that are conceived for design time use and can hardly meet the strict execution time constraints imposed by run time application.

The first verification methodology proposed in this thesis stands upon two complementary concepts: modeling and monitoring. The former aims at providing a semantic lens to interpret the data gathered from the running instances of the adaptive system. The latter is in charge of observing and measuring the relevant aspects of both the software and the execution environment and feed the information into the models, keeping them *alive at run time*

The models considered here describe the behavior of the software, as well as the relevant aspects of the environment, as a stochastic process. Such probabilistic models allow the formalization of a certain degree of uncertainty in the temporal progress of software execution, thus supporting both unsharp assumptions at design time and the intrinsic randomness of physical phenomena.

Models are assumed to capture an updated and consistent abstraction of the running software. They are not considered here as just a many-to-one relation between the real-world phenomena and a set of model elements, but as first class entities that can be used not only to verify the satisfaction of specific requirements, but also as a base for more complex reasoning to support adaptation. Indeed, when the running software no longer satisfies a requirement, a convenient adaptation has to be carried out.

The second methodology is instead based on a syntactic-semantic framework for the definition of incremental verification procedures. These procedures are driven by the syntactic structure of the artifact under analysis (described by a convenient formal grammar) and are encoded as semantic attributes associated with the production rules of the grammar itself. Incrementality is achieved by coupling the evaluation of semantic attributes with an incremental parsing technique.

This framework is general enough to effectively support the incremental verification of a large number of properties, including quantitative ones.

Model-Based Adaptation. Software adaptation is the second main focus of this thesis. Most of the adaptation approaches proposed in the past years are either thought for static environments, when time is not an issue, or only effective in specific domains. Self-adaptive software requires to deal with uncertainty and incomplete information from the system's self and its environment, to correlate local and global decision-making, and to provide scalability as well as formal assurance of the dependability of the adaptation mechanism.

Control theory has established effective mechanisms to make controlled plants behave as expected. Although the similarity with software adaptation is self-evident, most of the attempts to apply control theory to software applications failed because the intrinsic non linearity, the variety of usage profiles, and the interconnection of heterogeneous components make software systems hard to be modeled as a *dynamic system*, i.e. by means of differential equations. As result, the current use of control theory is limited to specific applications and hard to generalize to large classes of software.

This thesis tries to change such route. If modeling software as a dynamic system is in general an obstacle, a way to go around it is to derive the differential equations from the abstract behavioral model of the software. This two-step escape has been proved to be effective for all the software whose behavior can be described by a discrete time stochastic Markov model [175], with rewards, and for most of the goals expressible as the satisfaction of a probabilistic control tree logic assertion [97]. The obtained controllers can also provide formal assurance of their capabilities and notify the unfeasibility of the goal to higher level decision makers.

A specific control strategy has also been devised for the case of dynamic-binding, which is the main enabling technology for self-adaptation of service oriented applications. The specific issue of dynamic discovering and addition of new services at run time has been considered in proposing a control strategy that joins together the flexibility of the architecture and the formal assurances of its control theory foundation.

1.1 Contributions

The main contributions of this thesis are summarized in the following.

Run Time Efficient Probabilistic Model-Checking.
The model-checking task has been split in two phases to be carried out at design time and at run time, respectively. The role of the design time phase is to partially evaluate the problem, leaving at run time an effort as small as evaluating a closed-form expression. The procedure has been implemented and empirically compared with other existing approaches, showing a significant improvement in run time efficiency and a reasonable cost at design time. This contribution will the object of Chapter 4.

Sensitivity Analysis at Run Time.
The generation of closed-form expressions corresponding to the probability of satisfying a quantitative property allows for efficient sensitivity analysis with respect to the model parameters. The results of the analysis can support both the improvement of the system and the diagnosis of failures, and can be brought at run time. Sensitivity analysis is described in Chapter 4.4.

Syntax-Driven Incremental Quantitative Analysis.
A framework for the definition of incremental analysis procedures has been introduced for software artifacts. The analysis is driven by the syntactic structure of the software and encoded as the synthesis of semantic attributes. Incrementality is then automatically achieved by coupling the evaluation of semantic attributes with an incremental parsing technique. Syntax-driven incremental analysis is investigated in Section 5.

Software Control through Markov Models.
A general methodology has been defined for the control of tunable software [149] whose behavior can be described by a discrete time Markov model. The controller, automatically generated, has been proved to be robust to sudden changes in the environment and to monitoring inaccuracy, allowing for the continuous assurance of a wide family of quantitative requirements. This control methodology is presented in Chapter 6.

Reliability Driven Dynamic-Binding.
Dynamic-binding has been considered as a mean for a service to satisfy reliability requirements by continuously adjusting the choice among the available concrete implementations. Control-theoretical analysis and synthesis have been applied to prove effectiveness, stability, and scalability of the controller, and an auto-tuning procedure has been defined to set its configuration, even at run time. This topic is dealt with in Chapter 7.

1.2 Publications

Publications fundamental for the thesis contributions:

Run Time Efficient Probabilistic Model-Checking and Sensitivity Analysis.

3. A. Filieri, and G. Tamburrelli. Probabilistic Verification at Runtime for Self-Adaptive Systems. In Assurances for Self-Adaptive Systems book (Springer) – to be published.

2. A. Filieri, C. Ghezzi. *Further Steps Towards Efficient Runtime Verification: Handling Probabilistic Cost Models*. In Formal Methods in Software Engineering: Rigorous and Agile Approaches, FormSERA 2012. (Acceptance rate 50%)

1. A. Filieri, C. Ghezzi, and G. Tamburrelli. *Run-time efficient probabilistic model checking*. In International Conference on Software Engineering, ICSE 2011. (Acceptance rate 62/441, 14.1%) - **ACM Distinguished Paper Award**

Syntax-Driven Incremental Quantitative Analysis.

2. D. Bianculli, A. Filieri, C. Ghezzi, and D. Mandrioli. *A syntactic-semantic approach to incremental verification*. Internal Report, 2012.

1. S. Distefano, A. Filieri, C. Ghezzi, and R. Mirandola. *A compositional method for reliability analysis of workflows affected by multiple failure modes*. In Component Based Software Engineering, CBSE 2011. (Acceptance rate 29%)

Software Control through Markov Models.

2. A. Filieri, C. Ghezzi, A. Leva, M. Maggio. Discrete-time dynamic modeling for software and services composition as an extension of the Markov chain approach. In IEEE Multiconference on Systems and Control, MSC 2012. (Acceptance rate N/A) – to appear

1. A. Filieri, C. Ghezzi, A. Leva, M. Maggio. *Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability Requirements*. In Automated Software Engineering, ASE 2011. (Acceptance rate 37/252, 14.7%)

Reliability Driven Dynamic-Binding.

2. A. Filieri, C. Ghezzi, A. Leva, M. Maggio. Autotuning control structures for reliability-driven dynamic binding. In IEEE Conference on Decision and Control, CDC 2012. (Acceptance rate 50%) – to appear

1. A. Filieri, C. Ghezzi, A. Leva, M. Maggio. *Reliability-driven dynamic binding via feedback control*. In International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2012. (Acceptance rate 15/50, 30%)

Other publications related to this research:

7. A. Ciancone, A. Filieri, and R. Mirandola. *Testing Operational Transformations in Model-Driven Engineering*. In Innovations in Systems and Software Engineering, 2012 - to appear

6. A. Filieri, C. Ghezzi, and G. Tamburrelli. *A formal approach to adaptive software: Continuous assurance of non-functional requirements*. In Formal Aspects of Computing Journal, 2011.

5. A. Ciancone, A. Filieri, M. L. Drago, R. Mirandola, and V. Grassi. *Klapersuite: an integrated model-driven environment for non-functional requirements analysis of component-based systems*. In TOOLS, 2011. (Acceptance rate 19/66, 28.8%)

4. A. Ciancone, A. Filieri, and R. Mirandola. *Mantra: Towards model transformation testing*. In International Conference on the Quality of Information and Communications Technology, QUATIC 2010. (Acceptance rate 16/140, 11.4%) - **Best Paper Award of Verification and Validation Track**

3. A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola. *Reliability analysis of component-based systems with multiple failure modes*. In Component Based Software Engineering, CBSE 2010. (Acceptance rate 14/48, 29.2%) - **Best Paper Award**

2. A. Filieri. *QoS verification and model tuning @ runtime*. In European Software Engineering Conference/Foundation of Software Engineering - Doctoral Symposium, ESEC/FSE 2011.

1. A. Filieri, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Conquering complexity via seamless integration of design-time and run-time verification. In Conquering Complexity book (Springer), 2011.

1.3 Thesis Structure

The rest of this thesis is structured as follows.

Part II introduces the probabilistic models (Chapter 2) and the specification languages (Chapter 3) that will be used to abstract the behavior of a software system and to specify its quantitative non-functional requirements.

Part III describes the methodology for run time efficient probabilistic model-checking and the sensitivity analysis approach, in Chapter 4, and the syntactic-semantic incremental analysis framework in Chapter 5.

Part IV deals with the adaptation strategies. In particular, Chapter 6 is concerned with the control of tunable software systems through a Markov process abstraction of their behavior, while Chapter 7 faces the specific problem of reliability-driven dynamic-binding.

Finally, Part V reports conclusions and final remarks.
Each chapter includes a related work section tailored to its contents.

Part II

Modeling

Probabilistic Models 2

It is the mark of an instructed mind to rest satisfied with the degree of precision which the nature of the subject admits, and not to seek exactness when only an approximation of the truth is possible.

Aristotle

Modeling the run time of a system means capturing and describing its behavior, including the interaction with users and external environment. Indeed, physical execution resources, interdependence with third-party components, variability of usage profiles, and undiscovered defects produce a usually non negligible effect on the perceived behavior of the system. Requirements may change as well. All these changes are not controllable by the application and may occur autonomously and unpredictably [21, 73]. For example, the arrival rate of control data in an embedded system may change due to an unanticipated physical phenomena; the error rate of sensing device may change during time because of battery consumption; the workload of an e-commerce system my suddenly increase during Christmas or the Black Friday; technical issues on the software governing a stock market may mar the exchange of a company shares.

The focus of this research is on *non-functional* requirements [188], in particular *reliability*, *cost*, and *performance*, that are significantly affected by environmental factors. Such factors may be hard to predict when the application is designed. Moreover, even if the predictions were initially accurate, they may be likely to change while the application is running. Finally, predictions are usually based on (expensive) human experience, or historical data or the observation of similar systems; even when such data is available and supports design-time assumptions on the environment, sudden changes in the usage profile may invalidate them.

Capturing quantitative non-functional information requires to deal with the unavoidable uncertainty. Abstracting the software process to a finite and countable set of relevant states allows to formalize it via a finite-state stochastic process. A stochastic process is a family of random variables that is intended to model a time dependent stochastically evolving dynamical system. Each of those random variables represent the state of the system at a given time point. More precisely, given a sample space Ω, a *stochastic process* is a mapping $X : T \times \Omega \to S$, where T is the set of time points (in this work $\mathbb{N}$[1] or $\mathbb{R}$) and S is the *state space* of X. To simplify the notation, the ω-dependence can be avoided, identifying the state of the process at time $t \in T$ by just $X(t)$.

The temporal evolution of stochastic processes representing a software system is dictated by its previous history and both known facts (or assumptions) and random variables capturing the uncertainty about users and environment. Markov processes are a special class of stochastic systems satisfying the Markov property (here stated for $T \subseteq \mathbb{N}$):

Definition 2.0.1 (Markov Property.) A stochastic process $X = \{X(n) : n \geq 0\}$ is said to have the Markov property if for each $n \geq 0$ and a subset $A \subseteq S$:

$$\begin{aligned}&Pr(X(n+1) \in A \mid X(0) = x_0, \dots, X(n-1) = x_{n-1}, X(n) = x_n)\\&= Pr(X(n+1) \in A \mid X(n) = x_n)\end{aligned}$$

for all $(x_0, x_1, \dots, x_{n-1}, x_n) \in S^{n+1}$.

The Markov property states the conditional independence between future and past, given the present. Informally, the current state

[1] In this dissertation $0 \in \mathbb{N}$, unless otherwise specified.

determines the future evolution completely, that is with no influence from the past history.

The value $Pr(X(n+1) \mid X(n))$ is called *one-step* transition probability. Under the assumption of finiteness and countability of S, the one-step transition structure of X can be summarized through a square *transition matrix* P, whose entry p_{ij} if the value $Pr(X(n+1) = s_j \mid X(n) = s_i)$ with $s_i, s_j \in S$.

A stochastic process satisfying the Markov property is named *Markov process* [175]. There are many variants of Markov processes, suitable for representing several aspects of the modeled systems such as reliability, cost, execution time, or energy consumption.

In sections 2.1 and 2.2 two classes of Markov processes are introduced, namely the Discrete Time Markov Chains (DTMCs) and the *Discrete-Time Markov Reward Models* (D-MRM). For each class a formal definition is provided, as well as a brief review of its main mathematical properties.

2.1 Discrete-Time Markov Chains

Discrete Time Markov Chains [175] are widely established models of software reliability [67, 165, 192]. They are mostly used for design time reliability assessment of systems composed by interacting parts (e.g. component-based software, or service oriented architectures) [107], though different applications can be found in literature (e.g. [87, 121]). The adoption of DTMCs implies that the system's behavior meets, with some tolerable approximation, the Markov property. This issue will be discussed after the mathematical definition in Section 2.1.2.

DTMCs are stochastic processes satisfying the Markov property and having time domain $T \subseteq \mathbb{N}$. They are defined as Kripke structures [86] with probabilistic transitions among states. The state space S is here assumed finite and countable.

Formally, a (labeled) DTMC is a tuple (S, s_0, P, L) where

- S is a finite set of states
- $s_0 \in S$ is the initial state
- $P : S \times S \rightarrow [0, 1]$ is a stochastic matrix
- $L : S \rightarrow 2^{AP}$ is a labeling function. AP is a set of atomic propositions. The labeling function associates to each state the set of atomic propositions that are true in that state.

The notations $P(i, j)$, $P(s_i, s_j)$, and p_{ij} are interchangeably used to identify entries of P. An element p_{ij} represents the probability that the next state of the process will be s_j given that the current state is s_i. The set $\{p_{ij}\}$ for a state s_i is called the *next-state distribution* and is formally a categorical distribution [163], implying that $\sum_{s_j \in S} p_{ij} = 1$. Unless otherwise specified, atomic propositions $s = s_i$ and $s = i$ are defined for each state s_i and univocally identify state s_i.

The probability of moving from s_i to s_j in exactly 2 steps can be computed as $\sum_{s_x \in S} p_{ix} \cdot p_{xj}$, that is the sum of the probabilities of all the paths originating in s_i, ending in s_j, and having exactly one intermediate step. The previous sum is, by definition, the entry (i, j) of P^2. Analogously, the probability of reaching s_j from s_i in exactly k steps is the entry (i, j) of matrix P^k. As a natural generalization, matrix $P^0 \equiv I$ represents the probability of moving from state s_i to state s_j in zero steps, i.e. 1 if $s_i = s_j$, 0 otherwise.

Execution paths. A sequence of states $\pi = s_0, s_1, s_2, \ldots$ is an execution *path* through the DTMC if for any pair s_i, s_{i+1} $p_{i\ i+1} > 0$. The notation $\pi[i]$ with $i \geq 0$ is used to refer to the i-th state in the path π. A path is said *finite* if the number of states in the sequence is finite and its length is denoted as $|\pi|$. The probability for a finite path to be observed is 1 if $|\pi| = 1$, otherwise $\prod_{k=0}^{|\pi|-2} P(s_k, s_{k+1})$[2]. A state s_j is *reachable* from state s_i if there exists a finite (sub-)path starting in s_i and terminating in s_j.

States classification. A state s_i is said to be *transient* if:

$$\sum_{n=1}^{\infty} p_{ii}^{n} < \infty$$

in other words, the number of transitions into state s_i is finite. Non transient states are said *recurrent*. Formally, a state s_i is recurrent if:

$$\sum_{n=1}^{\infty} p_{ii}^{n} = \infty$$

A recurrent states will be visited infinitely often by the Markov process, while the number of visits to a transient state is finite and distributed as a geometric random variable [186]. A recurrent state s_i with $p_{ii} = 1$ is called *absorbing*. If a DTMC contains at least one absorbing state, the DTMC itself is said to be *absorbing*.

For simplicity, all the Markov models considered in this dissertation are assumed to satisfy the following property, unless otherwise specified:

Definition 2.1.1 (Well-formed DTMCs.) A DTMC model is *well formed* if:

- every recurrent state is an absorbing state
- all the states are reachable by the initial state
- from every transient state it is possible to reach at least one absorbing state.

[2] A detailed definition of the probability measures of DTMC path can be found in [15].

In an absorbing DTMC with r absorbing states and t transient states, rows and columns of the transition matrix P can be reordered such that P is in the following *canonical form*:

$$P = \begin{pmatrix} Q & R \\ \mathbf{0} & I \end{pmatrix} \tag{2.1}$$

where I is an r by r identity matrix, $\mathbf{0}$ is an r by t zero matrix, R is a nonzero t by r matrix and Q is a t by t matrix.

Note that since Q specifies only the transitions between transient states, some of its row sums are strictly less than 1. This is immediate for every well-formed DTMC. For the same argument it comes the following theorem:

Theorem 2.1.1 (Probability of Absorption) *In an absorbing Markov chain, the probability of the process to be eventually absorbed is* 1 *(i.e. $Q^n \to 0$ as $n \to \infty$.)*[3]

The number n_{ij} of visits to a transient state s_j, for a process started in s_i, can be computed as the probability of visiting it at the first step, or at the second, or at the third, and so on. In matrix form:

$$N = I + Q^1 + Q^2 + Q^3 + \cdots = \sum_{k=0}^{\infty} Q^k$$

The last sum is a geometric series whose convergence is ensured by the result of Theorem 2.1.1, that is $Q^n \to \mathbf{0}$ [80]. The limit of the sum is indeed $(I - Q)^{-1}$. The matrix N is called *fundamental matrix* of the DTMC.

The fundamental matrix will be the core of a set of verification algorithms introduced in Part III.

[3]From each transient state s_i it is possible to reach (at least) an absorbing state; let m_i be the minimum number of steps needed to reach an absorbing state from s_i and p_i the probability of not reaching an absorbing state in m_i steps; then $p_i < 0$. Let m be the largest of the m_i, that is the number of steps to reach the farthest absorbing state, and p the largest p_i. The probability of not being absorbed in m steps is less or equal than p; in $2m$ is less or equal than p^2; in $3m$ is less or equal than p^3, and so on. Since $0 \leq p < 1$, the probability of not being absorbed is monotone decreasing with respect to the number of steps. This argument can be applied to every transient state, hence, since Q^n represents the probabilities of visiting a transient state at step n, $\lim_{n\to\infty} Q^n = \mathbf{0}$.

2.1.1 Modeling with DTMCs

In this section a few hints concerning the modeling process are provided. A survey of methodologies and good practices is out of the scope of this work (the interested reader may refer, for example, to [42, 157, 175]), but it is worthy to briefly discuss a few relevant aspects.

Modeling Software. DTMCs have been used to model a variety of phenomena, e.g. chemical reactions, DNA sequences, financial trading, demographic evolution, human behaviors, or business processes. Their use for modeling software behavior should not be surprising since their notation is quite familiar to practitioners. Roughly speaking, DTMCs can be seen as conventional state-transition systems with annotations on transitions through which additional non-functional aspects can be specified. State-transition systems are commonly used in practice by software designers and can be used at different levels of abstraction to model software systems [81, 164].

Several widely accepted standards for software modeling can also be automatically translated into DTMC models by means of automated model transformations (e.g. [79] proposes an approach starting from activity diagrams, and [82] from sequence diagrams). Many integrated design frameworks can automatically translate their design models into corresponding Markov chains in order to provide quality assessment (e.g. [23, 44, 169, 194]). Reverse engineering approaches have also been proposed to extract Markov models from implemented software (e.g. [22, 26]).

Due to their characteristics, DTMCs have been widely used to model systems reliability [67, 88, 107]. The common idea behind the various approaches is that one or more special states represent failure condition. The occurrence of a failure is then represented by a transition toward one of them. In Section 2.1.2.1 an example of reliability oriented modeling is provided.

The Meaning of a State. A state of a DTMC represents, in general, an *observable* condition of the running system that is relevant from the modeling perspective at the chosen abstraction level. In this thesis it is further assumed that it is possible to classify the execution of the system into one and only one state of the DTMC.

There is not a general receipt for mapping the execution state into a DTMC one. In [41] a DTMC state corresponds to a "functional module", that is a part of the program flow graph that can be identified analyzing the code and whose reliability is reasonably independent from each other module. Analogous definition of module have been proposed by Littlewood [143, 144] again for reliability analysis. In [37] or [73] a DTMC state usually represent the invocation of a remote service; a special state, usually the initial one, represents the user and its outgoing transitions embed the usage profile[4]. In [71], a state represents not only the execution of a service, but also embeds information about possible types errors in the data-flow propagated up to that point. In [122], a state represents a large components implementing several services. In [87] a state represents an aggregation of several source files and functions of the GCC compiler implementing a delimited functionality, based on human expertise.

Though there is not a general way to model a software, the state classification proposed in the previous section leads to a basic guidance on the use of absorbing states. An absorbing state, when reached, is not going to be left. This property make absorbing states suitable for describing permanent conditions of software executions such as the occurrence of an unrecoverable failure or the completion of the process. In this thesis, unless differently specified, it is assumed that every model has at least an absorbing state representing the termination of the execution.

Uncertainty and Variability. The temporal evolution of a DTMC is completely defined by its transition matrix [175]. The probabilistic nature of transitions is suitable for capturing the randomness of modeled phenomena, being the labels of transitions originating from the same state parameters of a categorical distribution. Variations to a model are defined as changes in the values of its parameters, i.e. to the entries of P. In this thesis two types of values are allowed for elements p_{ij}: numeric and symbolic. The former are used to describe phenomena assumed as known and stable (e.g. the failure probability of a cloud-based storage is usually assumed as constant during the execution of a process). The latter are instead used to formalize phenomena that are either unknown at design time and/or subject to

[4]In general the usage profile does not account only for human users, but for every external interaction of the software [114].

change during the execution (e.g. the usage profile of a news service may change during the day) [37, 73]. The actual values assigned to symbolic entries of P define the operative conditions of the system. Such conditions may be discovered only at run time and may change along the run[5].

In the following of this thesis, symbolic entries of P are referred to as *parameters* of the DTMC model, and a model having at least a parameter is said *parametric*.

2.1.2 Validity of the Markov Assumption

From Definition 2.0.1, in a Markov process the next state to be executed depends, probabilistically, on the current state only and is independent from the previous history. Though it not always easy to establish the satisfaction of the Markov property by looking at the source code of a software, several experiments showed that the Markov assumption often holds at an higher level, such as the architectural one [41,167,173].

When the next action depends on the previous history, there are still several cases that can be conveniently approximated by a Markov process.

The first is the case where the next step is affected by a limited number of previous moves, let say k of them. This situation can be modeled by a k-th order Markov process, that is one where the next action depends only on the previous k. It can be proved that, if the state space is finite, the expressiveness of k-th order Markov processes is the same as for the Markov processes introduced in this chapter (which are formally 1-st order) [175]. Hence, an equivalent formulation can be defined.

If a the system behavior cannot be described by a k-th order Markov process, augmented Markov models have been proposed to deal with specific problems, e.g. [191].

Nonetheless, a software systems might expose an intrinsically non Markov behavior [85,193], thus the Markov assumption must be verified before proceeding with the analysis [29].

[5]Discovering of changes at run time requires the continuous observation of the running system and suitable learning strategies. Monitoring and learning strategies for Markov models are discussed in [73]

2.1.2.1 Modeling Example

In this section an example of software behavior modeled through a DTMC is described.

Figure 2.1 represents the model of a typical web architecture. The system comprises an HTTP Proxy server, a Web server and an Application server. In addition, structured data and static content (e.g., files, images, etc.) are stored on a Database and on a File server, respectively. Both of them are cached by ad-hoc cache servers.

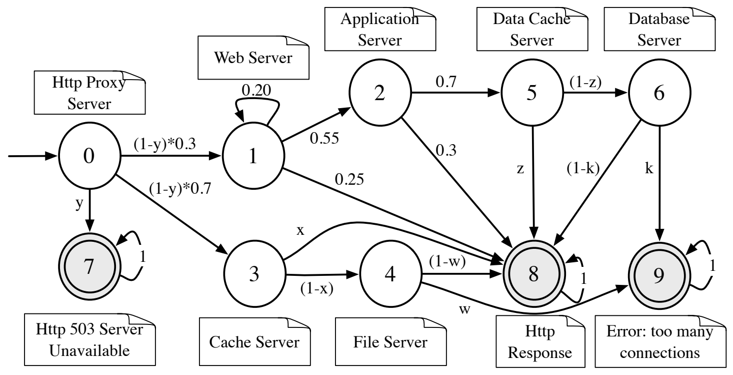


Figure 2.1: Example of a parametric DTMC.

States s_7, s_8 and s_9 are absorbing. s_7 represents the failure of serving an incoming request due to an unavailable server (e.g., overloaded server or maintenance downtime). s_9 represents the failure of the execution due to an excessive number of requests to the storage services. s_8 is the endpoint of a correct HTTP request.

Transitions among transient states probabilistically describe the control flow to manage an incoming HTTP request. For example the transitions (s_0, s_1) and (s_0, s_3) corresponds to the probability of the events "a dynamic content has been requested that require ad hoc processing" and "a static content has been requested", respectively. Transition (s_1, s_1) corresponds instead the probability of an HTTP self-redirect. Conversely, transitions to absorbing states indicate the final outcome in processing a request, or the occurrence of a failiure.

Parametric transitions indicate that the value of the corresponding probability is unknown and/or may change over time. For example transitions (s_3, s_4) and (s_5, s_6) indicate the cache hit probability that depends on the current distribution of user requests.

In matrix form, the model of Figure 2.1 is characterized by the following transient-to-transient (Q) and transient-to-absorbing (R) transition matrices:

$$Q = \begin{pmatrix} 0 & (1-y)0.3 & 0 & (1-y)0.7 & 0 & 0 & 0 \\ 0 & 0.2 & 0.55 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.7 & 0 \\ 0 & 0 & 0 & 0 & 1-x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1-z \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} y & 0 & 0 \\ 0 & 0.25 & 0 \\ 0 & 0.3 & 0 \\ 0 & x & 0 \\ 0 & 1-w & w \\ 0 & z & 0 \\ 0 & 1-k & k \end{pmatrix}$$

The example in Figure 2.1 is a small instance intended to show a practical application of the methods presented in the next part of the thesis. Real software systems are usually larger, though the same considerations apply regardless the model size.

2.2 Discrete-Time Markov Reward Models

A D-MRM [7] is a DTMC augmented with *rewards*. While the underlying DTMC captures the systems behavior (cf. Sect 2.1), rewards are non-negative real values through which a benefit (or loss) due to the residence in a specific state or the move along a certain transition can be quantified. As for DTMCs, the adoption of a Markov model implies that the modeled aspects of the system meet, with some tolerable approximation, the Markov property.

DTMCs are valuable formalism to model and reason about a software execution flow and its deviations (e.g. the occurrence of a failure) but are not able to directly represent some other common quantitative properties related, for example, to performance, energy consumption, or cost. Indeed, average execution time, number of I/O operations, cost of an outsourced operation, or energy can often be associated with states or transitions.

A D-MRM is a tuple (S, S_0, P, L, ρ, i) where:

- $S, S0, P$, and L are defined as for DTMCs,
- $\rho : S \rightarrow \mathbb{R}_{\geq 0}$ is a *state reward* function assigning to each state a non-negative real number,
- $\iota : S \times S \rightarrow \mathbb{R}_{\geq 0}$ is a *transition reward* function assigning a non-negative real number to each transition.

To clarify how rewards are gained, it is convenient to precisely state how the system modeled by the D-MRM evolves over a sequence of time steps. At step 0 the system enters the initial state s_0. At step 1, the system gains the reward $\rho(s_0)$ associated with the initial state and moves to a new state (say, $s1$), gaining also the reward $\iota(s_0, s_1)$. The cumulated reward when the system enters state $s1$ is $\rho(s_0) + \iota(s_0, s_1)$. At step 2, it gains the reward $\rho(s_1)$ associated with state $s1$, and then exits it gaining also the reward associated with the chosen transition, and so on. In summary, the state reward is acquired if the D-MRM resides in state s_i for one time step. The reward associated with a transition $\iota(s_i, s_j)$ is gained as the process makes a move from state s_i to state s_j. Due to this nature, state rewards are sometimes called *cumulative* while transition rewards are called *instantaneous* [132].

As for DTMC, a state $s_i \in S$ is said to be an *absorbing state* if $P(s_i, s_i) = 1$. If a D-MRM contains at least one absorbing state, the

D-MRM itself is said to be an *absorbing D-MRM*. Notice that if an absorbing state with reward > 0 can be reached then the cost of the process is ∞ because of the infinite sum of a positive constant value.

Definition 2.1.1 of well form for DTMCs is adopted as-is for D-MRM.

The temporal evolution of a D-MRM is dictated by the evolution of the underlying DTMC, i.e. the one-step transition probability is defined by matrix P and the entailed considerations apply.

As for DTMCs, assume that variability does not affect the structure of the models, only parameters. In our case, it only affects the possible values used to label transition probabilities and rewards. This is usually expressive enough to accommodate changes in the environment that affect our system.

2.2.1 Modeling with D-MRMs

Markov reward models extend the expressiveness of DTMCs and allow to represent a larger number of quantitative qualities. For example [38] discusses two case studies concerning the dynamic power management of disk drives, and the adaptive management of cluster availability within data centers; [150] uses D-MRMs to trade-off energy and reliability requirements of embedded systems; in [90], D-MRMs are used to modeling mobile code in wireless networks and its cost in terms of generated network traffic and energy consumptions of mobile nodes.

In [98], several specification techniques are proposed to map established performance, dependability, and cost models such as stochastic petri nets, queuing networks, fault trees, communication processes into D-MRMs. Thanks to their versatility, D-MRM can also be used to model domain specific qualities in a quite natural way.

State Rewards and Transition Rewards. Though state and transition rewards have a natural correspondence to the concepts of residing into a state and moving from a state to another, any transition reward can be mapped to an equivalent state reward and vice versa by automatic transformation of the underlying process.

Figure 2.2 sketches a transformation apt to replace transition rewards by state ones. The rational behind this procedure is the augmentation of the D-MRM with states representing to the “firing of a transition". The time scale has to be considered accordingly, since the transformation halve the frequency by two.

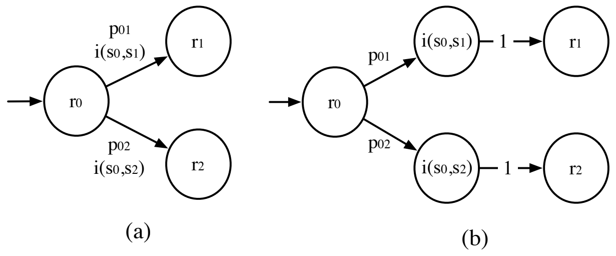


Figure 2.2: Translating transition rewards to state rewards.

For the previous consideration, without loss of generality, both models and verification algorithms presented in this thesis are focused on state rewards only.

2.2.1.1 Modeling Example

Following from Section 2.1.2.1, consider the following deployment for the web application. The database and the file server are deployed on a Cloud infrastructure in which bandwidth and space are billed (e.g. the Amazon Simple Storage Service[6]). In this setting, it is possible to associate to each state a pair of real values representing, respectively, the average latency in seconds and the average cost in 10^{-2} dollars. Latency includes the average processing time and the network latency, while the cost is based on the average CPU utilization.

[6]http://aws.amazon.com/s3

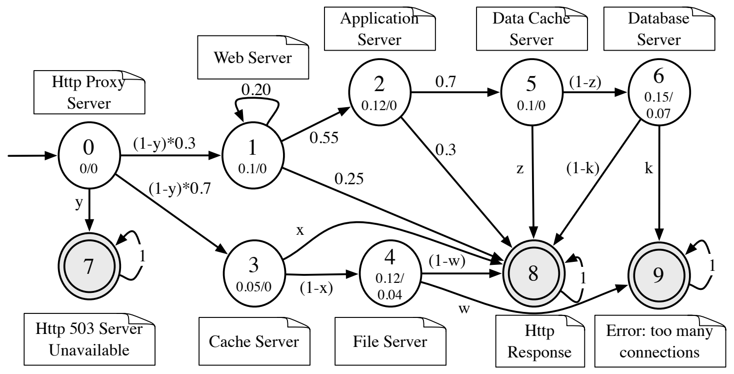


Figure 2.3: Example of a parametric D-MRM.

The resulting model is shown in Figure 2.3, where, for example, the average cost for a each request processed by the database is 0.0007$.

Probabilistic Specification Logics 3

Contemplate consistent notions having more than mere in-ternal consistency; they have a positive drive to come into being. The more they have of this, the more possible they are. 'The possible demands existence by its very nature, in proportion to its possibility, that is to say, its degree of essence'.

Gottfried Wilhelm von Leibniz

Classical analysis of Markov models usually focuses on *transient* or *steady-state* behavior [129]. These two probabilistic methods allow to investigate the probability of the process to be in a certain state at a certain time or in the long run, respectively. Despite their long tradition in mathematical research, transient and steady-state analysis are not naturally suited for expressing *behavioral properties*, such as the probability of eventually reaching a certain state or never hitting an error before completion.

Quantitative probabilistic behavioral properties are natural formulations of common software requirements, such as invariance, precedence, response, or constrained or unconstrained reachability that can be interpreted on probabilistic models, like Markov processes[1]. In general, they can be used to specify constraints on the probability that certain (un)desired behaviors may be observed on the running system. Examples of probabilistic properties considered in this thesis are (recalling the example in Figure 2.2):

- **R1** (*Reliability*): "The probability of successfully handling a request must be greater than 0.999"
- **R2** (*Cache hit probability*): "At least 80% of the requests are correctly handled without accessing the database or the file server"
- **R3** (*Complexity bound*): "70% of the requests must be successfully processed within 5 operations"
- **R4** (*Early risk fingering*): "No more than 10% of the runs can reach a state from which the risk of eventually raising an exception is greater than 0.95"
- **R5** (*Cost*): "The average cost for handling a request must be less $.03 \cdot 10^{-2}$ dollars"
- **R6** (*Response time*): "The average response time must be less than 0.022 seconds"

The previous informal requirements must be translated into a convenient formal language in order to apply automatic verification techniques. To this purpose, in Chapter 3.1, Probabilistic Computation Tree Logic (PCTL) will be defined. PCTL is a formalism suitable for the specification of behavioral properties of DTMCs. In Chapter 3.2 the logic will be extended to support the specification of reward-related properties too.

[1] Most of the readers should be familiar with the formal verification of behavioral properties of software on deterministic models [63]. Though in that case the goal is to providing absolute guarantee (true/false) of satisfaction, similar specification patterns [66, 93] are often straightforwardly adapted to probabilistic verification.

3.1 Probabilistic Computation Tree Logic (PCTL)

Probabilistic Computation Tree Logic (PCTL) [13,97] is a branching-time temporal logic, based on the logic CTL [15]. A PCTL formula expresses conditions on a state of a Markov process, and it is evaluated to either *true* or *false* on it.

Its syntax is recursively defined by the following rules:

$$\begin{aligned}
\phi &::= true \mid a \mid \phi \wedge \phi \mid \neg \phi \mid \mathscr{P}_{\bowtie p}(\psi) \\
\psi &::= X\phi \mid \phi U^{\leq t} \phi
\end{aligned}$$

where $p \in [0,1]$, $\bowtie \in \{<, \leq, >, \geq\}$, $t \in \mathbb{N} \cup \{\infty\}$, and a represents an atomic proposition.

The temporal operators X and U are called *Next* and *Until*, respectively. Formulae originated by the axiom ϕ are said *state formulae*; those originated by ψ are instead said *path formulae*. Notice that a path formula may only occur as argument of the modal operator $\mathscr{P}_{\bowtie p}(\cdot)$.

Unlike CTL, universal and existential path quantification is not defined for PCTL.

The semantics of a state formula is defined as follows:

$$\begin{array}{lll}
s & \models true & \\
s & \models a & \text{iff} \quad a \in L(s) \\
s & \models \neg\phi & \text{iff} \quad s \not\models \phi \\
s & \models \phi_1 \wedge \phi_2 & \text{iff} \quad s \models \phi_1 \ and \ s \models \phi_2 \\
s & \models \mathscr{P}_{\bowtie p}(\psi) & \text{iff} \quad Pr(\pi \models \psi | \pi[0] = s) \bowtie p
\end{array}$$

where $Pr(\pi \models \psi | \pi[0] = s)$ is the probability that a path originating in s satisfies ψ, and can be computed as described in Section 2.1.

A path π originating in s satisfies a path formula ψ according to the following rules:

$$\begin{array}{lll}
\pi & \models X\phi & \text{iff} \quad \pi[1] \models \phi \\
\pi & \models \phi_1 U^{\leq t} \phi_2 & \text{iff} \quad \exists 0 \leq j \leq t (\pi[j] \models \phi_2 \wedge (\forall 0 \leq k < j\, \pi[k] \models \phi_1))
\end{array}$$

As a short form for $trueU^{\leq t}\phi$, the use of the operator $\diamond$ (*eventually*) is allowed: $\diamond^{\leq t} \phi$. It is customary to abbreviate $U^{\leq\infty}$ and $\diamond^{\leq\infty}$ as U and $\diamond$, respectively.

PCTL can naturally represent a large number of properties of a DTMC. For example, it can express constraints on the probability of reaching an absorbing failure or success state, given the initial one. This property is an example of the general class of *reachability properties*. Reachability properties are expressed as $\mathscr{P}_{\bowtie p}(\Diamond\ \phi)$, which states that the probability of reaching a state where ϕ holds matches the constraint $\bowtie p$.

3.1.1 Specification Example

Recalling the example DTMC model of Section 2.1.2.1, and the set of requirements informally stated at the beginning of this chapter, a PCTL formalization of requirement **R1**-**R4** is provided in Table 3.1.1[2].

Table 3.1: PCTL formalization of requirements **R1**-**R4**.

ID	**Informal Definition**	**PCTL**
R1	(*Reliability*): "The probability of successfully handling a request must be greater than 0.999"	$\mathscr{P}_{\geq 0.999}(\Diamond\ s = s_8)$
R2	(*Cache hit probability*): "At least 80% of the requests are correctly handled without accessing the database or the file server"	$\mathscr{P}_{\geq 0.8}(\neg(s = s_4) \wedge \neg(s = s_6)\ U\ s = s_8)$
R3	(*Complexity bound*): "70% of the requests must be successfully processed within 5 operations"	$\mathscr{P}_{\geq 0.7}(\Diamond^{\leq 5}\ s = s_8)$
R4	(*Early risk fingering*): "No more than 10% of the runs can reach a state from which the risk of eventually raising an exception is greater than 0.95"	$\mathscr{P}_{\leq 0.1}(\Diamond\ \mathscr{P}_{\geq 0.95}(\ \Diamond s = s_7 \vee s = s_9))$

[2]Where $\phi_1 \vee \phi_2$ is a short form for $\neg(\neg\phi_1 \wedge \neg\phi_2)$.

3.2 Extending PCTL With Rewards (R-PCTL)

R-PCTL is an extension of PCTL where a new modal operator is added to allow for reward properties [15, 132]:

$$
\begin{aligned}
&\phi ::= true \mid a \mid \phi \wedge \phi \mid \neg \phi \mid \mathscr{P}_{\bowtie p} (\psi) \mid \mathscr{R}_{\bowtie r} (\Theta) \\
&\psi ::= X\ \phi \mid \phi\ U\ \phi \mid \phi\ U^{\leq t}\ \phi \\
&\Theta ::= I^{=k} \mid C^{\leq k} \mid \diamond\phi
\end{aligned}
$$

where the symbol $\bowtie$ stands for a relational operator in the set $\{\leq, <, \geq, >\}$, $p \in [0,1]$ is a probability bound, $r \in \mathbb{R}_{\geq 0}$, and $k \in \mathbb{Z}_{\geq 0}$.

The state properties defined by $\mathscr{R}_{\bowtie r}(\Theta)$ allow for the specification of *reward properties*.

Before introducing a formal definition for the semantics of the rewards fragment of R-PCTL, an intuitive description follows:

- $\mathscr{R}_{\bowtie r}(I^{=k})$ is true in state s if the expected state reward to be gained in the state entered at step k along the paths originating in s meets the bound $\bowtie r$.
- $\mathscr{R}_{\bowtie r}(C^{\leq k})$ is true in state s if, from state s, the expected reward *cumulated* after k steps meets the bound $\bowtie r$.
- $\mathscr{R}_{\bowtie r}(\diamond\phi)$ is true in state s if, from state s, the expected reward cumulated before reaching a state where ϕ holds meets the bound $\bowtie r$.

The third construct can be used, for example, to state the average cost of a run of the system, that is, the expected cumulated cost until the execution reaches a *completion* state.

A more detailed definition of the reward fragment semantics can be found in [132]. Intuitively, the expected reward $\mathscr{R}(\Theta)$ for all possible paths exiting a given state s and satisfying the pattern Θ can be computed as the sum of the rewards for each of those paths, weighted by the probability the path itself (cf. Section 2.1).

The following equations define how the expected reward X_Θ over a path π of the D-MRM is computed for each of the three specification patterns:

$$X_{I^{=k}}(\pi) = \rho(s_k) \tag{3.1}$$

$$X_{C^{\leq k}}(\pi) = \begin{cases} 0 & \text{if } k = 0 \\ \sum_{i=0}^{k-1} \rho(s_i) + \iota(s_i, s_{i+1}) & \text{otherwise} \end{cases} \tag{3.2}$$

$$X_{F\phi}(\pi) = \begin{cases} 0 & \text{if } s_0 \models \phi \\ \infty & \text{if } \forall i\ s_i \not\models \phi \\ \sum_{i=0}^{min\{j|s_j\models\phi\}-1} & \\ \qquad \rho(s_i) + \iota(s_i, s_{i+1}) & \text{otherwise} \end{cases} \tag{3.3}$$

As the reader could have noticed, the previous equations are more of a definition than an executable procedure (in particular (3.3)). In Section 4.3 algebraic techniques will be proposed to compute the values X_Θ, tacking into account the presence of both numeric and parametric rewards (and transition probabilities).

3.2.1 Specification Example

Recalling the D-MRM example of Section 2.2.1.1 and the quantitative non functional requirements informally stated at the beginning of this chapter, the introduction of R-PCTL allows a precise formalization of requirements **R5**-**R6** as reported in Table 3.2.1.

Notice that though the R-PCTL formulae of the two requirements share a similar structure, the state rewards to be considered for the first are the *average costs*, while the ones to be considered for the second are the *average latencies*.

Table 3.2: R-PCTL formalization of requirements **R5**-**R6**.

ID	Informal Definition	PCTL
R5	(*Cost*): "The average cost for handling a request must be less $.03 \cdot 10^{-2}$ dollars"	$\mathscr{R}_{\leq 0.03}(\Diamond\ s = s_7 \vee s = s_8 \vee s = s_9)$
R6	(*Response time*): "The average response time must be less than 0.022 seconds"	$\mathscr{R}_{\leq 0.022}(\Diamond\ s = s_7 \vee s = s_8 \vee s = s_9)$

Part III

Verification

The WorkingMomParadigm 4

It's not the strongest who survive, nor the most intelligent, but the ones most adaptable to change.

Charles Darwin

Efficient Verification @ Runtime

The automated verification of quantitative properties of software systems have made relevant progresses in the past few years, and are now included into several software engineering processes as part of the verification and validation tasks [131]. In particular, verification of (R-)PCTL properties of Markov models is quickly spreading through industrial applications in critical sectors, such as energy [156], aerospace [46], biology [133], or automotive [4]. The vast majority of the methods are based on model-checking techniques [15,47], where a large set of algorithms have been designed to exhaustively traverse the models and prove or refute the satisfaction of software requirements. Examples of probabilistic model-checking procedures can be found in [16,18,27,51,97,111,189].

Despite its generality and usability, model checking suffers from the state-explosion problem, which limits the size of models that can be verified under limiting time constraints of run time analysis [131].

Some approaches have brought state-of-the-art probabilistic model-checkers at run time [37], providing a suitable infrastructure for many applications. Nonetheless these approaches are not general enough for two reasons. First, the complexity of verification can be too high in case of large systems to make the analysis meet its time constraints [52, 111]. Second, analysis procedures may be unsuitable for low power devices where the large number of operations required for the mathematical iterative algorithms commonly used by model-checkers, resulting in excessive time and energy consumption.

Model-checking can be improved in many situations both in terms of analysis algorithms, e.g. by applying space-reduction techniques (e.g. [17, 116]), and via the reuse of previous results, thus opening the way for incremental analysis [130]. Also GPU implementations are under investigations for data-parallel implementations [33].

Parametric Model-Checking. Besides improving standard model-checking procedures, a different approach has recently gained relevance for run time analysis. In his seminal work [54], Daws describes a procedure for parametric probabilistic model-checking a subset of PCTL over DTMCs. This result trod an effective path for bringing probabilistic verification at run time, by allowing the separation of the analysis process in two steps. The first consists in the parametric analysis of the model with respect to the desired property, whose result is a closed mathematical expression depending on the symbolic variables appearing in the model. This step is quite complex in terms of computational time, but it can be accomplished once for all at design time, when time is usually not a strong constraint. At run time all that is needed to obtain the actual analysis response is to replace the symbolic variables with the actual values provided from modeling, as soon as they are discovered. The evaluation of a mathematical expression is in general a much simpler task than model-checking, and can be performed in a very short time even on low power devices, as shown in [70, 72].

The main contribution of this part of the dissertation is a novel method for parametric probabilistic verification of (R-)PCTL properties over discrete-time Markov models. In Section 4.1 the global picture of the entire analysis process is introduced. In Sections 4.2 and 4.3 specific algorithms for parametric analysis of PCTL and (R-

)PCTL properties are discussed. In Section 4.4, the impact of model parameters on the satisfaction of system's quantitative requirements is analyzed through sensitivity analysis. Related work is discussed in Section 4.5. Finally in Section 4.6 the efficiency of the proposed approach is empirically evaluated on a large set of randomly generated input models.

4.1 The Working Mom Paradigm

The "WorkingMom" paradigm owes its name to the analogy with a working parent who prepares the meal in the morning, when time is available, and then warms it up for lunch, when only a short break is possible. The application of this paradigm to probabilistic verification is similarly composed of two stages:

1. At *design time*, when time is not a critical constraint, a pre-processing phase leads to the parametric partial evaluation of the problem. The result of partial evaluation is a closed rational polynomial expression having as unknowns the parameters of the models.

2. At *run time*, when time is short, the verification procedure is finalized by evaluating the rational expression with the actual values of the model parameters.

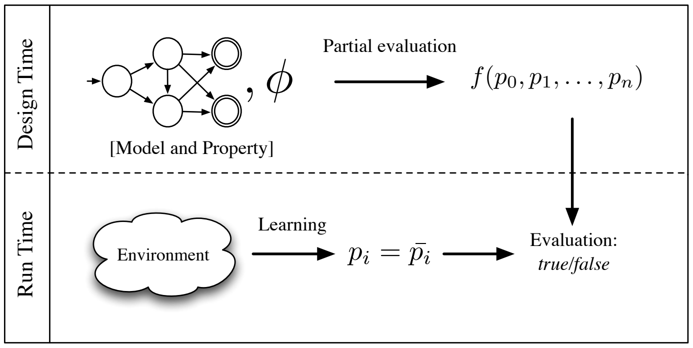


Figure 4.1: WorkingMom process overview.

Being essentially reduced to the evaluation of a polynomial, the run time stage is basically more efficient than executing a model-checking procedure from scratch. Furthermore, it can be executed on even on low power devices because it does not require any complex operation.

On the other hand, the length of the mathematical expression may grow quickly with the number of model parameters. For the verification of reachability properties, which are the most used in practice [93], the length of the expression may growth up to $O(n^{log(n)})$ for a fully symbolic model, i.e. when all the transitions are parametric [54]. On this concern, it is here assumed that, through careful design time analysis, run time variability can be restricted to a subset of environment parameters. Precisely, it is assumed that (1) the variable transitions in the model can be identified and (2) they are a small fraction of the total number of transitions. These assumptions are valid in many practical cases. If they do not hold, the WM approach may still be applied, but could yield smaller benefits in terms of speed-up of run time verification.

Generating the rational expression at design time is the most complex task and the focus of the following sections. The algorithms for design time analysis are based on symbolic algebra and are specific for different classes of models and properties.

Section 4.2 introduces the verification of PCTL, providing different strategies to improve design time efficiency for different problem settings. Section 4.3 presents instead the set of algorithms for the verification of R-PCTL properties.

4.2 PCTL Verification

This section illustrates the algorithms for partial evaluation of PCTL properties at design time. Pre-computation produces a rational expression for each property. PCTL properties can be verified against both DTMCs and D-MRMs.

To simplify the exposition, PCTL will be partitioned in several fragments. Section 4.2.1 deals with *flat*[1] formulae for the reachability of an absorbing state, showing the set of algorithms available for the WM. These formulae have the syntactic form $\mathscr{P}_{\bowtie p}\ (\mathit{true}\ U\ \phi_1)$[2] where ϕ_1 identifies one or more absorbing states.

In Section 4.2.2 the remaining fragments will be studied, providing a full set of WM algorithms for the analysis of PCTL.

4.2.1 Reaching an Absorbing State

Let us start by focusing on flat reachability formulae for absorbing states. Recalling the structure of the transition matrix for an absorbing DTMC given in Equation (2.1), the matrix $I-Q$ (where I is the identity matrix of the same size of Q) has an inverse N and $N = I+Q+Q^2+Q^3+\cdots = \sum_{i=0}^{\infty} Q^i$ [175]. Recall from Section 2.1 that an entry q_{ij} of the matrix Q represents the probability of moving from the transient state s_i to the transient state s_j in exactly one time step. The entry n_{ij} of N represents the number of times the Markov process is expected to visit the transient state s_j before being absorbed, given that it started from state s_i. A Markov process is considered *absorbed* when it reaches any of the absorbing states. Notice that $Q^n \to 0$ when $n \to \infty$ (cfr. Theorem 2.1.1), hence, after enough time, the process will always eventually be absorbed, no matter which state it started in.

Every time the process accesses a transient state s_i, it has a probability of being absorbed in the next time step in the absorbing state s_j given by the entry r_{ij} of the matrix R. Generalizing to all the pairs (s_i, s_j) where s_i is transient and s_j is absorbing, we can get the absorbing distribution B of the DTMC as:

$$B = N \times R$$

[1] In general, a formula is said *flat* if none of its sub-formulae contain the $\mathscr{P}_{\bowtie p}(\cdot)$ operator, but the formula itself.

[2] Or equivalently $\mathscr{P}_{\bowtie p}\ (\diamond\ \phi_1)$ (cfr. Section 3.1).

An entry b_{ij} of the matrix B represents the probability for the process of being eventually absorbed in s_j (in any number of states), given that it started from s_i. B is by construction a $t \times r$ matrix, where t is the number of transient states and r the number of absorbing ones.

Given a DTMC D and a set T of target absorbing states, the probability of reaching T from the initial state s_0 can be computed as:

$$Pr(true\, U\ \{s \in T\}) = \sum_{s_j \in T} b_{0j} \tag{4.1}$$

The goal of design-time pre-computation is to compute the value of Equation (4.1). Depending on the size of the system and the availability of a parallel or a sequential execution environment the computation of the matrix B can be performed in different ways.

An entry b_{ij} can be computed, by the definition of matrix product, as:

$$b_{ij} = \sum_{k=0..t-1} n_{ik} \cdot r_{kj} \tag{4.2}$$

Entries r_{ij} are readily available from matrix R. Entries n_{ik} belong instead to the i-th row of matrix N, that is the inverse of $I - Q$.

In Sections 4.2.1.1 and 4.2.1.2, two different approaches will be discussed for the computation of the entries n_{ik}.

The first, based on matrix algebra procedures, can be quite effective in case of a small number of parameters, even in a sequential execution environment. Furthermore, thanks to its formulation is intrinsically parallel and suitable for different kinds of parallelization.

The second reduce the problem to the solution of a system of linear equations. This approach is quite efficient for sequential execution environments thanks to the implementation of effective heuristics for sparse linear systems.

Finally, in Section 4.2.1.3 a different strategy is used to compute $Pr(F\{s \in T\})$. In this case locality properties of Markov processes will be exploited to again transform the problem in the solution of system of linear equations through the so-called *first-step* analysis.

4.2.1.1 Matrix-Based Approach

The design-time computation of an entry b_{ij} in general can only be done symbolically, since parametric transitions may be traversed to reach state s_j. The complexity of explicitly inverting matrix $I - Q$ by means of the Gauss-Jordan elimination algorithm [6] requires $O(t^3)$ operations. The computation of the entry b_{ij} once N has been computed requires $O(t)$ more products, thus the total complexity is $O(t^3 + t) \sim O(t^3)$ algebraic operations on polynomials.

The actual complexity can be significantly reduced if the number c of states having parametric outgoing transitions is small and the transition matrix of the DTMC is sparse, as very frequently happens in practice.

Let $W = I - Q$. The elements of its inverse N are defined as follows:

$$n_{ij} = \frac{1}{det(W)} \cdot \alpha_{ji}(W) \tag{4.3}$$

where $\alpha_{ji}(W)$ is the cofactor of the element w_{ji}. Thus:

$$b_{ik} = \sum_{x \in 0..t-1} n_{ix} \cdot r_{xj} = \frac{1}{det(W)} \sum_{x \in 0..t-1} \alpha_{xi}(W) \cdot r_{xj} \tag{4.4}$$

Complexity issues. Computing b_{ik} requires the computation of t determinants of square matrices with size $t - 1$. Let τ be the average number of outgoing transitions from each state ($\tau << n$ by assumption). Each of the determinants can be computed by means of Laplace expansion. Precisely, by expanding first the c rows representing the variable states (each has τ symbolic terms), at most τ^c determinants have to be computed and then linearly combined. Each sub-matrix of size $t - c$ does not contain any variable symbol, by construction, thus its determinant can be computed with $(t - c)^3$ operations among numeric values. The latter operation does not involve symbolic terms, hence it is in general much faster. Its actual complexity depends on the precision of floating-point (or rational numbers) representation. On the other hand, memory could easily become an issue for sequential environments because both intermediate results and a possibly large set of sub-matrices have to be stored for processing; for this reason in a sequential environment only small systems can be analyzed with this algorithm (cf. Section 4.6).

The final complexity is thus:

$$O(\tau^c \cdot (t-c)^3) \sim O(\tau^c \cdot t^3) \tag{4.5}$$

which significantly reduces the original complexity and makes the design-time pre-computation of reachability properties feasible in a reasonable time, even for large values of t.

As a point of comparison, the computation of reachability properties performed by probabilistic model-checkers is based on the solution of a system of n equations in n variables [15], which has, in a sequential computational model, a complexity equal to n^3 [32].

Notice that the procedure described in this section is naturally parallelizable in several ways. First, the sum in Equation (4.4) is intuitively formalizable in a map-reduce pattern [171], where the *map* operation is the computation of each cofactor and the *reduce* is the sum of the results. Furthermore, since the cofactor of a matrix containing symbolic entries can again be computed by Laplace expansion, it is possible to design a hierarchical map-reduce configuration. This approach is valid both for multicore and distributed execution environments. The main limitation in case of multicore could be the amount of memory required to store all the intermediate results. Second, either limited to the computation of numeric determinants, or applied to the computation of the cofactors, parallel algorithms for matrix algebra have been largely applied [78].

4.2.1.2 Equations-Based Approach

The matrix-based approach proposed in the previous section is a powerful tool for partially evaluating reachability formulae at design time in parallel execution environments. On the other hand, for sequential environments, the quick growth of complexity with the number of states having parametric outgoing transitions may be overly time-consuming (cfr. Equation 4.5).

For a DTMC modeling a software systems, it is usually the case that the transition matrix is 1) very sparse, since each component typically interact with a limited number of counterparts, and 2) may present regular topological patterns, reflecting the design rational.

Formulating the computation of the elements b_{ij} as the solution of a linear system of equations allows the exploitation of state-of-the-art heuristics and provide a noteworthy speed-up in the actual execution time.

Recalling the definition of inverse of a square matrix A, $A \cdot A^{-1} = I$. Hence, the i-th column of the matrix A^{-1} corresponds to the solution the following system of linear equations:

$$A \cdot v = e_i \tag{4.6}$$

where e_i is the i-th column of the identity matrix, i.e. a column vector having all zero elements but for the i-th that is 1, and v is the unknown vector corresponding to the i-th column of A^{-1}. Since to solve Equation (4.2) it is needed to compute the entries of the i-th row, not column, of the matrix $N = (I - Q)^{-1}$, it is possible to exploit a property of the transpose of invertible matrices, namely $(A^{-1})^T = (A^T)^{-1})$, to compute those entries.

Indeed, the i-th row of $(I-Q)^{-1}$, corresponds to the i-th column of $((I-Q)^{-1})^T)$, which is in turn equal to the i-th column of $((I - Q)^T)^{-1}$, by the just mentioned property.

The problem of calculating the row of the matrix N and, through (4.2), of B is thus reduced to the solution of a linear system of equations.

Complexity issues. Solving linear systems of equations is a well-studied mathematical problem, even though most of the algorithms concern numerical solutions and cannot deal with symbolic parameters [170]. The most popular algorithms to solve linear equation systems embedded in probabilistic model-checkers are iterative ones [161,176], which can efficiently solve even large systems with the desired precision in the final result and without requiring a large amount of memory.

In the WM approach it is not possible to adopt the same strategy because iterative methods do not deal conveniently with symbolic parameters. Indeed, the presence of unknown parameters makes hard to assess the convergence of the solution. For this reason *direct* method have been adopted, optimized for the solution of sparse linear systems [53].

Since [70], the WM approach is supported by a solver based on structured Gaussian elimination and Markowitz pivoting [53]. Structured Gaussian elimination is a variation of the widely used method to triangularize linear systems that allows to reduce the solution of a large sparse equation system to the solution of a small dense one. This collapse can significantly reduce the size of the system to be actually solved. A core element of structured Gaussian elimination is the strategy used to select the order in which elements of the original system will be eliminated. In fact, each elimination step may reduce the sparsity of the obtained system, reducing in turn the global effectiveness of the method. This problem is known as *fill-in*. In order to reduce the fill-in during the elimination steps, Markovitz pivoting has been adopted as the selection strategy of the next element to be eliminated. Other strategies can be more efficient for specific cases but their discussion is beyond the scope of this paper. The interested reader may refer for example to [53].

Finally, in order to avoid any loss of accuracy during intermediate computation steps, the WM solver uses infinite precision rational numbers for all the numeric values appearing in the models. All the mathematical procedures have been implemented in Maple 15[3].

An empirical evaluation of the design-time phase complexity will be provided in Section 4.6.

Example 4.2.1 (Reachability Analysis) Following from Section 3.1.1, requirement **R1** asserts that the probability of reaching the absorbing state s_8 must be greater than 0.999.

Applying the partial evaluation algorithms defined in this section, it comes out that the parametric expression corresponding to the probability of reaching state s_8 is:

$$\begin{aligned} f(k,w,x,y,z) = &-.7\cdot w - y - .144375\cdot k + 1 \\ &+ .7\cdot y\cdot w - .7\cdot y\cdot x\cdot w + .144375\cdot z\cdot k \\ &+ .144375\cdot y\cdot k + .7\cdot x\cdot w - .144375\cdot y\cdot z\cdot k \end{aligned} \tag{4.7}$$

As soon as the monitors provide the current value of the mode parameters, the corresponding value of f can be compared with the threshold 0.999 in order to verify the satisfaction of **R1**.

4.2.1.3 First-Step Analysis

The probability of being absorbed into a state s_j given that the process started in state s_j can be restated as the probability of hitting state s_j in a finite time, again given that the process started in s_i. Let $C \subseteq S$ be the set of absorbing states, $\bar{t} = inf\{k \geq 0 : \pi[k] = s_j\}$ be the first hitting time of s_j.

The function $u^* : S \rightarrow [0,1]$ assigning to each state the probability of being absorbed into s_j at the (finite) time $\bar{t}$ can be defined as:

[3]`http://www.maplesoft.com`

$$u^*(s_i) = Pr(\pi[\bar{t}] = s_j \wedge \bar{t} < \infty \mid \pi[0] = s_i) \tag{4.8}$$

For each transient state s_i, by conditioning on the first step of the process, $u^*(s_i)$ can be obtained by the following system of equations:

$$u^*(s_i) = \sum_{s_k \in S} p_{ik} \cdot u^*(s_k) \qquad s_i \notin C \tag{4.9}$$

subject to the boundary conditions on absorbing states:

$$u^*(s_y) = \begin{cases} 1 & \text{if } s_y = s_j \\ 0 & \text{if } s_y \in C - \{s_j\} \end{cases}$$

The linear system in (4.9) is obtained by a direct application of the Markov properties, i.e. the first step of the process does only depend on the initial state, the second step only on the first state, and so on. The extension to reachability of a set target absorbing states is straightforward.

First-step analysis underlies also the core routines of most probabilistic model-checkers, with a slightly different problem formulation [127].

Solving (4.9) in presence of symbolic parameters requires essentially the same process described in Section 4.2.1.2, and thus the same considerations apply about complexity.

A first matrix formulation of the problem would be $u^* = P \cdot u^*$ on C, subject to $u^* = 1$ on s_j and $u^* = 0$ on $C - \{s_j\}$. A mathematically identical form, free of boundary conditions, is:

$$u^* = f + \hat{P} \cdot u^* \tag{4.10}$$

where $\hat{P}(x,y) = (P(x,y) : x, y \notin C$ is the restriction of P to transitions between transient states, and $f[x] = p_{xj}$.

It is immediate to observe that f in Equation (4.10) is the column of matrix R (see Equation 2.1) corresponding to state s_j and that $\hat{P}$ is the matrix Q. Hence:

$$u^* = f + \hat{P} \cdot u^*$$

$$(I - \hat{P}) \cdot u^* = f$$

$$u^* = (I - \hat{P})^{-1} \cdot f$$

proving that $u^*[i]$ is exactly the element b_{ij} computed in Sections 4.2.1.1 and 4.2.1.2.

4.2.2 Extending to the Entire PCTL

Reachability of an absorbing state is the most widely used class of PCTL properties [93]. Nonetheless there are relevant requirements that cannot be expressed within this class.

In this section, algorithms will be provided to extend the WM approach to handling all the remaining fragments of PCTL.

Flat unbounded Until $\mathscr{P}_{\bowtie p}$ $(\phi_1\ U\ \phi_2)$ formulae will be the first class of properties discussed. Being flat, neither ϕ_1 and ϕ_2 nor their sub-formulae can contain the operator $\mathscr{P}_{\bowtie p}(\cdot)$. This class is indeed a superclass of the fragment $\mathscr{P}_{\bowtie p}(\diamond\phi)$ studied in the previous section. Afterward, in Section 4.2.2.2, algorithms to verify the bounded operators X and $U^{\leq t}$ will be presented, concluding the verification of the flat fragment. Formulae having nested $\mathscr{P}_{\bowtie p}(\cdot)$ operators will be treated in Section 4.2.2.3.

4.2.2.1 Flat Until Formulae

The core idea for analyzing generic flat until formulae is to reduce the problem to the analysis of equivalent reachability formulae, and then apply the solution procedures already seen. This reduction process passes through the following transformation of the DTMC model.

Starting from a DTMC $D = (S, s_0, \mathbf{P}, L)$ and a flat until formula $\mathscr{P}_{\bowtie p}$ $(\phi_1\ U\ \phi_2)$, a DTMC $\bar{D}$ is derived from D through following procedure:

1. Add two absorbing states s_{goal} and s_{stop}
2. For all the states where ϕ_2 holds, remove all the outgoing transitions and put a single one (with probability 1) toward s_{goal}
3. For all the states where $\neg(\phi_1 \vee \phi_2)$ holds, remove all the outgoing transitions and put a single one toward s_{stop}.

hence the state space of $\bar{D}$ is $S \cup \{s_{\text{goal}}, s_{\text{stop}}\}$; the labeling function of $\bar{D}$ is accordingly extended by the two atomic predicates S_{goal} and S_{stop} holding only in states s_{goal} and s_{stop} respectively. The transition matrix of $\bar{D}$ will be identified by $\bar{P}$.

Theorem 4.2.1 (Flat Until Verification) *$\mathscr{P}_{\bowtie p}\,(\phi_1\; U\; \phi_2)$ holds in state s_i of D iff $\mathscr{P}_{\bowtie p}\,(\diamond\, s_{goal})$ holds in state s_i of $\bar{D}$.*

Proof 4.2.1 *For a path π of D originating in s_i and satisfying the path formula $\phi_1 U \phi_2$ there exists $k \geq 0$ such that $\pi[k] \models \phi_2$ and for all the $0 \leq j < k\ :\ \pi[j] \models \phi_1 \wedge \neg\phi_2$. By construction, there will exist one and only one path $\bar{\pi}$ in $\bar{D}$ such that $\forall 0 \leq j \leq k\ :\ \bar{\pi}[j] \equiv \pi[j]$ and $\bar{\pi}[k+1] \models S_{goal}$. Furthermore, $Pr(\pi) = Pr(\bar{\pi})$ because, by construction, $\forall 0 \leq j < k$ $P(\pi[j], \pi[j+1]) = \bar{P}(\bar{\pi}[j], \bar{\pi}[j+1])$ and $\bar{P}(k, s_{goal}) = 1$.* □

By Theorem 4.2.1, verifying on $\bar{D}$ the property $\mathscr{P}_{\bowtie p}\,(\diamond\, s_{\text{goal}})$ provides the same result as verifying $\mathscr{P}_{\bowtie p}\,(\phi_1\; U\; \phi_2)$. At this point, it is possible to apply the same mathematical procedures defined in Section 4.2.1.

Before proceeding with the remaining fragments of PCTL, in the next two paragraphs two algorithms related to flat until formulae are introduced. The first one exploits first-step analysis to compute the probability of a flat until formulae on the DTMC D directly. The second is matrix-based approach to compute the probability of reaching a transient state, again directly from D.

First-Step Analysis for Flat Until. First-step analysis for flat until formulae $\mathscr{P}_{\bowtie p}\,(\phi_1\; U\; \phi_2)$ is again based on the system of linear equations (4.9), though they can be extended to all the states of D, not only the transient ones:

$$u^*(s_i) = \sum_{s_k \in S} p_{ik} \cdot u^*(s_k) \tag{4.11}$$

then, the following boundary conditions are introduced (let C be the set of absorbing states of D):

$$u^*(s_y) = \begin{cases} 1 & \text{if } s_y \models \phi_2 \\ 0 & \text{if } s_y \models \neg(\phi_1 \vee \phi_2) \\ 0 & \text{if } s_y \in C\ \wedge\ s_y \not\models \phi_2 \end{cases} \tag{4.12}$$

The boundary conditions (4.12) essentially describe the elementary cases of for the computation of u^*. The first and second case present an immediate correspondence to step 2 and 3 of the construction of $\bar{D}$ defined previously. The third case accounts for the possible presence of absorbing states satisfying ϕ_1 but not ϕ_2, thus not included in the second case; from such states there is clearly no chance to satisfy $\phi_1 U \phi_2$ because of their absorbing nature.

The solution of the system of linear equation defined by (4.11) and (4.12) involves symbolic computations in presence of model parameters. Algorithms defined in Section 4.2.1.2 directly apply also in this case.

As a final remark, the numerical routines used for probabilistic model checking in [127] or [15] can be easily matched with equations (4.11) and (4.12), though the solution strategies in those cases are quite different.

Example 4.2.2 (Flat Until) Following from Section 3.1.1, requirement **R2** is an example of flat until formula. The process is indeed required to reach state s_8 without passing through states s_4 and s_6.

According to the construction procedure of the derived DTMC $\bar{D}$ introduce earlier in this section, state s_8 will be connected with probability 1 to s_{goal}, while states s_4 and s_6 are connected with probability 1 to state s_{stop}.

The probability of following a path that satisfied the condition imposed by **R2** can be then computed as the probability of reaching the absorbing state s_{goal} and the result is:

$$\begin{aligned} f(k,w,x,y,z) = {} & 0.7 \cdot x - 0.155625 \cdot y + 0.144375 \cdot z \\ & - 0.144375 \cdot y \cdot z - 0.7 \cdot y \cdot x + 0.155625 \end{aligned} \tag{4.13}$$

Notice that the parameters k and w do not appear in the rhs of Equation (4.13), since they have been actually removed in the construction of $\bar{D}$ and their value is thus irrelevant for the verification of **R2**, as can be intuitively assessed by looking at the DTMC of Figure 2.1.

Reachability of Transient States. An elegant procedure to compute the probability of reaching a transient without any intermediate construction has been presented in [72], based on the theoretical background from [175].

First of all, the probability of reaching a transient state from an absorbing one is trivially 0, while the probability of reaching s_j from itself is trivially 1. For any two distinct transient states s_i and s_j, let f_{ij}^k be the probability that the first hitting of state s_j happens at time k, given that the process started from s_i (recalling the notation in Section 2, X_k is the random variable representing the state of the process at time k):

$$\begin{cases} f_{ij}^{0} = 0 \\ f_{ij}^{k} = Pr(X_k = s_j \wedge \forall 0 \leq y < k \; X_y \neq s_j | X_0 = s_i) \end{cases} \tag{4.14}$$

Let

$$f_{ij} = \sum_{k=1} \tag{4.15}$$

Thus, f_{ij} represents the probability of ever reaching state s_j given that the process started from s_i. Notice that, for every well-formed DTMC (cfr. Section 2.1) $f_{0j} > 0$ because every state of the model has to be reachable from the initial state. Furthermore, since the only recurrent states allowed in a well-formed DTMC are the absorbing ones, $f_{ii} < 1$ for every the transient state s_i.

Though the definition in (4.15) formalizes the probability of reaching a transient state, the computation of its actual value is not straightforward from the definition. On the other hand, in Section 2.1 the fundamental matrix N has been defined, whose entries n_{ij} represent the expected number of visits to the transient state s_j before absorption, given that the process started in s_i. Assuming to know the values of f_{ij}, n_{ij} can be derived by conditioning on whether state s_j is ever visited:

$$\begin{aligned} n_{ij} = & \; E(\text{number of visits to state } s_j \mid X_0 = s_i) \\ = & \; n_{jj} \cdot f_{ij} \end{aligned} \tag{4.16}$$

In other words, the value n_{jj} is the expected number of "returning" visits to s_j given that it is eventually reached from state s_i (cf. [175], pg. 190).

From Equation 4.16, it is immediate to derive:

$$f_{ij} = \frac{n_{ij}}{n_{jj}}$$

Summing up, $s_i \models \mathscr{P}_{\bowtie p} \; (\diamond \;\; s = s_j)$ iff $f_{ij} \bowtie p$. This relationship allows for the use of the matrix-based algorithms provided in Section 4.2.1.1.

4.2.2.2 Flat Bounded Formulae

Flat bounded formulae of PCTL are those having as arguments of the $\mathscr{P}_{\bowtie p}(\cdot)$ operators path formulae involving the Next and Bounded Until operators.

Next. The set of paths to be considered in order to estimate the probability of a path formula $X\ \phi$ in a state s_i is composed by all the 1-step long paths originating in s_i. Under the hypothesis that ϕ is flat, the states that satisfy ϕ can be identified once for all at design time. The transition matrix P contains the probability of moving from a state to another in a single step. Hence, computing the probability of reaching, from a state s_i, a state where ϕ holds in 1 step, can be achieved by:

$$Pr(X\ \phi_1) = \sum_{s_j \models \phi_1} p_{ij} \tag{4.17}$$

Bounded Until. Each path originating in a state s_i and satisfying $\phi_1 U^{\leq t} \phi_2$, at a certain step $k \leq t$ will reach a state s_j where ϕ_2 holds, and for all the previous steps ϕ_1 has to hold. Referring to a DTMC $\bar{D}$ constructed as in Section 4.2.2.1, each of those paths corresponds to a path in $\bar{D}$ that exactly at time step $k+1$ reaches the state s_{goal}. Hence, any path of D satisfying $\phi_1 U^{\leq t} \phi_2$ corresponds to a path in $\bar{D}$ being at time $t+1$ in state s_{goal}.

The probability distribution of the states reached after exactly $t+1$ time steps in $\bar{D}$ can be computed by elevating the transition matrix $\bar{P}$ to the $(t+1)$-th power:

$$Pr(X_k \models \phi_2 \wedge k \leq t \mid X_0 = s_i) = \bar{P}^{t+1}(s_i, s_{\text{goal}}) \tag{4.18}$$

Summarizing, $s_i \models \mathscr{P}_{\bowtie p}\ (\phi_1 U^{\leq t} \phi_2)$ on D iff $\bar{P}^{t+1}(s_i, s_{\text{goal}}) \bowtie p$ on $\bar{D}$.

Example 4.2.3 (Flat Bounded Until) Requirement **R3** from Section 3.1.1 involves the evaluation of a flat bounded until formula.

The parametric expression corresponding to the probability of reaching state s_8 within 5 steps is reported in Equation (4.19).

$$\begin{aligned} f(k,w,x,y,z) &= 0.10548 - 0.10548 \cdot y + (0.0231 - 0.0231 \cdot y) \cdot z \\ &+ (0.165 - 0.165 \cdot y) \cdot (0.7 - 0.7 \cdot z) \cdot (1-k) \\ &+ (0.165 - 0.165 \cdot y) \cdot (0.3 + 0.7 \cdot z) \\ &+ (0.7 - 0.7 \cdot y) \cdot (1-x) \cdot (1-w) + (0.7 - 0.7 \cdot y) \cdot x \end{aligned} \tag{4.19}$$

4.2.2.3 Nested Formulae

The analysis of PCTL has been so far restricted to its flat fragment, that is, the set of formulae where the arguments of a $\mathscr{P}_{\bowtie p}(\cdot)$ operator are Boolean combinations of atomic propositions only. The peculiarity of flat formulae is that it is always possible at design time to identify the states where a state formula ϕ holds, and thus generate a parametric expression by means of the procedures defined so far.

In the case of *nested* formulae, that is formulae $\mathscr{P}_{\bowtie p}(\Psi)$ where at least one sub-formula of ψ contains the operator $\mathscr{P}_{\bowtie p}(\cdot)$, some information needed to compute the desired parametric expression may only become available at runtime. For instance, consider requirement **R4** from the example in Chapter 2: the set of states from which an error will eventually occur with probability greater than .95 will only be know at run time, because it depends on the actual value of the model parameters. Thus, the probability of reaching any of this states cannot be computed at design time with the procedures already defined, because it would not be possible to identify the target states. Indeed, to evaluate a formula with nested $\mathscr{P}_{\bowtie p}(\cdot)$ operators, it is required to know in which states its sub-formulae are satisfied. The same consideration can be applied recursively to sub-formulae of a sub-formula, until a flat one is reached that can be directly analyzed.

To deal with this issue, it is required a way to delay at run time the evaluation of a nested formula, when all the knowledge concerning its sub-formulae has been gathered, without loosing the benefits of parametric verification.

Focusing on until formulae, the solution provided in Section 4.2.2.1 is based on the construction of the modified DTMC $\bar{D}$. Such a construction requires to disconnect certain states from their successors and to connect them to either s_{goal} or s_{stop}. Then, for what has been previously explained, the resulting parametric expression would be computed as the reachability of the absorbing state s_{goal} in $\bar{D}$.

In order to delay at runtime the decision about the connection of a state to s_{goal} or to s_{stop}, all is needed is the addition of three more parameters per state. The first will be a coefficient α_i that multiplies all the elements p_{ij} of D. The second and the third are, respectively, two terms $\beta_{i\ \text{goal}}$ and $\beta_{i\ \text{stop}}$ in place of the entries $\bar{P}(s_i, s_{\text{goal}})$ and $\bar{P}(s_i, s_{\text{stop}})$. The three additional parameters can assume values 0 or 1, and their intuitive purpose is the following: assigning 0 to a parameter $alpha_i$ disconnects state s_i from all its successors; assigning 1 to either $\beta_{i\ \text{goal}}$ or $\beta_{i\ \text{stop}}$ connects state s_i to state s_{goal} or s_{stop}, respectively.

Computing the probability of a path $\diamond\, s_{\text{goal}}$ at design time leads to a parametric expression having as variables both the model parameters and the additional parameters α_i, $\beta_{i\ \text{goal}}$, and $\beta_{i\ \text{stop}}$ for each state s_i. At runtime, when information about the sub-formulae of a nested formula becomes available, the value of the additional parameters can be set in order to adapt the expression to reflect the convenient transformation of $\bar{D}$. Applying this procedure recursively on nested formulae allows to keep the benefits of parametric analysis, though it would require at most as many evaluations as the nesting depth of the formulae. Assuming most of the nested formulae to have just a few nesting levels, the impact on runtime complexity would still be limited. Another drawback for run-time analysis of nested formulae is that the resulting mathematical expressions could be longer than in the case of flat formulae due to the presence of more parameters, but the evaluation time is still not comparable with the execution of a model-checking routine for a system large enough.

At design time, the computation of next and bounded until nested formulae follows the same principle described for until ones, and they have to be computed on the model instrumented with the additional parameters α_i, $\beta_{i\ \text{goal}}$, and $\beta_{i\ \text{stop}}$. The adaptation of the mathematical procedure for the Next operator is straightforward. The main issue with this approach is the computational complexity at design time. Indeed, the additional parameters may have a high impact on the execution time of the algorithms from Section 4.2.1. To leverage this issue, parallel implementations may be used on high performance distributed environments, or, for not too large systems, the results for each combination of the α and β parameters can be stored in a direct access table; Notice though that the number of entries of such table would be $O(3^{|S|})$ and the size of each of them would be up to $O(|S|^{\log(|S|)})$ because all the transitions in the model would become symbolic.

4.3 R-PCTL Verification

In this section, the verification of R-PCTL properties on D-MRMs will be discussed. Equations (3.1), (3.2), and (3.3) of page 35 formalize the semantics of the three specification patterns used to express reward-related properties.

Some of mathematical procedures presented in this section are based on the notion of *expected reward* along a set of paths originating from a state s_i. In Section 3.2 this value has been intuitively defined as the sum of the rewards cumulated along each of the paths, weighted by the probability for that path to be taken. Since such a sum may contain infinite terms, it could be unfeasible to compute it directly from its definition.

Applying a first-step analysis, the computation of the expected reward for a (non empty) path originating in s_i can be computed by the following linear equation:

$$r_i = \rho(s_i) + \sum_{s_j \in S} p_{ij} \cdot (\iota(s_i, s_j) + r_j) \tag{4.20}$$

where r_i is the expected reward over all the paths originating in s_i

As stated in Section 2.2, in this thesis only state rewards are considered, since they are expressive enough for all the specification purposes. This is equivalent to state that the function $\iota(\cdot,\cdot)$ in Equation (4.20) is identically 0.

Notice from Equation (4.20) that if a state s_j is absorbing and its state reward $\rho(s_j)$ is greater than 0 the equation has no solutions. Indeed, in such a situation, for all the states from which s_j is reachable the expected reward would be ∞. For this reason, it is here assumed that $\rho(s_j) = 0$ for all the absorbing states s_j.

In the remaining of the section, Section 4.3.1 discusses the verification of unbounded formulae of the class $\mathscr{R}_{\bowtie r}(\diamond\phi)$, while Section 4.3.2 will deal with the bounded operators $I^{=k}$ and $C^{\leq k}$. In Section 4.3.3 a set of special cases of reward analysis will be discussed to show further added value of parametric formulae.

4.3.1 Unbounded Formulae

A formula $\mathscr{R}_{\bowtie v}(\diamond\phi)$ is intuitively true in a state s_i if the expected cumulated reward before reaching a state satisfying ϕ meets the constraint $\bowtie v$.

In order to simplify the exposition, only flat R-PCTL formulae will be discussed, meaning that in path formulae $\diamond\phi$, ϕ can only be a Boolean combination of atomic propositions. The extension to the nested fragment of R-PCTL can be achieved by instrumenting the D-MRM with additional parameters as it has been done previously for nested PCTL formulae in Section 4.2.2.3.

The expected cumulated reward over all the paths satisfying $\diamond\phi$ and originating in a state s_i can be computed subjecting the linear system (4.20) to the following boundary conditions (let $C \subseteq S$ be the set of absorbing states of the D-MRM):

$$r_i = \begin{cases} 0 & \text{if } s_i \models \phi \\ \infty & \text{if } s_i \in C \wedge s_i \not\models \phi \end{cases} \tag{4.21}$$

The rationale behind (4.21) is intuitive: a state s_i satisfying ϕ is the terminal state of a path π that satisfies the path formula $\diamond\ \phi$ and thus the end of the reward accumulation. On the other hand, an absorbing state that does not satisfy ϕ marks a path that will never satisfy $\diamond\ \phi$ and thus contribute to the accumulation of rewards as an infinite cost, for the definition in (3.3).

The solution of (4.20) subject to (4.21) is a rational polynomial expression having as unknowns the model parameters, whether they label transition probabilities or state rewards. For the algorithms to solve this system and their complexity, refer to Section 4.2.1.2.

Summing up, $s_i \models \mathscr{R}_{\bowtie v}\ (\diamond\phi)$ iff $r_i \bowtie v$.

Example 4.3.1 (Unbounded Reward) The parametric verification of requirements **R6**, as defined in Section 3.2.1, leads to the following expression:

$$\begin{aligned} X_{\diamond(7\leq s\leq 9)} = {} & 0.21734375 + 0.084\cdot y\cdot x - 0.084\cdot x \\ & - 0.21734375\cdot y - 0.02165625\cdot z \\ & + 0.02165625\cdot y\cdot z \end{aligned} \tag{4.22}$$

Expected Cumulated Reward Before Absorption. Assuming that for all the absorbing states s_y $\rho(s_y) = 0$, the expected reward before absorption involves only the visits of transient states. From Section 2.1, an entry n_{ij} of the fundamental matrix N represents the expected number of visits to the transient state s_j before absorption, given that the process started in s_i. Since after each visit to s_j the reward $\rho(s_j)$ is gained, let ρ be a column vector with elements $[\rho(s_0), \rho(s_1), \rho(s_2), \dots]$ and $C \subset S$ the set of absorbing states, the expected cumulated reward before absorption can be computed as:

$$X_{\diamond(s_y \in C)} = N \cdot \rho \tag{4.23}$$

The latter equation is equivalent to (4.20) subject to (4.21) restricted to $S - C$. Indeed:

$$r = \rho + Q \cdot r$$

$$(I - Q) \cdot r = \rho$$

$$r = (I - Q)^{-1} \cdot \rho$$

4.3.2 Bounded Formulae

A formula $\mathscr{R}_{\bowtie v}\ (I^{=k})$ is true in a state s_i if the expected state reward at time k meets the bound $\bowtie v$. By the definition of expected value, it can be computed as the sum of the rewards of every state reachable in exactly k time steps, weighted by the probability of reaching it. Recall that the probability of reaching a state s_j from a state s_i in exactly k time steps is the entry (s_i, s_j) of the matrix P^k.

In a more compact way, let ρ be a column vector with elements $[\rho(s_0), \rho(s_1), \rho(s_2), \dots]$. The expected reward $X^{=k}$ can be computed by the following equation:

$$X_{I^{=k}} = P^k \cdot \rho \tag{4.24}$$

where an element $X_{I^{=k}}[i]$ corresponds to the expected reward from state s_i. Hence, $s_i \models \mathscr{R}_{\bowtie v}\ (I^{=k})$ iff $X_{I^{=k}}[i] \bowtie v$.

Finally, a formula $\mathscr{R}_{\bowtie v}\ (C^{\leq k})$ is true in a state s_i if the expected reward cumulated after k time steps satisfies the constraint $\bowtie v$. For the previous considerations, the expected reward gained at the j-th step is exactly $P^j \cdot \rho$. Thus, to compute the cumulated reward up to the k-th step with $k \geq 1$ it is possible to apply the following equation:

$$X_{C^{\leq k}} = \sum_{j=0}^{k-1} P^j \cdot \rho \tag{4.25}$$

When $k = 0$, $X_{C^{\leq k}} = 0$ by definition (3.2). Hence, for all $k > 0$, $s_i \models \mathscr{R}_{\bowtie v}\ (C^{\leq k})$ iff $X_{C^{\leq k}}[i] \bowtie v$.

4.3.3 Special Applications of Reward Analysis

Expected Absorption Time. Rewards can be used to encode a number of properties related to the process. A popular example is the computation of the expected absorption time, that is, the expected number of steps before the process is absorbed. This value can be intuitively computed, for a process started in state s_i, as $\sum_j n_{ij}$. A more elegant formalization assigns to each transient state s_j a reward $\rho(s_j) = 1$ and compute the expected reward before absorption. Besides being more elegant, this solution allows to accomplish the computation through the solution of a linear system of equation, which is usually much more efficient than computing N on a sequential execution environment.

Moment Generating Function of Cumulative Reward to Absorption. Up to this point, all the requirements have been expressed in terms of the expected rewards. On the other hand, a designer may be interested in more information about the entire distribution, not only its expected value. For example, a high variance in the energy consumption may increase the risk of overrun of the power system, though the average consumption is fairly manageable. Fist-step analysis provides a suitable tool for computing the moment generating function of the distribution of the cumulative reward to absorption. Let $C \subset S$ be the set of absorbing states, and T be the first hitting time of an absorbing state, $T = inf\{n \geq 0 : X_n \in C\}$ (as in Section 4.2.1.3). The generating function of the θ-th moments is defined as (cf. [163]):

$$u^*(\theta, s_i) = E\left(e^{\theta \sum_{j=0}^{T-1} \rho(X_j)} \mid X_0 = s_i\right) \tag{4.26}$$

The application of first-step analysis to (4.26) yields:

$$u^*(\theta, s_i) = \sum_{s_j} e^{\theta\rho(s_j)} \cdot P(s_i, s_j) \cdot u^*(\theta, s_j) \tag{4.27}$$

subject to the boundary conditions that $u^*(\theta, s_j) = 1$ for all $s_j \in C$.

Equation (4.27) can also be easily adapted to a matrix formulation.

Added-Value of Parametric Analysis. The only requirement on the function $\rho(\cdot)$ is that is range must be a subset of $[0, +\infty) \cap \mathbb{R}$. This means that after producing the parametric formulae corresponding to a requirement, the actual value of the reward parameters can be evaluate to convenient real functions whose value depends on exogenous measures. For example a function $\mathit{energyCost}(t)$, whose value depends on the hour of the day, can be assigned to the parametric state reward for the servers. This way a two levels analysis can be accomplished to relate the actual cost of energy to the hour of the day the servers are operating. Furthermore, the same expression can be assigned to different parameters, implicitly capturing possible dependencies among their values.

As a final remark, parametric formulae can provide a significant speed-up in applications such as repeated experiments [128], Monte Carlo analysis [103, 152], and search-based engineering [3, 148, 172, 190].

All of these analysis can be accomplished without re-computing the parametric formulae, but simply assigning different values to its parameters.

4.4 Sensitivity Analysis

The parametric closed formulae obtained in Setions 4.2 and 4.3 are not only a suitable tool for run-time efficient quantitative verification, but also for *sensitivity analysis*.

Roughly speaking, the purpose of sensitivity analysis is to assess how the value of a global property of a system is affected by changes in the quality of its parts. This assessment is valuable in different scopes, such as:

- **Decision making support**: If limited resources are available for system improvement (whether in terms of money or time) it would be better to focus on improving the parts which heavier affect the satisfaction of system requirements.

- **Robustness assessment**: if reasonably wide spreads of the quality scores of parts of the system do not significantly affect the global quality provided, then the system operativeness is robust with respect to variability or uncertainty of those scores.

- **Improvement guidance**: errors are more likely to lead to global failures if they occur in important parts of the system. Sensitivity can assign a degree of importance to each part and drive V&V phases.

A formal definition of sensitivity follows:

Definition 4.4.1 (Sensitivity.) Let $\mathbf{p} = \{p_0, p_1, \ldots, p_n\}$ be the set of parameters of a Markov model D, $f : \mathbf{p} \to \mathbf{A} \subseteq \mathbb{R}$ a function of $\mathbf{p}$ (corresponding to a quantitative property of D).
The sensitivity S of f with respect to $\mathbf{p}$ is defined as:

$$S(f, \mathbf{p}) = (\vec{\nabla} f)_{\mathbf{p}} = \left[\frac{\partial f}{\partial p_0}, \frac{\partial f}{\partial p_1}, \ldots, \frac{\partial f}{\partial p_n} \right]$$

For e given operative point $\bar{\mathbf{p}}$, the entry $\partial f / \partial p_i$ evaluated in $\bar{\mathbf{p}}$ yields an insight about how f changes when p_i changes by a small ε. It is then possible to rank the parameters $\{p_i\}$ according to the impact $\partial f/\partial p_i$ they have on the global quality f, supporting this way design decisions[4].

Sensitivity analysis has been performed according to Definition 4.4.1 in several papers. In [41], an analytical procedure is provided for DTMC-based reliability analysis. Reliability has been formalized as the probability of reaching an absorbing "success" state of the DTMC. Sensitivity is then computed with respect to the probability of specific transitions, by algebraic operations on the transition matrix. A similar approach is presented in [84], where the sensitivity is computed not only for reliability but also for a measure of the response time. In [50] and [68] reliability analysis is extended to describe also error propagation among components, and the sensitivity of reliability qualities is computed with respect to the probabilities that a component experiences an unrecoverable failure or introduces an error in the data-flow that will be propagated to other components.

Using the WM approach to generate closed-form expressions for relevant quality metrics make the computation of sensitivity far more efficient than applying the aforementioned approaches, allowing its computation even at run time. The other side of the coin remains the design-time effort of WM, though it has to be done only once.

Furthermore, sensitivity can be computed in closed-form as well, paving the way for a global analysis of S. A closed-form of S can also be used for reverse sensitivity analysis, i.e. the identification of operative conditions that are particularly sensitive to specific parameters. Such information could allow to design mechanisms that keep the system in a "safe" region, where its properties are less sensitive to uncertainty or variability of external parameters.

The main limitation of sensitivity analysis concerns the dependency on the actual values of ε. Indeed, sensitivity is the more accurate the more ε gets closer to 0. More precisely, from Taylor's theorem [147], the error is in the order of $O(|\varepsilon|^2)$ (as $\varepsilon \to 0$). The closed-form expressions for both f and S, allows to compute the error ξ as an analytic function of ε by computing the distance between f and its first-order Taylor approximation $\tilde{f}$:

[4]Notice that sensitivity is just an index. It does not account for feasibility of a change, nor on dependencies among parameters. For example, if a parameter corresponds to the probability of a transition, a designer must consider that a change in its value has an impact on the other transitions originating from the same state and proceed accordingly.

$$\begin{cases} \xi = f(\bar{\mathbf{p}}+\varepsilon) - \tilde{f}(\bar{\mathbf{p}},\varepsilon) \\ \tilde{f}(\bar{\mathbf{p}},\varepsilon) = f(\bar{\mathbf{p}}) + (\vec{\nabla} f)_{\bar{\mathbf{p}}}(\varepsilon) \end{cases} \tag{4.28}$$

The error ξ can be expressed in a closed-form and analyzed on the entire domain of $\mathbf{p}$ (with the possible exception of the singular points of f corresponding to the zeros of its denominator).

Uncertainty and Perturbation analyses. To conclude this section, two alternative techniques for the impact analysis of uncertainty and variability are briefly reported [99].

In *uncertainty analysis*, a probability distribution is associate to the parameter space to describe the uncertainty about their values:

$$F_{\mathbf{p}}(\mathbf{x}) = Pr(\mathbf{p} \leq \mathbf{x}) = Pr(p_0 \leq x_0 \cap p_1 \leq x_1 \cap \ldots \cap p_n \leq x_n)$$

Now, since $\mathbf{p}$ is considered as a random variable, also the algebraic transformation $f(\mathbf{p})$ will be a random variable. The aim of uncertainty analysis is to derive $F_f(x) = Pr(f(\mathbf{p}) \leq x)$. It is in general impossible to obtain an analytical expression for $F_f(\cdot)$ [181]. Monte Carlo simulation has been used to approximate in several researches to approximate $F_f(\cdot)$ [106].

In [152], uncertainty analysis is applied for DTMC-based software reliability analysis. For example, the failure probability of an embedded component may be a function of the random variable describing the environment temperature. Monte Carlo simulation is applied by sampling from the temperature distribution, deriving the failure probability of the component, and then computing the global system reliability through model-checking. Considering that the number of samples needed to even small system is at least in the order of 10^3, replacing the standard model-checking with the evaluation of the analytical expression produced by the WM may produce a significant saving of time[5].

Finally, *perturbation analysis* tries to analyze the deviation $\mid f(\bar{\mathbf{p}}) - f(\bar{\mathbf{p}}+\varepsilon) \mid$ given a bound on $\mid \varepsilon \mid$. This type of analysis has been performed analytically in Probability theory [59], besides its suitability for Monte Carlo simulation. If a quantitative property can be analyzed through the WM approach, the availability of a closed-form

[5]This idea has been discussed with the authors of the paper and is underway at the time this thesis is being written.

expression for f makes perturbation analysis efficient and quite informative.

4.5 Related Work

Quantitative verification at run time is necessarily subject to strict time constraints. Though they could be efficient enough for certain application domains (e.g. [38]), traditional model-checking techniques are conceived for design time and are usually time consuming.

Improving the efficiency of the current model-checkers is one of the goals of the research community. On a different direction some approaches have been proposed to tackle the specific issues of run time analysis.

In this section three of them are discussed which are representative of three different verification paradigms applied to the problem of probabilistic quantitative verification at run time: *incremental analysis*, *parameter space exploration*, and *parametric model-checking*. The last approach will be treated at a higher level of details because of its closeness to the WM approach proposed in this chapter.

4.5.1 Incremental Verification

Incremental approaches are in general composed of two phase: 1) the impact of a change in the artifact is localized, and 2) results from previous analyses are reused as much as possible in order to avoid unnecessary re-computation.

Incremental analysis approaches for non probabilistic systems have been proposed that mostly focus on improving the generation or the exploration of the state space of the model by identifying which previous results are still valid after a change and which have to be recomputed [49, 94, 102, 124, 136, 179].

Concerning incremental quantitative verification of probabilistic models, the only two approaches have been published so far.

In [130,135], the probabilistic models under analysis are discrete-time Markov Decision Processes (MDPs), which are a super set of the D-MRMs, while the target property is the reachability of a set of target states[6]. In the target usage scenario for this technique, the model has to be analyzed repeatedly, after a few transition probabilities change.

The first step of the proposed analysis procedure is to partition the Markov model into its maximal strongly connected components

[6]An MDP can be seen as D-MRM where besides probabilistic transitions, non deterministic ones are allowed.

(SCCs), allowing, on a first hand, a speed up in the first analysis, as shown in [45]. Indeed, roughly speaking, SCCs can be analyzed in isolation and then the local results can be combined to obtain the probability of reaching the target states.

When a change occurs, the set of the SCCs that have been directly affected is generated. Then, a search algorithm is applied to identify all the SCCs indirectly involved. This search algorithm is based on the topological order of the SCCs. Indeed, let C be and SCC and $Pre^*(C) \subset S$
C the set of states from which C is reachable; Let also assume that the any of the target states is reachable from C. A change in the transition probabilities included in C may affect the probability of its predecessors to reach the target, but not the one of its successors. This observation allows to define an efficient search strategy that goes through the SCCs of a model and re-analyzed only those which may have been affected by the change.

Furthermore, this analysis procedure is also parallelizable, thanks to the partial ordering among the SCCs: at any step an SCC can be processed independently from the others as long as its successors have been analyzed.

The approach in [151], named Δ *evaluation*, is instead concerned with incremental reliability analysis based on conveniently structured DTMCs. The structure of those model is the one proposed by [41], where each software module is represented by a state of the DTMC, and can fail, making a transition toward an absorbing failure state, transfer the control to another module, or completing the execution by moving toward an absorbing success state. Assuming that a single entry of the transient-to-transient transition sub-matrix Q changes, there is no need to re-compute the reliability of the entire system from scratch, but a few simple arithmetic operations can be used to correct the previous reliability value.

Despite its efficiency, the Δ evaluation can only deal with a single change per time in the matrix Q and does not provide support for generic PCTL properties, but only for the reachability of a specific absorbing state (notice that at least two absorbing states are required for the approach to work correctly [151]).

Finally, a novel approach to the definition of incremental verification procedures will be proposed in Chapter 5 that is not tailored to Markov processes but can be used for the incremental quantitative analysis of software artifacts defined by specific formal languages.

4.5.2 Parameter Space Exploration

This approach invert the perspective of verification: instead of verifying if an parametric MDP satisfies a PCTL property ϕ, tries to synthesize the set of parameter evaluations that make the mode satisfy ϕ.

Though this technique was not explicitly designed for run time analysis, its application is straightforward since a design time computation can in principle explore the whole parameter space and store in a convenient lookup table all the evaluations that make the model satisfy ϕ. This way, when a change occurs at run time, a quick access to the lookup table can provide immediate answer to the verification problem.

However, only two approaches have been proposed so far, each one with its own limitations.

In [76], given a parametric MDP and an evaluation of the model parameters, the procedure fixes all the non deterministic choices to their optimal combination, i.e. the one that maximizes the probability of reaching a set of target states for that evaluation. Such combination of the non deterministic choices is called *optimal schedule* in the context of MDPs. When the optimal schedule has been found for a specific evaluation, the parameter space is explored starting from the given evaluation until the maximum bounded region for which the scheduler is still optimal is found.

Concerning the analysis of a D-MRM, all the transitions are probabilistic, thus only one schedule exists. Given a parameter evaluation that satisfies a flat reachability property, the largest region containing that evaluation is found for which the property still holds. The algorithm does not reveal if any other evaluations outside that region satisfy the property too.

[96] can instead deal with the entire PCTL, still verified on MDPs. The idea is to divide the parameter space into hyper-rectangles such that all the elements of a hyper-rectangle either satisfy the desired property or not. The approach is iterative and keeps partitioning the search space until a minimum size for the regions has been reached.

Since it is in general impossible to cover the parameter space by hyper-rectangles, a part of it may remain undecided. Hence the verification procedure is not complete.

4.5.3 Parametric Model Checking

Parametric model checking pursues the same goal as the Working-Mom approach, i.e. to precompute a closed rational expression corresponding to the probability of satisfying a desired quantitative property. The first approach for parametric model-checking of DTMCs has been proposed in [54]. The main contribution of that seminal work concerned the synthesis of parametric closed formulae through a *state elimination algorithm*, analogous to the one used in automata theory to synthesize regular expressions from finite state automata [104].

More precisely, Daws' algorithm allows to compute a closed mathematical expression corresponding to flat reachability property. As already explained in Section 4.2.1, this corresponds to computing $Pr(\mathit{true}\ U\ \phi)$, with the further constraint that ϕ cannot contain any instance of the operator $\mathscr{P}_{\bowtie p}(\cdot)$.

For it shares an approach similar to the WM, Daw's algorithm for parametric model-checking will be deeply presented in this section in order to allow a fine comparison, both theoretical and empirical. In particular, the next section describes the Daw's algorithm for reachability analysis. Afterward, in Section 4.5.3.2, its extension for the analysis of a subset of rewards formulae will be discussed. A comparison with the WM approach is discussed in Section 4.5.3.3 and performed on a large set of test cases in Section 4.6.

4.5.3.1 Flat Reachability Analysis.

The core idea of Daws' algorithm is to consider the probability values labeling DTMC transitions as letters of an alphabet. Under this interpretation the DTMC can be seen as a finite state automaton for which it can be synthesized a variant of the regular expressions by adapting the well-known state elimination algorithm [104]. Such variants of the regular expressions are named stochastic regular expressions (SREs) [54] and actually correspond to rational mathematical expressions. The construction of SREs corresponding to the evaluation of flat reachability formulae on a DTMC is addressed by the first part of this section.

Given a flat reachability formula $\mathit{true}\ U\ \phi$ and a well-formed DTMC D, it possible to identify the set of states T of D that satisfy ϕ, i.e. the *target* states.

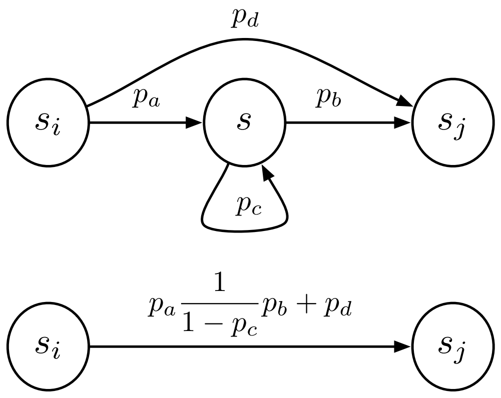


Figure 4.2: SRE synthesis algorithm.

In order to simplify the exposition, assume for now that all the target states are absorbing.

First of all, all the states (and the corresponding transitions) from which it is not possible to reach any of the target states are pruned out. The resulting model may no longer be a DTMC, since the elimination of a subset of the transitions may lead to sub-stochastic states (for which the sum of the outgoing probabilities is lower than 1). Nonetheless the reduced model preserves all the information needed for the computation of the reachability formulae (a proof of correctness can be found in [95]).

The algorithm proceeds by eliminating all the states of the reduced model but the targets and the initial state. A state elimination step is described in Figure 4.2. When eliminating state s, the algorithm considers all the pairs (s_i, s_j) where s_i is a direct predecessor of s and s_j is a direct successor of s. When eliminating s, the transition probability from s_i to s_j is increased by a term representing the probability of reaching s_j from s_i through s. Such a term is essentially the sum of the probabilities of all the possible paths and can be computed by iterating on the length k of a path:

$$\sum_{k=0}^{\infty} p_a p_c^k p_b = \frac{p_a p_b}{1-p_c} \tag{4.29}$$

The state elimination terminates when the model is composed of the initial state s_0 directly connected to each of the target states. Each of these transitions will be labeled by an SRE representing the probability of reaching the specific target state. The value of $Pr(\mathit{true}\ U\ \phi)$ is just the sum of all those SRE.

As it can be guessed from Figure 4.2, an SRE is essentially a rational expression, whose numerator and denominator are polynomials having as unknowns the labels of DTMC transitions.

In order to generalize the approach to deal with transient target states too, it suffices to pre-process the DTMC by making all the target states absorbing. Indeed, a formula $(\mathit{true}\ U\ \phi)$ is satisfied by a path as soon as it firstly reaches any of the states in which ϕ holds, hence its satisfiability is not affected if the states in which ϕ holds are made absorbing. Turning a transient state into absorbing could make other states unreachable from s_0; such unreachable states have to be pruned out in order to regain a well-formed model.

In [95], Daws' algorithm has been implemented in the tool PARAM. An effective improvement provided by PARAM to the original algorithm of [54] consists in replacing the transition labels corresponding to numeric transitions by their actual value after each state elimination step. This allows to exploit arithmetic simplification of the intermediate SREs that can significantly speed-up both memory consumption and subsequent mathematical operations due to state eliminations, as shown in [95].

The result of executing PARAM is a closed rational expression having as variable only the symbolic parameters of the model, since numeric ones have been already evaluated by the tool. Such an expression can then evaluated at runtime, as for the WM approach.

4.5.3.2 Cumulative Rewards Analysis

A second major contribution of PARAM with respect to Daws' algorithm is its extension to deal with D-MRMs. Given as input a D-MRM D and a set of target states T, PARAM is able to compute the expected cumulative reward until a state in T is reached. This type of properties is formalized by formulae like $\mathscr{R}_{\bowtie v}\ (\diamond\phi)$, as discussed in Section 4.3.1.

The algorithm is again based on the state elimination procedure. Considering the pairs (s_i, s_j) of direct predecessors and direct successors of a state s, respectively, the goal is to obtain the transition reward $\iota(s_i, s_j)$ for the new transition from s_i to s_j after eliminating

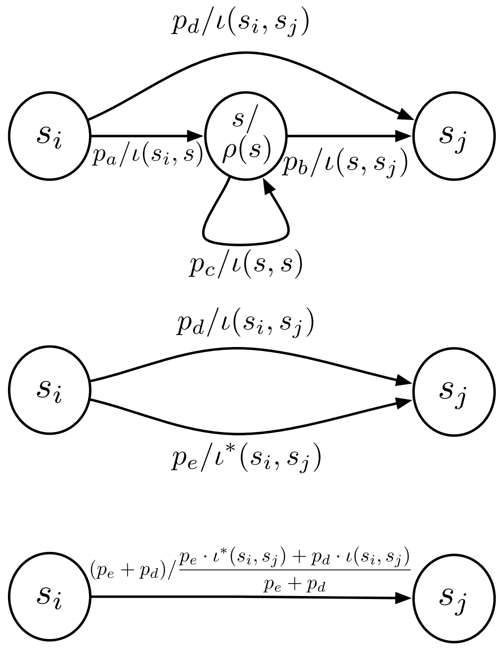


Figure 4.3: State elimination for D-MRM.

s. A step of this state elimination procedure is described in Figure 4.3, where a label p/r represents either the transition probability and the transition reward ι or the state name and its state reward ρ, respectively.

The value of p_e is computed as for reachability analysis as $\frac{p_a p_b}{1-p_c}$, while the value of r_e can be computed as the sum of the reward accumulated over all the possible paths from s_i to s_j through s as (with respect to the path length k):

$$\begin{aligned} r_e &= \sum_{k=0}^{\infty} (p_a p_c^k p_b) \cdot (\iota(s_i,s) + \rho(s) + (\rho(s) + \iota(s,s)) \cdot k + \iota(s,s_j)) \\ &= \iota(s_i,s) + \rho(s) + \iota(s,s_j) + \frac{p_c}{1-p_c}(\rho(s) + \iota(s,s)) \end{aligned} \quad (4.30)$$

The proof of correctness of the algorithms reported in this section can be found in [95].

As for reachability analysis, the resulting expected accumulated reward is again a rational expression with numerator and denominator being polynomials having as unknowns the symbolic parameters of the D-MRM, whether transition probabilities or rewards. The evaluation of such an expression at runtime requires to replace the parameters with the numeric values coming from monitors, as for the WM.

4.5.3.3 Comparing with the WorkingMom.

The comparison between the WM and the Daw's algorithm, enhanced by PARAM, is discussed from two perspective: the expressiveness of the properties analyzable by the two approaches and their design time complexity. An empirical comparison of the derived tools is instead proposed in Section 4.6.

Concerning the expressiveness of the supported logic fragments, PARAM has been designed to deal only with flat reachability properties and unbounded rewards. The WM approach can instead cover the entire R-PCTL, providing mathematical procedures for each of its fragments.

Concerning the design time complexity an assessment of the Daw's algorithm is required first. Analogously to regular expressions on finite state automata, the size of a SRE can grow as $O(n^{\log(n)})$, where n is the number of states of the Markov model [91]. Such long expressions may take time to be manipulated at each state elimination step and may require a high memory consumption when the size of the model growths. The number of state elimination steps are in the order of $O(n^3)$, as it can be easily proved [95], but the actual time each of them takes heavily depends on the size of the operands involved in the mathematical operations.

Though in the worst case $O(n^{\log(n)})$ constitutes a complexity lower bound for computing SREs, in realistic software models most of the transitions can be assumed to be labeled by numeric values [72]. Hence, by this assumption, instead of computing the full SRE taking transition labels as literals, PARAM intertwines the state elimination algorithm and the partial evaluation of numerical terms appearing in SREs. In other words, at each step of the state elimination algorithm, the numeric labels are treated as numbers, thus allowing for the arithmetic simplification of intermediate results.

This induces a significant saving in the size of intermediate rational function representations, and hence an improvement in the actual computation time, as empirically shown in [95].

On the other hand, if a state having symbolic outgoing or incoming transitions is eliminated in one of the first steps, all the subsequent operations will involve polynomial operands. Thus the improvement of PARAM could be more significative in terms of memory saving than in execution time, that grows quickly with the number of model parameters, as will be shown in Section 4.6.

Concerning the algorithms used by the WM approach, they belong to two different families (cf Section 4.2): based on matrix algebra or on the solution of a linear system.

Concerning matrix-based algorithms, their complexity, as defined in Equation (4.5), is $O(\tau^c \cdot t^3)$, where τ is the average number of outgoing transitions from a state, c is the number of states having at least a parametric outgoing transition and t is the number of transient states. Assuming $t \approx n$, the complexity of these algorithms may on a first hand seem higher than the one provided by PARAM. On the other hand, the t^3 operations are in this case numeric operations. Hence their actual execution time is much slower then the one required to operate with polynomials. However, if the number c of states having parametric outgoing transitions growths, the complexity of the matrix based algorithms growths exponentially with it, making the approach unfeasible. Another problem of the matrix based approach concerns the memory required to store intermediate results. Indeed the algorithm is based on c successive Laplace expansions, and each of them requires the computation and storage of approximately τ sub matrices.

Concerning equation-based algorithms, their complexity is theoretically the same as the one of PARAM ($O(n^3)$). However, in most practical cases the equation systems obtained from the analysis of software models are 1) very sparse, and 2) usually present some rationale in the connections among the states. The first observation justify the adoption of the fill-in reduction techniques sketched in Section 4.2.1.2. These techniques significantly reduce the actual computation time as well as the demand of memory, though in terms of memory consumption PARAM is in general more efficient than the WM because it has to store less intermediate result. The empirical comparison of the two tools provided in Section 4.6 will give a glimpse of the actual speed-up achievable by means of the WM solution strategy.

The second observation can be exploited too in order to speed up the execution time. Indeed, regular topologies, such as banded systems, support the application of faster solution algorithms [170].

4.6 Empirical Evaluation

This section discusses a set of experiments conducted to empirically assess the design time effort required by the WorkingMom approach. In order to provide a basis for comparison, the same test cases have been analyzed through the PARAM model-checker too.

All the test cases have been generated randomly and are well-formed models. The algorithm used for the generation is available online[7], as well as the full data set for the experiments. Each test case in the data set is identified by the seed used to initialize the random generator, to make all the experiments replicable.

Each test case has exactly two absorbing states and five outgoing transitions from each state. The properties analyzed in this chapter are only of the forms $\mathscr{P}_{=?}(\diamond\phi)$ and $\mathscr{R}_{=?}(\diamond\phi)$, where ϕ uniquely identify one of the absorbing states. These types of properties have been chosen for two reasons: first, they stress the core routines for all the unbounded properties (cf. Sections 4.2 and 4.3); second, they can be analyzed also by PARAM, which does not support the full R-PCTL, thus a comparison is possible.

The version of PARAM used for the tests is the 1.8 for 64 bit processors. The binary distribution has been downloaded from the official website[8]. PARAM uses by default a bisimulation preprocessing in order to reduce the size of the input model [116]. Since the focus of the comparison is on the verification algorithms only, and considering that the same preprocessing can be applied before the WM too, bisimulation has been disabled by using the "–no-lump" flag for command line invocation of PARAM.

The execution time for PARAM is reported as measured by the tool itself, launched with the statistics flag enabled. The execution time of the Maple implementations of the WM algorithms has been measured using the *time[real]()* built-in function to record the start and the end of the execution, according to the official directives of the tool. The execution time reported here does not include the warm up of the tools, that is anyway independent from the specific instances and negligible with respect to the analysis time.

The execution environment for all the experiments is a Dual Intel(R) Xeon(R) CPU E5530 @ 2.40GHz with 8 Gb of ram, running

[7] `http://filieri.dei.polimi.it`

[8] `http://depend.cs.uni-saarland.de/tools/param`

GNU Linux Ubuntu Server 11.04 64bit. All the tests reported in this section did not overrun the available memory.

The first set of experiments analyzes the matrix based algorithms and their dependence on the number of states and the number of model parameters. The results are reported in Section 4.6.1 and compared with PARAM. The second set of experiments analyzes the design time efficiency of the equation based algorithms with respect to the same dimensions of the problem, by comparing the WM implementation both with PARAM and the built-in solver of Maple 15; the results are discussed in Section 4.6.2. Finally, in Section 4.6.3, a set of experiments has been conducted to stress current implementation of the WM and to provide a realistic assessment of its execution time.

4.6.1 Matrix Based Algorithms

The execution of the matrix based algorithms is highly memory consuming. Despite the temporal efficiency, the size of the test models has been limited to 30 states in order to make the experiments feasible within the memory bound of the execution environment.

For every pair *(number of states, number of parameters)*, 50 random cases have been analyzed.

Figure 4.4 reports the average execution time for the test suite, while Figure 4.5 reports the maximum execution time measured.

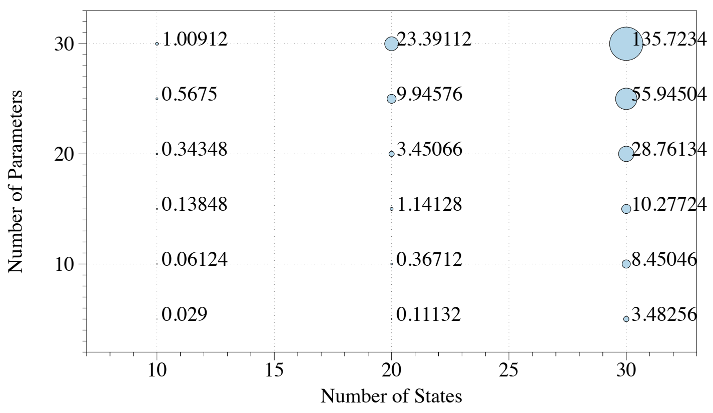


Figure 4.4: Matrix based approach: average execution time (s) vs number of states / number of parameters.

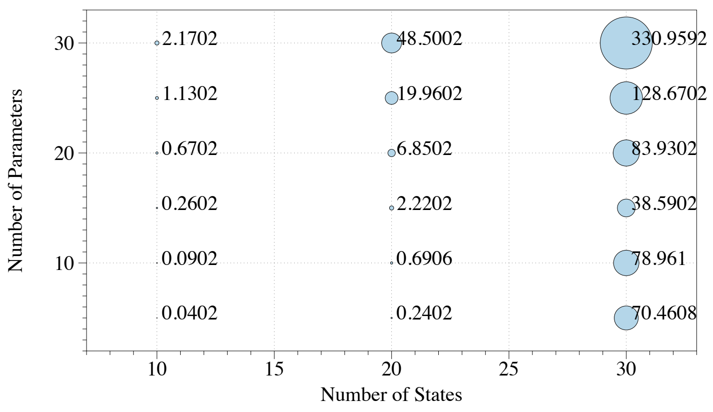


Figure 4.5: Matrix based approach: maximum execution time (s) vs number of states / number of parameters.

The choice of reporting both the average and the maximum execution time is due to the high variance of the results population.

Indeed, the topology of each single model affects the actual execution time in way hard to predict. Recalling the algorithm of Section 4.2.1.1, specific topologies may reduce the number of non zero cofactors, speeding up the actual execution time. This phenomenon is intuitively more relevant in presence of a large number of model parameters because in such case the Laplace expansion of the parametric rows leads to smaller numerical cofactors; due to the sparsity of the matrix, such small cofactors are more likely to be trivially singular, and this condition is efficiently identified by Maple.

On the other hand, the relation between the average execution time and the maximum one, seems in this case to be essentially linear. In particular the worst cases took at most twice the time of the average cases. All the samples have been analyzed in reasonable time ($<$ 6 minutes), though the largest ones almost saturate the available memory $<$ 8Gb.

The growth of the execution time is quite regular, with respect to both the number of states and the number of parameters.

The theoretical complexity described in Equation (4.5) is empirically verified in Figure 4.6. On the x-axis is reported the complexity index $O1 = \tau^c \cdot (n-c)^3)$ computed for each sample set (where τ is the average number of transitions originating in a state, in this section constantly 5; n is the number of state in the model; c is the number of states having at least one parametric outgoing transition).

The polynomial fitting of the empirical data with respect to the complexity index $O1$ reported the following relation:

$$17.686 + 1.4706e - 6 \cdot O1$$

with a correlation between the estimated mode and the data set of 0.93103 and the determination index R^2[9] equals to 0.86681.

[9]In statistics, R^2 is a quality index for models whose main purpose is the prediction of future outcomes on the basis, in this case, of the measured execution times. For its use in this section, $R^2 \in [0,1]$ and a perfect fitting between them model and the data would correspond to $R^2 = 1$.

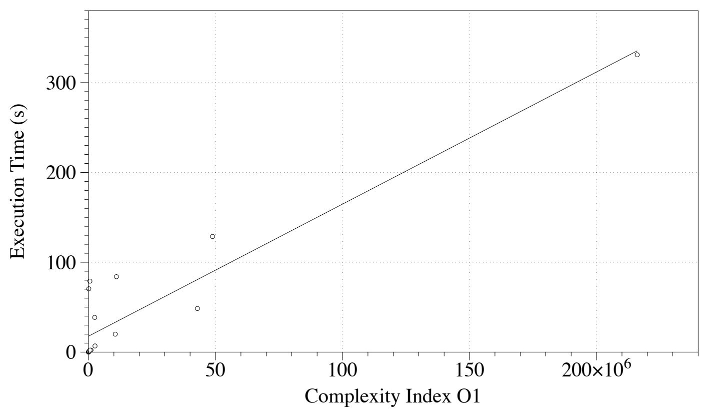


Figure 4.6: Matrix based approach: empirical validation of the complexity assessment in (4.5).

The verification of the same sample cases with PARAM was not always possible because the execution time of the model checker when the models contain more than 10 parameters drastically increases. Considering the execution time of the matrix based algorithms above reported, all the runs of PARAM taking more than 5 hours have been interrupted. For this reason Figures 4.7 and 4.8 have the y-axis truncated at 10.

By looking at Figures 4.7 and 4.8, three observations can be proposed. First, the execution time of the matrix based approach is significantly smaller than the execution time of PARAM, both in the average and in the worst case. Second, there is a higher variability in the execution times of PARAM verifications, as can be verified by the ratio between the maximum and average value for each sample set (up to 360). Third, there is a monotonic trend of the execution time of PARAM with respect to the number of parameters, but this is not the case for the number of states, at least in the average case (cf. Figure 4.7).

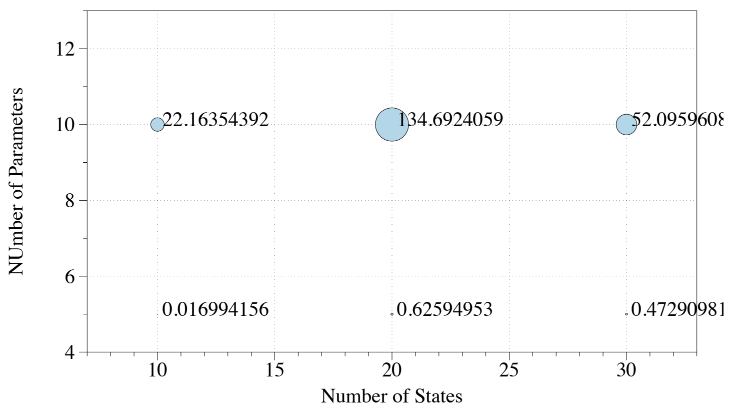


FIGURE 4.7: PARAM: average execution time (s) vs number of states / number of parameters.

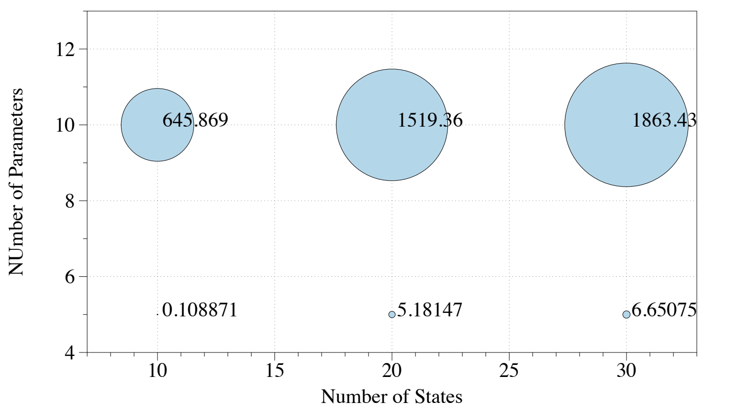


FIGURE 4.8: PARAM: maximum execution time (s) vs number of states / number of parameters.

4.6.2 Equation Based Algorithms

In this section, equation based algorithms defined for the WM approach are compared with both PARAM and the solution of the linear equation systems by means of the built-in solver provided by Maple 15.

Due to the high variability noticed in the result sets of the three analyzers, the plots in this section reports both the average execution time, as a thick black line, and the maximum measured execution time, as a dashed thin line.

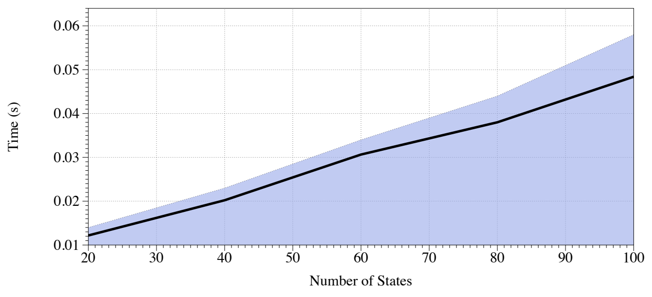
In Figure 4.9 the reachability of an absorbing state has been analyzed. All the input models have exactly 5 parametric transitions. 25 samples have been analyzed for each point in the graphs. The three tools provide reasonable performances, though the WM and the Maple built-in solver are quite faster. There is also a regular monotonic trend in the performance of the equation based procedures, while PARAM presents a higher variability.

In Figure 4.10, the same reachability property has been analyzed, this time for random input models having exactly 10 parametric transitions. 25 samples per point have been analyzed, as before. In this test suite the performance of the three tools is no longer comparable, with the equation based solvers running in seconds, while PARAM took up to 5 hours. Furthermore, for models as large as 100 states, PARAM always took more than 5 hours and the corresponding records have been discarded; this is the reason for the 0 execution time reported on the graphs.

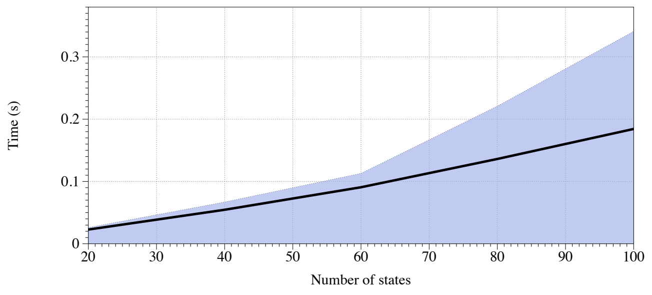
Figure 4.11 shows instead the execution time of the three solvers analyzing the property $\mathscr{R}_{=?}(\Diamond\phi)$, again with respect to the number of states of the input models. Each input model has exactly 7 parameters, 2 of which are parametric rewards; 1 absorbing states; 5 outgoing transitions for each transient state. For each input size, 50 samples have been taken.

Observing Figures 4.9, 4.10, and 4.11 the gap in the performance of the three solvers is evident as soon as the complexity of the model grows, either in the number of states or in the number of parameters. With 100 states and 10 parameters PARAM took almost always more than 5 hours to verify the reachability property. The equation based solvers are can perform the same tasks in tens of seconds.

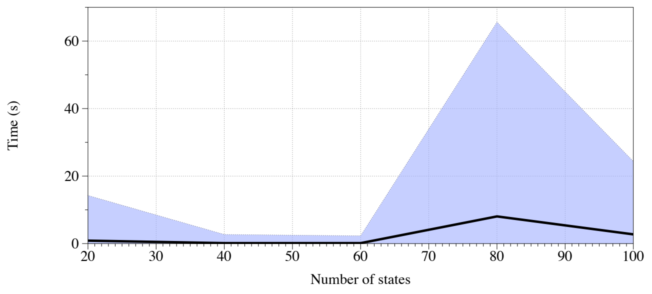
On the other hand, no significant difference can be noted between the Maple built-in solver and the WM implementation at this



(a) Equation based algorithms of WM.



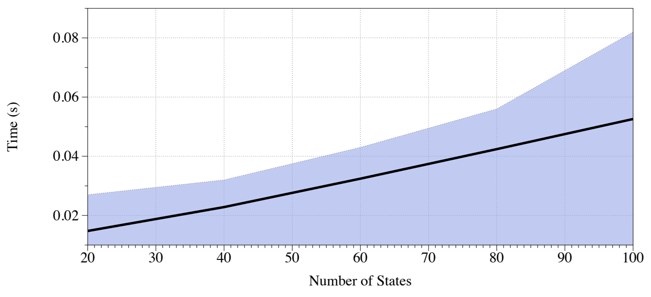
(b) Maple built-in solver.



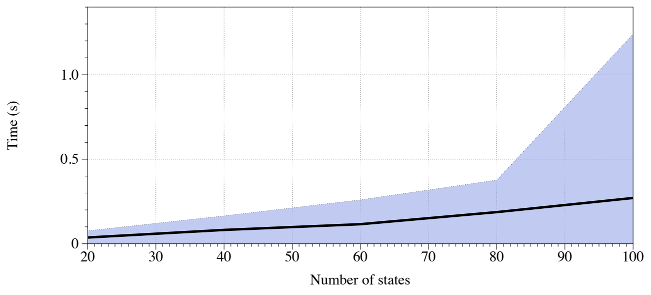
(c) PARAM.

FIGURE 4.9: Execution time vs number of states: flat reachability, 5 parametric transitions.

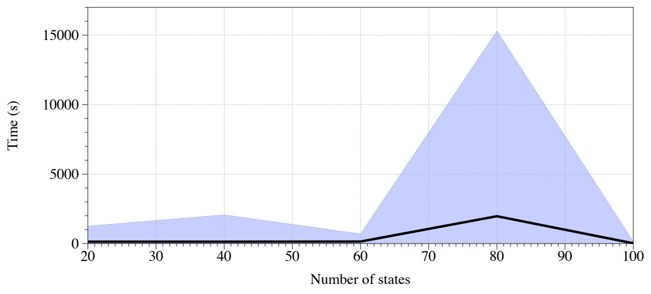
level of complexity of the input models. In the next section the two



(a) Equation based algorithms of WM.



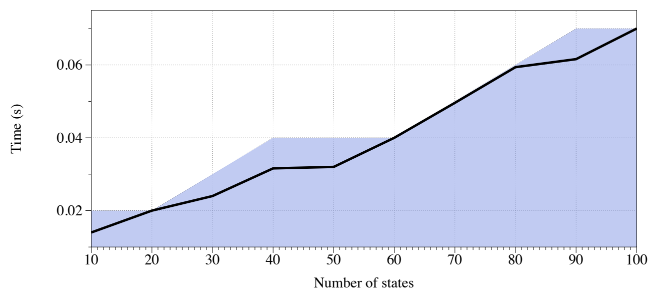
(b) Maple built-in solver.



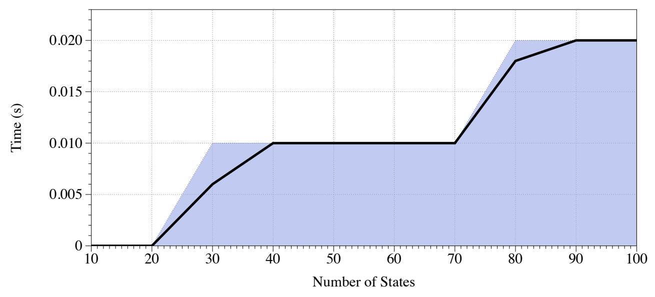
(c) PARAM.

Figure 4.10: Execution time vs number of states: flat reachability, 10 parametric transitions.

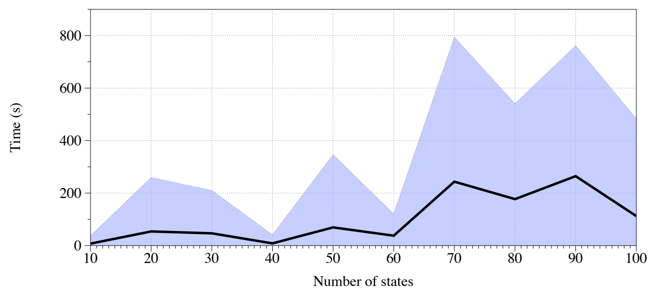
equation based procedure will be stressed with more complex models



(a) Equation based algorithms of WM.



(b) Maple built-in solver.



(c) PARAM.

FIGURE 4.11: Execution time vs number of states: cumulative reward, 5 parametric transitions, 2 parametric rewards.

in order to show the benefits of the WM implementation.

4.6.3 Empirical Complexity of the WM

In this section a set of complex input models are analyzed to compare the performance of the Maple 15 built-in solver and the WM implementation.

In Figures 4.12 and 4.13 a set of input models composed by 100 states and 5 outgoing transitions per state are analyzed with the two equation based solvers. The property under analysis is a flat reachability formula. For each point of the graph about 100 samples have been taken.

The execution time of the WM implementation significantly overcomes the performance of the built-in solver of Maple. Indeed, the former never took more than 5 minutes, while the latter ran for more than 4 hours in the worst case. Hence, the fill-in reduction strategies adopted for the WM implementation proved to be effective in speeding up the design time partial evaluation.

Another observation can be done on the two figures. Though the average execution time is still monotonically growing, the maximum execution time does not have a regular trend. This issue appears for the larger models, where the impact of topology is not negligible.

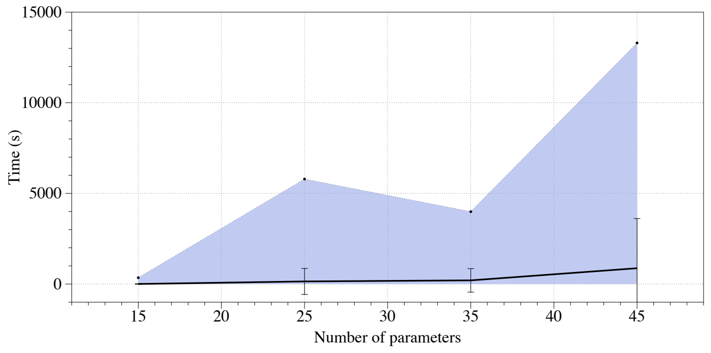


Figure 4.12: Stress test of the Maple built-in solver: 100 states, up to 45 transition parameters.

To further stress the WM implementation and to try an empirical assessment of the actual performance of the tool (which is notably less than the one expected from the worst case analysis of Section

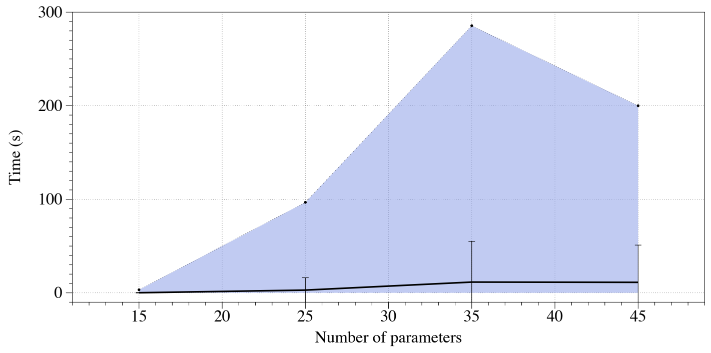


Figure 4.13: Stress test of the WM implementation: 100 states, up to 45 transition parameters.

4.2.1.2), in Figure 4.14 a set of input models composed by 200 states and 5 outgoing transitions per state have been analyzed. The property under analysis is still a flat reachability formula. For each point of the graph 25 samples have been collected.

An empirical fitting of the two curves on the graph is reported. The empirical complexity nicely fits a 3rd order polynomial function of the number of parameters, with high values of the determination index R^2 confirming the goodness of the fit. This is however an empirical measure based on 150 random cases and can only be considered as an indicative estimation of the actual temporal complexity of the WM implementation. It is anyway notable that the execution time even with so large models never took more than half an hour, proving the feasibility of the WM approach. Models with more parameters have been analyzed too, but very often they exhausted the available memory (8Gb) and the analyses have been interrupted

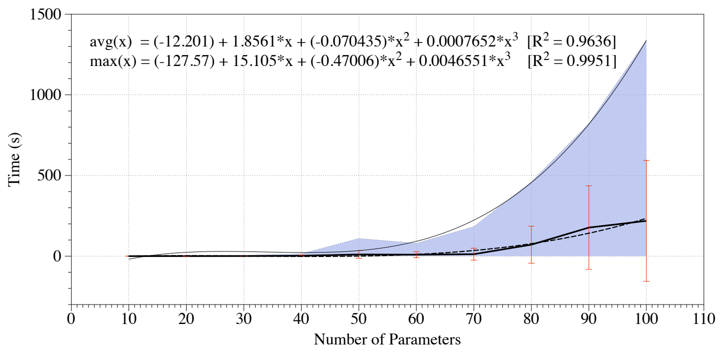


Figure 4.14: Stress test of the WM implementation with 200 states and up to 100 parameters.

As a final observation, the red error bars on the graph of Figure 4.14 represent the standard deviation of the sample set for the corresponding number of parameters. As already noted for Figure 4.13, the variability of the execution times grows for more complex models. This issue seems to be related to topological factors and further investigation are being conducted in order to formally characterize them.

Syntax-Diven Analysis 5

The program and the correctness proof grow hand in hand.

Edsger Dijkstra

The approaches proposed in the previous chapters dealt with model-based quantitative verification. In particular, the behavior of the running software has been formalized through a Markov process that captures its temporal evolutions and provides room to model uncertainty and variability. Such models pave the way to the application of efficient model-checking techniques, as well as the applicability of the WorkingMom analysis.

A closer look at the verification techniques discussed in Part I reveals a latent modus operandi concerning the run time verification of adaptive software: the actual purpose of keeping models "alive" at runtime is to provide a semantic lens through which the data fed by monitors can be interpreted. Generalizing this observation, quantitative analysis can be considered as a special semantic evaluation of the software that is continuously updated whenever new information is gathered from the running instances. This insight has a natural parallelism with the classic semantic interpretation of programs [119].

This chapter investigates concepts typical of the semantic interpretation of formal languages and extends their application to the problem of efficient quantitative analysis at run time. The enabling driver of efficiency will be in this case incrementality (cf. Chapter 4.5).

The result of this investigation is SiDECAR (Syntax-DrivEn inCrementAl veRification), a general framework to define verification procedures, which are automatically enhanced with incrementality by the framework itself. The framework follows a syntactic-semantic approach, since it assumes that the artifacts to be verified have a syntactic structure described by a formal grammar, and that the verification procedure is encoded as synthesis of semantic attributes [119], associated with the grammar and evaluated by traversing the syntax tree of the artifact.

Among the various classes of formal languages, SiDECAR is based on Floyd grammars [74], which allow for re-parsing, and hence semantic re-analysis, to be confined within an inner portion of the input that encloses the changed part. This property is the key for an efficient incremental verification procedure: since the verification procedure is encoded within attributes, their evaluation proceeds incrementally, hand-in-hand with parsing.

SiDECAR is a general framework for the definition of incremental syntactic-semantic verification procedures. To show it effectiveness for quantitative verification, it is used in this chapter for reliability and cost analysis, but the methodology can be applied to more different types of analysis, including functional verification.

The rest of the chapter is structured as follows. Section 5.1 introduces some background concepts on Floyd grammars and attribute grammars. Section 5.2 shows how SiDECAR exploits Floyd grammars to support syntactic-semantic incremental verification. In section 5.3, an incremental quantitative analysis procedure is defined for structured programs within SiDECAR. Finally, Section 5.4 discusses the ins and outs of the framework and compares with related work.

5.1 Background

Before explaining how SiDECAR works, in this section the basic definitions of Floyd grammars and attribute grammars are reported. For more information on formal languages and grammars, the interested reader could refer to [92] and [174].

5.1.1 Floyd Grammars

A *context-free (CF)* grammar G as a tuple $G = \langle V_N, V_T, P, S \rangle$, where V_N is a finite set of non-terminal symbols; V_T is a finite set of terminal symbols, disjoint from V_N; $P \subseteq V_N \times (V_N \cup V_T)^*$ is a relation whose elements represent the rules of the grammar; $S \in V_N$ is the axiom or start symbol.

The following naming convention is adopted in the rest of this chapter, unless otherwise specified: non-terminal symbols are in capital letters, such as $\langle A \rangle$; terminal ones are enclosed within single quotes, such as '+' or are denoted by lowercase letters at the beginning of the alphabet $(a, b, c, \ldots)$; lowercase letters at the end of the alphabet $(u, v, x, \ldots)$ denote terminal strings; ε denotes the empty string. For the notions of *immediate derivation* $(\Rightarrow)$, *derivation* $(\overset{*}{\Rightarrow})$, and the language $L(G)$ generated by a grammar G please refer to the standard literature, e.g., [92].

A rule is in *operator form* if its right hand side (rhs) has no adjacent non-terminals; an *operator grammar (OG)* contains only rules in operator form.

$$\begin{aligned}
\langle S \rangle &::= \langle A \rangle \mid \langle B \rangle \\
\langle A \rangle &::= \langle A \rangle \text{ '+' } \langle B \rangle \mid \langle B \rangle \text{ '+' } \langle B \rangle \\
\langle B \rangle &::= \langle B \rangle \text{ '*' 'n'} \mid \text{'n'}
\end{aligned}$$

(a) Arithmetic expressions grammar

	'n'	'*'	'+'
'n'		$\gtrdot$	$\gtrdot$
'*'	$\doteq$		
'+'	$\lessdot$	$\lessdot$	$\gtrdot$

(b) Precedence matrix

FIGURE 5.1: Example of an operator grammar ('n' stands for any natural number) and its operator precedence matrix

Floyd grammars (FGs) [74], also called operator precedence grammars, are defined starting from operator grammars by means of binary relations on V_T named *precedence*. Given two terminals, the precedence relations between them can be of three types: *equal-precedence* ($\doteq$), *takes-precedence* ($\gtrdot$), and *yields-precedence* ($\lessdot$). The meaning of precedence relations is analogous to the one between arithmetic operators and is the basic driver of deterministic parsing for these grammars. Precedence relations can be computed in an automatic way for any operator grammar.

The precedence relations can be represented by a $V_T \times V_T$ matrix, named *operator precedence matrix (OPM)*. An entry $m_{a,b}$ of an OPM represents the set of operator precedence relations holding between terminals a and b. For example, Fig. 5.1(b) shows the OPM for the grammar of arithmetic expressions in Fig. 5.1(a). Precedence relations have to be neither reflexive, nor symmetric, nor transitive, nor total. If an entry $m_{a,b}$ of an OPM M is empty, the occurrence of the terminal a followed by the terminal b represents a malformed input, which cannot be generated by the grammar. Notice however that this is not the only class of possible malformed input.

Definition 5.1.1 (Floyd Grammars) An OG G is a Floyd grammar if and only if its OPM is a conflict-free matrix, i.e., for each $a, b \in V_T, |m_{a,b}| \leq 1$.

Definition 5.1.2 (Fisher Normal Form) A FG is in Fischer Normal Form (FNF) if, for any two rules $\langle A \rangle \Rightarrow \alpha$, $\langle B \rangle \Rightarrow \beta$, $\alpha = \beta$ implies $\langle A \rangle = \langle B \rangle$; $\alpha \in V_N$ (renaming rule) iff $\langle A \rangle = \langle S \rangle$; $\alpha = \varepsilon$ implies $\langle A \rangle = \langle S \rangle$; $\langle S \rangle$ occurs in no rhs.

The grammar of Fig. 5.1(a) is in FNF. In the rest of this chapter, it is assumed, without loss of generality, that all the FGs are in FNF. Also, the input strings are assumed to be implicitly enclosed between two '#' special characters, corresponding to the begin and end of the input, such that '#' yields precedence to any other character and any character takes precedence over '#'.

The key feature of FG parsing is that a sequence of terminal characters enclosed within a pair $\lessdot\ \gtrdot$ and separated by $\doteq$ uniquely determines a rhs to be replaced, by a shift-reduce algorithm, by the corresponding left-hand side (lhs). Notice that in the parsing of these grammars non-terminals are "transparent", i.e., they are not considered for the computation of the precedence relations.

Example 5.1.1 (Parsing FGs.) Consider the syntax tree of Fig. 5.2 generated by the grammar of Fig. 5.1(a): the leaf '6' is preceded by

'+' and followed by '*'. Because '+' $\lessdot$ '6' $\gtrdot$ '*', '6' is reduced to $\langle B \rangle$. Similarly, in a further step '+' $\lessdot$ $\langle B \rangle$ '*' $\doteq$ '7' $\gtrdot$ '*' and the reduction $\langle B \rangle \Rightarrow \langle B \rangle$ '*' '7' is applied (notice that non-terminal $\langle B \rangle$ is "transparent"), and so on.

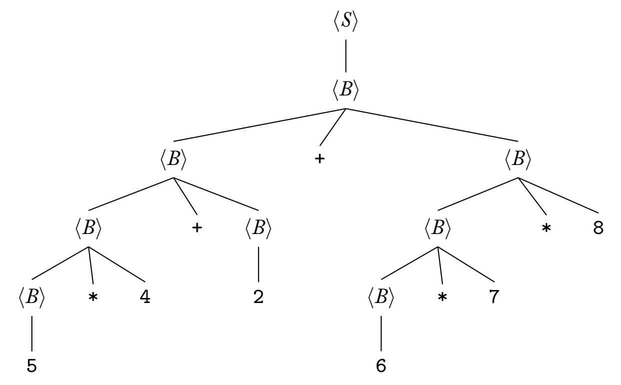


FIGURE 5.2: Abstract syntax tree of the expression '5*4+2+6*7*8'

5.1.2 Attribute Grammars

Attribute Grammars (AGs) have been proposed by Knuth as a way to express the semantics of programming languages [119]. AGs extend CF grammars by associating *attributes* and semantic functions to the rules of a CF grammar; attributes define the "meaning" of the corresponding nodes in the syntax tree.

In this chapter only *synthesized* attributes are considered, which characterize an information flow from the children nodes (of a syntax tree) to their parents. Notice however that more general attribute schemas do not add semantic power but have been introduced just to simplify the specification [119] .

An AG is obtained from a CF grammar G by adding a finite set of attributes SYN and a set SF of semantic functions. Each symbol $X \in V_N$ has a (possibly empty) set of (synthesized) attributes $SYN(X)$; $SYN = \bigcup_{X \in V_N} SYN(X)$.

In the following, the symbol α denotes a generic element of SYN; each α takes values in a corresponding domain T_α. The set SF consists of functions, each of them associated with a rule p in P. For each attribute α of the lhs of p, a function $f_{p\alpha} \in SF$ synthesizes the value of α based on the attributes of the non-terminals in the rhs of p.

Example 5.1.2 (Attribute schema for arithmetic expressions.) For example, the grammar in Fig. 5.1(a) can be extended to an attribute grammar that computes the value of an expression. All nodes have only one attribute called *value*, with $T_{value} = \mathbb{N}$. The set of semantic functions SF is defined as in Fig. 5.3, where semantic functions are enclosed in braces next to each rule:

$$\begin{array}{lll}
\langle S\rangle & ::= \langle A\rangle & \{value(\langle S\rangle) = value(\langle A\rangle)\} \\
\langle S\rangle & ::= \langle B\rangle & \{value(\langle S\rangle) = value(\langle B\rangle)\} \\
\langle A_0\rangle & ::= \langle A_1\rangle \text{ `+' } \langle B\rangle & \{value(\langle A_0\rangle) = value(\langle A_1\rangle) + value(\langle B\rangle)\} \\
\langle A\rangle & ::= \langle B_1\rangle \text{ `+' } \langle B_2\rangle & \{value(\langle A\rangle) = value(\langle B_1\rangle) + value(\langle B_2\rangle)\} \\
\langle B_0\rangle & ::= \langle B_1\rangle \text{ `*' `n'} & \{value(\langle B_0\rangle) = value(\langle B_1\rangle) * eval(\text{`n'})\} \\
\langle B\rangle & ::= \text{`n'} & \{value(\langle B\rangle) = eval(\text{`n'})\}
\end{array}$$

Figure 5.3: Example of attribute grammar

The $+$ and $*$ operators appearing within braces correspond, respectively, to the standard operations of arithmetic addition and multiplication, and $eval(\cdot)$ evaluates its input as a number. Notice also that, within a rule, different occurrences of the same grammar symbol are denoted by distinct subscripts.

5.2 Syntactic-Semantic Incrementality

SiDECAR exploits a syntactic-semantic approach to define verification procedures that are encoded as semantic functions associated with an attribute grammar. In this section, FGs, equipped with a suitable attribute schema, are exploited to define verification procedures that automatically provide incrementality in a natural and efficient way.

5.2.1 The Locality Property and Syntactic Incrementality

The main reason for the choice of FGs is that, unlike more commonly used grammars that support deterministic parsing, they enjoy the *locality property*, i.e., the possibility of starting the parsing from any arbitrary point of the sentence to be analyzed, independent of the context within which the sentence is located.

In fact for FGs the following proposition holds.

Proposition 5.2.1 (Locality property.) *If* $a\langle A\rangle b \overset{*}{\Rightarrow} asb$*, then, for every* t, u*,* $\langle S\rangle \overset{*}{\Rightarrow} tasbu$ *iff* $\langle S\rangle \overset{*}{\Rightarrow} ta\langle A\rangle bu \overset{*}{\Rightarrow} tasbu$*. As a consequence, if* s *is replaced by* v *in the context* $\langle ta, bu\rangle$*, and* $a\langle A\rangle b \overset{*}{\Rightarrow} avb$*, then* $\langle S\rangle \overset{*}{\Rightarrow} ta\langle A\rangle bu \overset{*}{\Rightarrow} tavbu$*, and (re)parsing of* $tavbu$ *can be stopped at* $a\langle A\rangle b \overset{*}{\Rightarrow} avb$*.*

Hence, if the derivation can be obtained —by means of a bottom-up parser— $a\langle A\rangle b \overset{*}{\Rightarrow} avb$, a *matching condition* with the previous derivation $a\langle A\rangle b \overset{*}{\Rightarrow} asb$ is satisfied and the old subtree rooted in $\langle A\rangle$ can be safely replaced with the new one, independently on the global context $\langle ta, bu\rangle$ (only the local context $\langle a, b\rangle$ matters for the incremental parsing).

Example 5.2.1 (Incremental parsing of FGs.) Consider the string and syntax tree of Fig. 5.2. Assume that the expression is modified by replacing the term '6*7*8' with '7*8'. The corresponding new subtree can clearly be built independently within the context ⟨'+', '#'⟩. The matching condition is satisfied by '+'$\langle B\rangle$'#' $\overset{*}{\Rightarrow}$ '+''6''*''7''*''8''#' and '+'$\langle B\rangle$'#' $\overset{*}{\Rightarrow}$ '+''7''*''8''#'; thus the new subtree can replace the original one without affecting the remaining part of the global tree.

If, instead, the second '+' is replaced by a '*', the affected portion of syntax tree would be larger and more re-parsing would be necessary[1].

In general, the incremental parsing algorithm, for any replacement of a string w by a string w' in the context $\langle t,u \rangle$, automatically builds the minimal "sub-context" $\langle t_1, u_1 \rangle$ such that for some $\langle A \rangle$, $a\langle A \rangle b \overset{*}{\Rightarrow} at_1wu_1b$ and $a\langle A \rangle b \overset{*}{\Rightarrow} at_1w'u_1b$.

The locality property has also been shown to support an efficient parallel parsing technique [19], which is not yet exploited here.

On the other hand, the locality property has a price in terms of generative power. For example, the LR grammars traditionally used to describe and parse programming languages in general do not satisfy it. Indeed, FGs cannot generate all the deterministic languages, as LRs do. This limitation, however, is more of theoretical interest than of real practical impact. Most real-life programming languages in fact can be generated by a suitable FG [92].

At the current advancement state of SiDECAR, it includes an incremental parser for FGs that exhibits the following features:

- linear complexity in the length of the string, in case of parsing from scratch
- linear complexity in the size of the *modified subtree(s)*, in case of incremental parsing
- $O(1)$ complexity of the matching condition test, thanks to an hash based data structure that stores the previous matches.

5.2.2 Semantic Incrementality

In a bottom-up parser, semantic actions are performed during a reduction. This allows the re-computation of semantic attributes after a change to proceed hand-in-hand with the re-parsing of the modified substring.

Suppose that, after replacing a substring w with w', incremental re-parsing builds a derivation $\langle N \rangle \overset{*}{\Rightarrow} xw'z$, with the same nonterminal $\langle N \rangle$ as in $\langle N \rangle \overset{*}{\Rightarrow} xwz$, so that the matching condition is verified. Assume also that $\langle N \rangle$ has an attribute α_N.

Two situations may occur related to the computation of α_N:

[1]Some further optimization could be applied by integrating the matching condition with techniques adopted in [83].

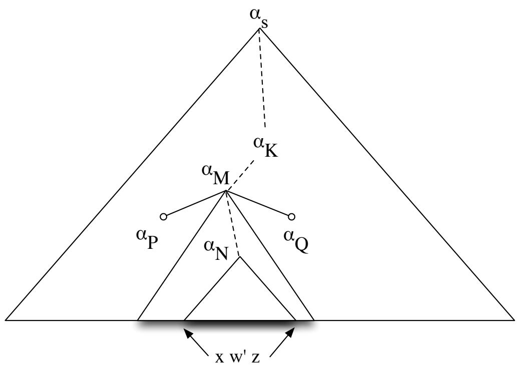


FIGURE 5.4: Incremental evaluation of semantic attributes

1. The α_N attribute associated with the new subtree rooted in $\langle N \rangle$ has the same value as before the change. In this case, all the remaining attributes in the rest of the tree will not be affected, and no further analysis is needed.

2. The new value of α_N is different from the one it had before the change. In this case (see Fig. 5.4) only the attributes on the path from $\langle N \rangle$ to the root $\langle S \rangle$ (e.g., $\alpha_M, \alpha_K, \alpha_S$) can change and thus need to be re-computed. The values of the other attributes not on the path from $\langle N \rangle$ to the root (e.g., α_P and α_Q) do not change: there is no need to re-compute them.

5.3 Incremental Quantitative Analysis

In this section the SiDECAR is used to define an incremental procedure for the quantitative analysis of a structured program. The types of properties under analysis are reliability and cost.

The programs appearing in this section are written in the *Mini* language, whose grammar is shown in Fig. 5.5. It is a minimalistic language that includes the major constructs of structured programming.

For the sake of readability and to reduce the complexity of attribute schemas, the constructs concerning variables has been omitted in the *Mini* language, focusing only on function calls and control constructs. Adding the management of variables is a simple exercise.

```
⟨⟨S⟩⟩          ::= 'begin'⟨stmtlist⟩ 'end'

⟨stmtlist⟩     ::= ⟨stmt⟩ ';'⟨stmtlist⟩
                 | ⟨stmt⟩ ';'

⟨stmt⟩         ::= ⟨function-id⟩ '(' ')'
                 | 'if'⟨cond⟩ 'then'⟨stmtlist⟩ 'else'⟨stmtlist⟩ 'endif'
                 | 'while'⟨cond⟩ 'do'⟨stmtlist⟩ 'endwhile'

⟨function-id⟩ ::= ...

⟨cond⟩         ::= ...
```

Figure 5.5: The grammar of the *Mini* language

Mini language resembles the simplified workflow language proposed in [60] for the analysis of reliability and failure propagation in workflow models[2] Indeed, the verification problem presented here for *Mini* can be viewed as a high-level abstraction of a similar verification problem for service compositions in the context of service-oriented architectures, where the call to possibly faulty functions mimics the call to third-party services.

[2]The attribute schema of [60] can be used as-is in SiDECAR, enhancing that analysis procedure with incrementality in a seamless way.

To show the benefits of incrementality, intrinsically supported by SiDECAR, for the verification procedure defined in the next subsections, two versions of the example program (shown in Fig. 5.6) will be analyzed. A third analysis will involve the change in the monitored value of the reliability of one of functions and will be described later, after introducing the attribute schema.

The invocation of function *opC()* at line 3 of version 1 is replaced by the execution of a while loop in version 2 (lines 3 to 5). Notice that this is a structural change in the control flow of the program. Figure 5.7 depicts the syntax tree of version 1 of the program, as well as the subtree that is different in version 2; nodes of the tree have been numbered for quick reference.

```
1 begin
2  opA();
3  opC();
4  if (e1)
5   then opB();
6   else opA();
7  endif;
8 end
```

(a) Version 1

```
1 begin
2  opA();
3  while (e2) do
4   opD();
5  endwhile;
6  if (e1)
7   then opB();
8   else opA();
9  endif;
10 end
```

(b) Version 2

FIGURE 5.6: The two versions of the example program

Before proceeding to the details of the analysis procedure, and the corresponding attribute schema, a few notational conventions are introduced. Given a *Mini* program P, F_P is the set of functions and E_P is the set of conditions of *if* and *while* statements in P. The subscript P in F_P and E_P is omitted whenever the program is clear from the context.

5.3.1 Applying SiDECAR

To provide a simplified quantitative analysis procedure based on SiDECAR the first step is to describe the semantics of the involved qualities, in this case reliability and cost.

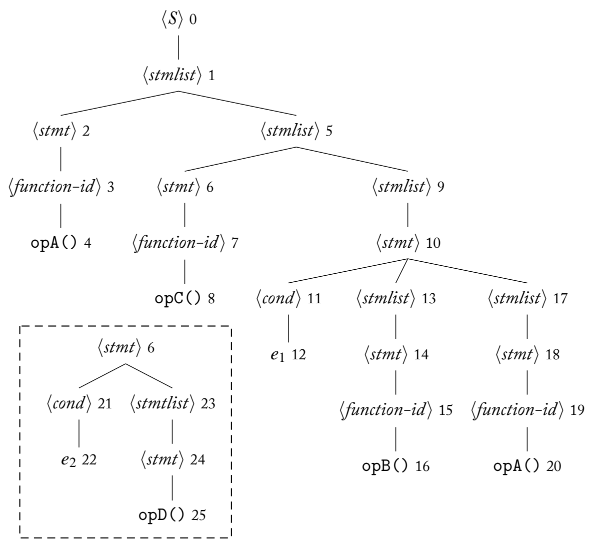


Figure 5.7: The syntax tree of version 1 of the example program; the subtree in the box shows the difference (node 9) in the syntax tree of version 2

Let assume that each function $f \in F$ has a probability $Pr_S(f) \in [0,1]$ of successfully completing its execution and a cost $C(f) \in \mathbb{R}$. To simplify the later formalization, let assume that the cost is paid at the moment of invocation, regardless the correct execution of the function. Assume also that for each expression $e \in E$ there is a probability $Pr_T(e)$ of the condition to be satisfied.

The values $Pr_S(f)$, $Pr_T(e)$, and $C(f)$ are analogous to the concepts of reliability, probability of moving toward a certain state, and state reward stated in Chapter 4. They depend on the usage profile and environmental conditions, and can be guessed at design time on the basis of the designer's experience, the knowledge of the application domain, or from previous executions; at run time, they can instead be gathered from the running instances by combining monitoring and statistical inference techniques [73].

The reliability of a program is computed as the expected probability value of its successful completion. Analogously, concerning cost analysis, the expected cumulative cost of a run is computed. Every failure is assumed to be unrecoverable and independent from the others.

The reliability of a sequence of statements is the probability that all of them are executed successfully. Given the independence of the failure events, the probability of their conjunction can be computed as the product of the reliability value of each statement. The cost of a sequence of statements is the sum of the costs of each of them.

For an *if* statement with condition *e*, its reliability is the reliability of the *then* branch weighted by the probability of *e* to be *true*, plus the reliability of the *else* branch weighted by the probability of *e* to be *false*. The cost of an *if* statement can be computed analogously. These intuitive definitions are formally grounded on the law of total probability and the assumption of independence.

For a *while* statement with condition *e* and body *b*, the expected reliability and the expected cost can be computed by reasoning in a first-step way (cf. Section 4.2.1.3) as in the following equations:

$$\begin{aligned} Pr_S(\langle while \rangle) &= Pr_F(\langle e \rangle) \cdot 1 + Pr_T(\langle e \rangle) \cdot Pr_S(body) \cdot Pr_S(\langle while \rangle) \\ C(\langle while \rangle) &= Pr_T(\langle e \rangle) \cdot (C(body) + Pr_S(body) \cdot C(\langle while \rangle)) \end{aligned} \tag{5.1}$$

whose solutions are:

$$\begin{aligned} Pr_S(\langle while \rangle) &= \frac{1 - Pr_T(\langle e \rangle)}{1 - Pr_T(\langle e \rangle) \cdot Pr_S(body)} \\ C(\langle while \rangle) &= \frac{Pr_T(\langle e \rangle) \cdot C(body)}{1 - Pr_T(\langle e \rangle) \cdot Pr_S(body)} \end{aligned} \tag{5.2}$$

It is now possible to encode this analysis through the following attributes:
- $SYN(\langle S \rangle) = SYN(\langle stmlist \rangle) = SYN(\langle stmt \rangle) = \{\gamma, \vartheta\}$;
- $SYN(\langle cond \rangle) = \{\delta\}$;
- $SYN(\langle function\text{-}id \rangle) = \{\eta\}$;

where:

- γ represents the reliability of the execution of the subtree rooted in the node the attribute corresponds to.
- ϑ represents the cumulative cost of the subtree rooted in the node the attribute corresponds to.

- δ represents $Pr_T(e)$, with e being the expression associated with the corresponding node.
- η is a string corresponding to the literal value of an identifier.

The attribute schema is defined as follows:

1. $\langle\langle S\rangle\rangle$ ::= 'begin' $\langle stmtlist\rangle$ 'end'
   $\gamma(\langle S\rangle) := \gamma(\langle stmtlist\rangle)$
   $\vartheta(\langle S\rangle) := \vartheta(\langle stmtlist\rangle)$

2. a) $\langle stmtlist_0\rangle$ ::= $\langle stmt\rangle$ ';' $\langle stmtlist_1\rangle$
   $\gamma(\langle stmtlist_0\rangle) := \gamma(\langle stmt\rangle)\cdot\gamma(\langle stmtlist_1\rangle)$
   $\vartheta(\langle stmtlist_0\rangle) := \vartheta(\langle stmt\rangle)+\vartheta(\langle stmtlist_1\rangle)$

   b) $\langle\langle stmtlist\rangle\rangle$ ::= $\langle stmt\rangle$ ';'
   $\gamma(\langle stmtlist\rangle) := \gamma(\langle stmt\rangle)$
   $\vartheta(\langle stmtlist\rangle) := \vartheta(\langle stmt\rangle)$

3. a) $\langle\langle stmt\rangle\rangle$ ::= $\langle function\text{-}id\rangle$ '('')'
   $\gamma(\langle stmt\rangle) := Pr_S(f)$
   $\vartheta(\langle stmt\rangle) := C(f)$
   with $f\in F$ and $\eta(\langle function\text{-}id\rangle)=f$

   b) $\langle stmt\rangle$ ::= 'if' $\langle cond\rangle$ 'then' $\langle stmlist_0\rangle$ 'else' $\langle stmlist_1\rangle$ 'endif'
   $\gamma(\langle stmt\rangle) := \delta(\langle cond\rangle)\cdot\gamma(\langle stmtlist_0\rangle) + (1-\delta(\langle cond\rangle))\cdot\gamma(\langle stmtlist_1\rangle)$
   $\vartheta(\langle stmt\rangle) := \delta(\langle cond\rangle)\cdot\vartheta(\langle stmtlist_0\rangle) + (1-\delta(\langle cond\rangle))\cdot\vartheta(\langle stmtlist_1\rangle)$

   c) $\langle\langle stmt\rangle\rangle$ ::= 'while' $\langle cond\rangle$ 'do' $\langle stmtlist\rangle$ 'endwhile'
   $$\gamma(\langle stmt\rangle) := \frac{1-\delta(\langle cond\rangle)}{1-\delta(\langle cond\rangle)\cdot\gamma(\langle stmtlist\rangle)}$$
   $$\vartheta(\langle stmt\rangle) := \frac{\delta(\langle cond\rangle)\cdot\vartheta(\langle stmtlist\rangle)}{1-\delta(\langle cond\rangle)\cdot\gamma(\langle stmtlist\rangle)}$$

4. $\langle cond\rangle$ ::= …
   $\delta(\langle cond\rangle) := Pr_T(e)$
   with $\eta(\langle cond\rangle)=e$

It is now possible to use SiDECAR to analyze the two versions of the example program of Figure 5.6. In the steps of attribute synthesis numbers are used to refer to corresponding nodes in the syntax tree of Figure 5.7.

For the following analyses, let assume the reliability of the four functions are: $Pr_S(opA) = .97$, $Pr_S(opB) = .99$, $Pr_S(opC) = .95$, and $Pr_S(opD) = .975$; the costs are $C(opA) = 2$, $C(opB) = 5$, $C(opC) = 7$, and $C(opD) = 3$; the probability of satisfying the two conditions are $Pr_T(e_1) = .3$, and $Pr_T(e_2) = .6$.

Example Program - Version 1

Given the abstract syntax tree in Fig. 5.7, evaluation of attributes leads to the following values (η attributes omitted):

$$
\begin{aligned}
\gamma(2) &:= .97\\
\vartheta(2) &:= 2\\
\gamma(6) &:= .95\\
\vartheta(6) &:= 7\\
\delta(11) &:= .3\\
\gamma(14) &:= .99\\
\vartheta(14) &:= 5\\
\gamma(13) &:= .99\\
\vartheta(13) &:= 5\\
\gamma(18) &:= .97\\
\vartheta(18) &:= 2\\
\gamma(17) &:= .97\\
\vartheta(17) &:= 2\\
\gamma(10) &:= .3\cdot .99 + .7\cdot .97 = .976\\
\vartheta(10) &:= .3\cdot 5 + .7\cdot 2 = 2.9\\
\gamma(9) &:= .976\\
\vartheta(9) &:= 2.9\\
\gamma(5) &:= .95\cdot .976 = .9272\\
\vartheta(5) &:= 7 + 2.9 = 9.9\\
\gamma(1) &:= .97\cdot .9272 = .899384\\
\vartheta(1) &:= 2 + 9.9 = 11.9\\
\gamma(0) &:= .899384\\
\vartheta(0) &:= 11.9
\end{aligned}
$$

The resulting values for $\gamma(0)$ and $\vartheta(0)$ represent the expected reliability and the expected cost of a run of the program.

Example Program - Version 2

Version 2 of the example program differs from version 1 because the invocation to function *opC()* at line 3 is replaced by the execution of a while loop (lines 3 to 5 of version 2).

The incremental parsing is able to identify the minimum context thanks to the precedence relation. The subtree corresponding to the modified part of the input is shown in the dashed box of Figure 5.7.

Since the matching condition is satisfied, this subtree is hooked into node 6 of the original tree. Re-computation of the attributes proceeds upward to the root, leading to the following final values (only those who changed from the previous evaluation are reported):

$$
\begin{aligned}
\delta(21) &:= .6\\
\gamma(24) &:= .975\\
\vartheta(24) &:= 3\\
\gamma(23) &:= .975\\
\vartheta(23) &:= 3\\
\gamma(6) &:= \frac{1-.6}{1-.6\cdot .975} = .9638554217\\
\vartheta(6) &:= \frac{.6\cdot 3}{1-.6\cdot .975} = 4.3373493976\\
\gamma(5) &:= .9638554217\cdot .976 = .9407228916\\
\vartheta(5) &:= 4.3373493976 + 2.9 = 7.2373493976\\
\gamma(1) &:= .97\cdot .9407228916 = .9125012049\\
\vartheta(1) &:= 2 + 7.2373493976 = 9.2373493976\\
\gamma(0) &:= .9125012049\\
\vartheta(0) &:= 9.2373493976
\end{aligned}
$$

Example Program - Changing an Attribute

Let assume a new value for the reliability of *opD()* is provided by monitors. The same incremental analysis of the previous section would be applied in this case too. The number of operations required to propagate the change from a leaf of the three to its root is still $O(\log(n))$, where n is the number of nodes.

Concluding, these examples show how SiDECAR re-analyzes only a limited part of the program and re-computes only a small subset of the attributes, reusing as much as possible from the previous analyses.

5.4 Discussion and Related Work

SiDECAR introduces a general methodology for the definition of incremental verification procedures, which has been applied in the previous section to deal with quantitative analysis.

Indeed, SiDECAR has only two usage requirements:

- the artifact to be verified should have a syntactic structure derivable from a FG;
- the verification procedure has to be formalized as synthesis of semantic attributes.

The expressiveness of FGs, discussed in section 5.2, and the well-known versatility of AGs guarantee that there is no practical limitation in using SiDECAR. Moreover, incrementality is automatically provided by the framework without any further effort for the developer.

The parsing algorithm used within SiDECAR has a temporal complexity linear in the length of the changes to be analyzed. Hence any change in the program has a minimal impact on the adaptation of the abstract syntax tree, without any further constraint on the grammar. Semantic incrementality allows for low-impact (re)evaluation of the attributes, by proceeding along the path from the node corresponding to the change to the root, whose length is normally logarithmic with respect to the length of the program. The use of SiDECAR may result in a significant reduction of the re-analysis and semantic re-evaluation steps. The saving can be very relevant in the case of large programs and rich and complex attributes schema.

The generality and flexibility of FGs allow for using SiDECAR in a natural way much richer languages than the *Mini* example shown in this Chapter; on the other hand, having attribute grammars the same computational power as Turing machines, they enable formalizing in this framework any algorithmic schema at any sophistication and complexity level.

The generality of the methodology advocated by SiDECAR widens the scope of application to a number of scenarios. For example, at design time, SiDECAR can effectively support designers in evaluating the impact of changes in their products, in activities such as what-if analysis and regression verification, possibly integrated within IDE tools. Existing techniques for automated verification based either on model-checking or on deductive approaches, as well as their optimizations, could be adapted to use SiDECAR, exploiting the benefits of incrementality.

At run time, the incrementality provided by SiDECAR could be the key factor for efficient online verification of continuously changing situations, which could then trigger and drive the adaptation of self-adaptive systems. Furthermore, SiDECAR could also bring at run time the same analyses so far limited to design time because of efficiency reasons.

5.4.1 Related Work

In this section only related work concerning frameworks to define incremental verification procedures is briefly considered. Incremental approaches to quantitative verification have been already discussed in Section 4.5.

Several methodologies have been proposed in the literature as the basis for incremental verification techniques[3]. Most of them are grounded in the assume-guarantee [113] paradigm. This paradigm views a system as a collection of cooperating modules, each of which has to guarantee certain properties. The verification methods based on this paradigm are said to be compositional, since they allow reasoning about each module separately and deducing properties about their integration. If the effect of a change can be localized inside the boundary of a module, the other modules are not affected, and their verification does not need to be redone.

This feature is for example exploited in [48], which proposes a framework for performing assume-guarantee reasoning in an incremental and fully automatic fashion. Assume-guarantee based verification has been exploited also for probabilistic reasoning (e.g., in [134]), even though the author is not aware of approaches using it in an incremental fashion.

5.4.2 SiDECAR and the WorkingMom

The main limitation of the WorkingMom approach is that the closed-form parametric expression is tightly related to the structure of the

[3]Notice that the use of the term *incremental model checking* in the specific context of bounded model checking [28] has a different meaning, since it refers to the possibility of changing the bound of the checking.

model. Indeed, it is not possible, in general, to modify the structure of the Markov model and avoid to re-run the design time partial evaluation.

SiDECAR may instead allow the application of the Working-Mom paradigm to the analysis of artifacts described by instances of a FG; in such case the main limitation WM has for the analysis of Markov models could be overcome.

Consider the example of Section 5.3. The attribute schema can be trivially adapted to deal not only with numeric attributes but also with parametric expressions. This way, the design time partial evaluation would 1) be far more efficient because the complexity of parsing is linear in the number of statements (the cost of a single operation has essentially the same complexity as the WM), and 2) be able to incrementally re-evaluate the parametric formula in case of structural changes.

The first benefit has been already proved in [60], and an implementation in ANTLR is available (not incremental). The second benefit is being investigated in order to bring the incremental re-computation of the parametric expressions at run time too, possibly after an initial parsing conducted once for all at design time.

Part IV

Control

Model-Based Software Adaptation

Despite the large number of works applying some notions from control theory, the control of software systems can still be considered in its very preliminary stage. Developing accurate models for software is in fact hard. The presence of intrinsic non linearities, the variety of usage profiles, maintenance changes, and the interconnection of heterogeneous components are some of the reasons why it is so hard to directly provide a comprehensive behavioral model suitable for control [58, 61, 199].

This difficulty usually lead to the design of controllers focused on particular operating regions or conditions, or ad hoc solutions which address specific computing problems using control theory, but do not generalize [145, 183, 185]. Just to mention an example, [101] defined a controller for .NET thread pools that is not straightforwardly adapted to other architectures, though the high level task is quite similar.

At the same time, the quest for continuous verification of quality properties often leads to the definition of formal models able to capture a number of aspects of the running software that significantly characterize its behavior and support the assessment of some of its properties. These models are sometimes simple enough to allow the systematic synthesis of controllers capable of driving the modeled dynamics towards the required quality goal.

By controller is here intended, in a broad sense, any system that, properly coupled to the software system, makes it fulfill its requirements whenever they are feasible. Requirements can be strict constraints on the behavior (e.g. reliability equal to a certain value) or formulated as the optimization of certain metrics on the observed software executions (e.g. minimization of outsourcing costs or maximization of throughput).

In particular, any controller system should be able to deal with:

1. *(partial) changes of the requirements.* For example, if for some reason the required target value for the overall reliability changes, the controller should be able to drive the system toward a new operative state satisfying the requirement.

2. *sudden changes or fluctuations around the nominal operative point.* Interdependence among software parts and components involves the use of third-party services, remote storage, computing resources out of the control of a company, and so on. All these parts are characterized by the values of certain QoS metrics, usually stated within convenient service level agreements. During normal execution those values may deviate from nominal values because of external factors hardly predictable a priori (e.g. load conditions or hardware failures).

3. *accuracy errors in measurement and monitoring.* Quantitative assessment of the running system usually relies on monitoring and/or other measurement procedures. Each of these might get stuck into temporary biases, be affected by noise, or might require a certain time to produce an appropriate accuracy. A controller should provide a reasonable behavior even in presence of transitory errors on measured values. This capability, besides reducing the sensitivity to measurement errors, allows for the use of less invasive monitoring instruments, sometimes required for high accuracy but expensive in terms of performance overhead.

This research is aimed at supporting the claim that control theory provides a number of instruments that software engineers can exploit to ensure the achievement of non-functional goals in presence of changes in the environment. In particular this part of the thesis underlines the role that intermediate behavioral models can play in filling the gap between the domains of Software Engineering and Control theory, and provides a concrete instantiation of this paradigm for the continuous assurance of several quantitative properties.

The rest of this part is organized as follows. Chapter 6 describes a general methodology to synthesize controllers for software whose behavior is described through a DTMC or a D-MRM. These controllers can pursue the continuous assurance of any quantitative property expressible in the flat fragments of PCTL and R-PCTL, respectively. Finally, in Chapter 7, the special case of reliability-driven dynamic binding will be further investigated, showing how a different model can improve both efficiency and scalability.

Software Control through Markov Models 6

The purpose of software engineering is to control complexity, not to create it.

Pamela Zave

This chapter investigates a general methodology for the automatic synthesis of controllers for software systems whose behavior can be formalized through a Markov process (cf. Section 2.1.2). Markov models can abstract the control flow the software similarly to a finite state automaton, adding further information about the probability of certain transitions to be taken and rewards to be gained after visiting a certain state. The goal of the controllers targeted by this research is to keep a quantitative property expressible through (R-)PCTL *as close as possible* to its satisfaction. The proposed methodology can in principle be applied for all the properties that can be partially evaluated by the WM, though outside from the flat fragment its use is in general limited and computationally expensive. This issue will be briefly discussed in Section 6.5, while the rest of this chapter will be focused on the flat fragments of the two logics.

To simplify the presentation, the approach will be presented for the specific case of reliability assurance. Its generalization to all the other requirements will be straightforward (cf Section 6.5).

In the following, Section 6.1 will formally state the problem and introduce a running example for reliability assurance. Section 6.2 will discuss how a software model expressed as a Markov process can be translated into a dynamic system of equations. Section 6.3 present the control strategy and discusses a set of relevant formal qualities provided by the controller, that will be empirically assessed in Section 6.4. Finally, Section 6.5 will discuss the extension of the approach to R-PCTL and the limitations of the proposed controllers, and Section 6.6 discuss some related work.

6.1 Control-Oriented Modeling

Reliability can be generically defined as the probability of successfully accomplishing the assigned task [41]. Assume that each run of the system can be either successful or failed. The goal of the controller is to make the system continuously provide a *target* value for its reliability (or, conversely, for its failure probability).

The overall abstract picture of the controlled system is provided in Figure 6.1, where blocks *System* and *Controller* represent the modeled system and the controller, respectively. The raw *output* of the system is a sequence of events that represent the success or failure of a run. The *Learning Block* estimates the *actual* failure probability of the system from the output sequences (usually by statistical inference as in [31, 73]). The goal of the controller is to provide *input* values to the system so that the resulting output gets as close as possible the required target (and reach it if feasible), despite the possible *disturbances* affecting the system.

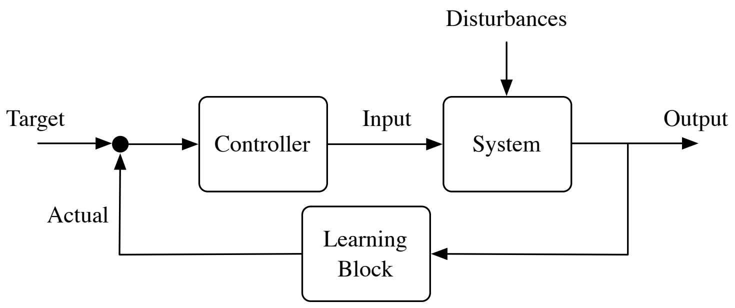


FIGURE 6.1: Block diagram of feedback control.

To understand what are inputs and disturbances in this picture, it necessary to look back at the underlying behavioral model. DTMCs are a suitable abstraction for reliability analysis (cf. Chapter 4.2). Indeed, the occurrence of a failure in a state of the process can be conveniently represented by a transition toward a "failure" state. Furthermore, it is here assumed for simplicity that a failure is a permanent condition, thus represented by an absorbing state. Analogously, the success of the execution can be formalized as the reaching of an absorbing "success" state after the last operation of the software workflow.

The temporal evolution of the process is univocally dictated by the transition matrix of the process. According to Chapter 2, some of the transitions are labeled with constant probabilities, representing immutable internal phenomena, while others are labeled by parameters, accounting for variabilities in the process dynamics. Changing the value of a parameter corresponds to a change in the distribution of the next event. At the software level this could for example increase or decrease the chances that a certain functionality is selected. In the extreme cases, by setting a probability to 0 (or 1) a certain functionality could be actually excluded (or included). A subset of the model parameters is the input of the system. By imposing their values, the controller tries to ensure continuous satisfaction of the target reliability. Disturbances are in turn represented by the remaining parameters that represent environmental phenomena, whose variations can only by measured but not directly influenced. Examples of disturbances are changes in the failure probability of outsourced operations or in the usage profile.

Formally, the set of the model parameters can be partitioned in the two subsets $\{c_0, c_1, \ldots, c_k\}$ and $\{r_0, r_1, \ldots, r_m\}$ representing the *control* and *disturbance* parameters, respectively. In the remaining of the chapter, the vectors $\mathbf{c}$ and $\mathbf{r}$ will contains the elements of the two sets in an arbitrary order.

The reliability of the system is formalized by the PCTL property $\mathscr{P}_{=?}(\diamond\, success)$[1] evaluated in the initial state. By means of the WM approach, this value can be evaluated to an analytical formula of the model parameters: $s = f(\mathbf{r}, \mathbf{c})$.

The goals of the controller are thus: 1) find an assignment $\bar{\mathbf{c}}$ of the control parameters such that s meets its target value $\bar{s}$, compensating the current disturbances, and 2) drive the system toward the new setting in a reasonable time and avoiding spikes and overshooting.

Further requirements will be imposed to the controller later, after the translation of the problem in terms of control theoretic concepts. But before proceeding, a small example is introduced in the next section to practically illustrate the approach.

[1]The notation $\mathscr{P}_{=?}$ is a short form for the value $Pr(\psi)$, corresponding to the probability of satisfying the path formula ψ.

6.1.1 A Representative Example

The simple case study of this chapter consists of a model for an image processing application. A high level software model is shown in Figure 6.2. The purpose of the system is to apply a filter to incoming images, followed by a beautifying post-processing phase. It is equipped with three different implementations of the filter: 1) direct filtering via internal software, 2) iterative filtering via internal software, and 3) direct filtering via outsourcing to an external service. The DTMC model for the image processing application is provided in Figure 6.3. The figure shows that all the operations have a certain probability of failure (represented by transitions entering state S_F, short notation for the failure state). State $S_1$1 represents the action of choosing among the different filtering options. The probabilities that govern this choice and the probability of applying one more iteration after the execution of the iterative filter (represented by state S_2) are the control variables. Control variables are indicated by parameters c_i in Figure 6.3 (referring to Figures 6.2 and 6.3, c_{1a} is the probability of choosing *the iterative filter*, c_{1b} is the probability of choosing *the internal direct filter* and thus $1 - c_{1a} - c_{1b}$ the probability of outsourcing; c_5 is the probability of requesting another iteration of the iterative filter). These values can be changed online by the controller while the software is executing. The controller in fact observes the overall behavior (i.e., the overall probability of success or failure) and the disturbances (i.e. the failure probability of each operation), and tries to guarantee the target global reliability by tuning the control parameters.

All the alternatives are implemented by black-box services that can be invoked and observed from outside only. For each of these services, a run-time monitor collects failure (or success) rates and estimates its reliability as the probability that an invocation to the service will not fail[2]. Notice that, the reliability of each operation is time-varying and the overall reliability depends on these values. Even if their nominal values are known at design time, unpredictable events could alter them. This scenario is fairly realistic, for example, in case of services shared with other customers. In such case, at different

[2]Estimates are here assumed to be statistically correct [163] and representative of the average or worst case, depending on the desired analysis scenario. Interested readers can refer to [73] for a deeper discussion about DTMC parameters estimation at runtime.

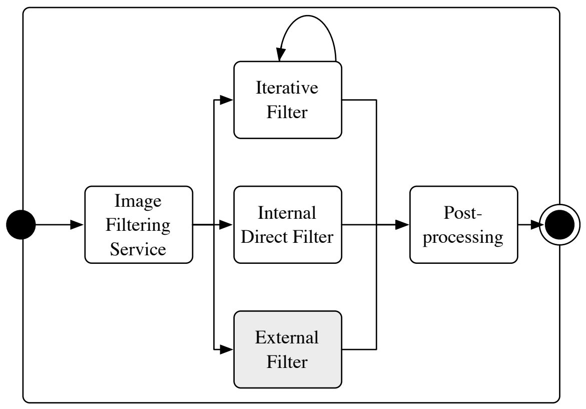


Figure 6.2: Schema of the software system.

times, their availability depends on load conditions of computational resources. Service reliabilities for each service are thus just observable values subject to variations during time (disturbances).

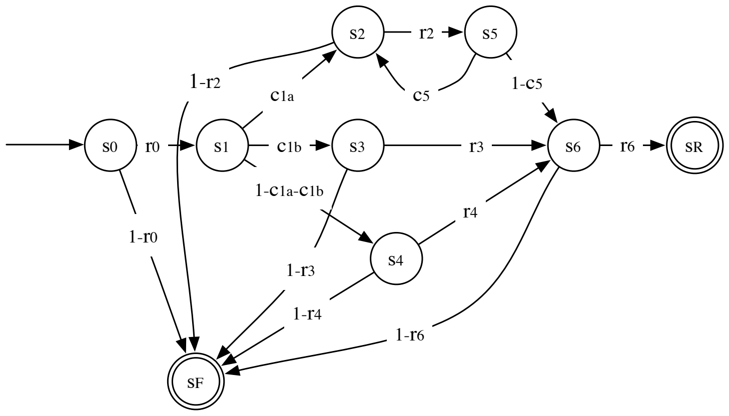


Figure 6.3: DTMC mode for the example system.

By applying the WM partial evaluation on the DTMC of Figure

6.3, it is possible to obtain a closed formula that makes the dependency of reliability (s) on control variables ($\mathbf{c}$) and measured reliabilities ($\mathbf{r}$) explicit:

$$s = r_0 \cdot r_6 \cdot \left(\frac{c_{1a} \cdot (-1 + c_5) \cdot r_2}{-1 + c_5 \cdot r_2} + c_{1b} r_3 + (1 - c_{1a} - c_{1b}) \cdot r_4 \right) \tag{6.1}$$

Equation (6.1) will be later used in Section 6.2 to build a dynamic control-theoretical model of the software behavior, and in Section 6.3 for the actual design of the controller.

6.2 Software Models as Dynamic Systems

In this section the dynamic evolution of the running software, as observed via the corresponding DTMC model, is cast in the control-theoretical framework of discrete-time dynamic systems [140], through which achieve self-adaptation[3].

A discrete-time dynamic system is described by the equations

$$\begin{cases} x(k+1) &= f\left(x(k), u(k), d_x(k)\right) \\ y(k) &= g\left(x(k), u(k), d_y(k)\right) \end{cases} \tag{6.2}$$

where $x \in \mathbb{R}^{n_x}$, $u \in \mathbb{R}^{n_u}$ and $y \in \mathbb{R}^{n_y}$ are called respectively the state, input and output vectors, $d_x \in \mathbb{R}^{n_x}$ and $d_y \in \mathbb{R}^{n_y}$ the state and output disturbance vectors (being those zeros if there are no variations with respect to the nominal conditions), $f(\cdot,\cdot,\cdot)$ and $g(\cdot,\cdot,\cdot)$ are real-valued vector functions of convenient dimensions, and k is an integer index counting the instants – not necessarily evenly spaced in time – when the system undergoes an evolution step. In a more general form, $f(\cdot,\cdot,\cdot)$ and $g(\cdot,\cdot,\cdot)$ could depend on an arbitrary number of real-valued parameters, possibly time-varying. The term "step k" denotes the time span between the k-th and the $(k+1)$-th instant.

Vector d_x represents disturbances corresponding to observable values in the environment that affect the system's state. Vector d_y accounts for measurement errors of the controlled variables.

The first equation in (6.2) is called the *state equation*, and dictates what the system state will be at the end of step k given what it was at the beginning, and given also the actions exerted on the system in that step, that are assumed to be correctly summarized by the values of u and d_x at its beginning. The state equation represents the dynamic system's character as *difference* equations, i.e., owing to the contextual presence of two subsequent index values. In other words, the state equation gives the system "memory of the past", and explains why the same action generally yields different effects depending on the system condition when it is applied. The input vector u represents *manipulated variables*, that can be used to influence the system's behavior, while the state disturbance d_x accounts for any action other than u, i.e., for any external entity that actually influences the system

[3]The reader interested in a general background on control theory may refer, for example, to [75].

state, and that in some cases can possibly be measured, but never manipulated.

The second equation in (6.2) is instead called the *output equation*. It is not dynamic, as shown by the presence of a single temporal index, and in most problems of interest it describes what one measures (vector y) to appreciate the system's behavior. The disturbance vector d_y represents possible alterations of said measurements, due e.g. to measurement noise, but *not* of the actual evolution of the system state.

A key concept of control theory is that modeling an object in the form (6.2) improves *per se* insights into that object by providing a formal model for its dynamic evolution. In fact, it naturally leads to distinguishing whether the same action yielded a different outcome than the previous time it was applied because the modeled object was in a different state, or something other than that affected it, or some of its parameters changed, or any combination thereof [62,140]. Such a distinction is of critical importance when *controlling* the system, because trying to "counteract the wrong cause" for an undesired effect can be disruptive. Also, models like (6.2) inherently give a quantitative and generally tractable meaning to the idea that "the system's reaction to a stimulus is related to its previous conditions", thanks to the explicit reference to the time.

On a similar front, observe that a model like (6.2) is concerned not only with the condition that the system can possibly reach under constant inputs as $k \to \infty$, but also with the way the system evolves in time. More generally, in the absence of disturbances and assuming complete knowledge of the system (i.e., of $f(\cdot,\cdot,\cdot)$ and $g(\cdot,\cdot,\cdot)$), the *initial state* $x(k_0)$ and the *input trajectory* $u(k)$, $k \geq k_0$, uniquely determine the *state trajectory* $x(k)$ and the *output trajectory* $y(k)$ for $k \geq k_0$. Disturbances and/or time-variances may alter that nominal behavior, and ultimately motivate the use of feedback, as explained in the next section.

Modeling Software Reliability. The general framework briefly discussed so far, has to be specialized to the case of control-oriented software modeling. Suppose that the adaptation mechanism, no matter how designed, acts at time points identified by an index k, as in (6.2). Also, let the average duration of a step be significantly longer than the time scale of the controlled system's dynamics. Translating from the control jargon to the case of interest, this means for example that if at the beginning of a step a controller changes the value of a control parameter, then at the end of the same step the effects of its action can be *measured*.

Assuming that the values of control parameters are perfectly known (because set by the controller), (6.2) reduces to

$$s(k+1) = \tilde{f}(r(k) + \Delta r(k), c(k)) \tag{6.3}$$

where $s(k+1)$ is the reliability in step k (the state x, as defined in the general model of Equation 6.2), $\mathbf{c}(k)$ are the control variables set for step k (i.e., decided at its beginning and kept constant through the step), $\mathbf{r}(k)$ the expected service reliabilities for step k (which in this example are estimated via monitoring, but in other case may also be nominal and possibly constant values), and $\Delta\mathbf{r}(k)$ accounts for any discrepancy between the real and expected service reliabilities in step k, produced for example by the convergence time of the monitors or measurement noise.

The form of function $\tilde{f}$ stems directly from Equation (6.1) and describes the relation between s and the model parameters (notice that $s : \mathbb{R}^n \to \mathbb{R}$, where n is the total number of parameters).

The output value y corresponds to s itself, that is, the observed measure of the system is exactly its reliability.

Although model (6.3) is nonlinear and time-varying, it has the very interesting property of being a "pure delay" system, i.e., the state vector does not appear on the right hand side of the state equation. In system-theoretical terms, what is here done is taking the steady-state model coming from DTMCs and using it in a dynamic framework under the assumption that any action at the beginning of a step has exhausted its effect at the end of that step.

6.3 Controlling the System's Dynamics by Feedback

In a nutshell, the idea of feedback can be summarized as plugging the controlled system into a larger one whose role is to set and manage the system's input dependently on its output, the control goal, and, possibly, on its state or an estimation of it when it is not directly measurable. Additionally, disturbance measurements can be considered, if available. Recalling the model of Equation (6.2), this means in general setting up a control law in the form:

$$\left\{\begin{array}{rcl} x_c(k+1) & = & f_c(x_c(k), w(k+1), \widehat{y}(k), \\ & & \widehat{x}(k), \widehat{d_x}(k), \widehat{d_y}(k)) \\ u(k+1) & = & g_c\left(x_c(k), w(k+1), \widehat{y}(k)\right) \end{array}\right. \tag{6.4}$$

where the hat symbol recalls that in the real world only measurements (or estimates) of some quantities are available (i.e., from now on, $\widehat{q}$ means a measured or estimated value of q, for any variable q). In (6.4) x_c is the controller's state vector (x in Equation 6.2 that refers to a general system), w the *set point* – i.e., the desired behavior for (part of) the controlled system's state and output, e.g. the desired reliability. Notice that the controller state and the desired behavior are assumed to be known exactly. The control law can be defined explicitly or, as it will be the case in this section, implicitly, stating the controller's state and output as the solution of an optimization problem. Notice also that the effect of the control applied at the k-th step is measurable and visible in the feedback value at time $k+1$.

By joining (6.2) and (6.4), a closed-loop system is obtained as feedback connection of the controlled system with its controller. The input of the closed-loop systems are the values of the set point w and of the disturbances d_x and d_y; its state is the union of x and x_c; and its output can be taken as (a function of) the set of variables for which a desired behavior is specified.

For a properly designed controller, formal guarantees can be provided on the behavior of the closed loop, also in presence of time variances and/or disturbances, as explained, for example, in [12,141,159]

Referring to the DTMC model as the software application model, there are transition probabilities that can be controlled (the control parameters), identified by $\mathbf{c}(k)$, and others that are not dependent on any action but are observable and measurable during software execution (disturbance parameters), identified by $\mathbf{r}(k)$, e.g. software failures. The corresponding dynamic model for the DTMC process has been introduced in Equation (6.3) and the value of w is set to $\bar{s}$, the target value for s.

Based on the current value $s(k)$, an estimation of the future value $\hat{s}(k+1)$ is available. Such estimation is obtained by substituting the estimated or measured disturbances in the function f and using the control variables computed for step k (notice that any measurement of s includes also the effect of d_y):

$$\hat{s}(k) = f(\hat{\mathbf{r}}(k), \mathbf{c}(k)). \tag{6.5}$$

Now let $J(k) = f_j(\mathbf{c})$ be a cost function on the control variables $\mathbf{c}(k)$. For example, consider the two control variables $c_{1a}(k), c_{1b}(k)$ corresponding to the probabilities of sending an incoming request to three different services (the third one is constrained to be $1 - c_{1a}(k) - c_{2b}(k)$); the cost of those values could be the cost of sending the request to each of the available services.

When no significant cost functions $J(k)$ can be naturally derived from the application domain[4], the designer can introduce a non informative cost function (such as a constant value) to indicate no preferences among all the feasible solutions.

The control law is implicitly specified as the solution of the following optimization problem:

$$\min J(\mathbf{c}) \tag{6.6}$$

subject to the constraints

$$\begin{gathered} ||w - \hat{s}(k)|| \leq \alpha ||w - s(k-1)|| \\ \forall c_i(k), 0 \leq c_i(k) \leq 1 \end{gathered} \tag{6.7}$$

where α is a real value in the range $(0,1)$ that affects the convergence rate of the solution: intuitively it imposes the error to be reduced in the next step by a factor α.

To preserve the semantic of the model, the set of constraints has to be extended with the implicit probabilistic constraints (i.e. the sum of outgoing transitions from each state has to be 1), as done for the control variables c_i.

[4]Cost estimation methodologies in the context of software reliability are an open research field. The interested reader could refer, for example, to [166].

6.3.1 Formal Assessment

As first consideration, notice that for each step k where a solution of the optimization problem (6.6) exists, the error $w - s(k)$ has an exponential decay. This is obtained by construction, based on how the controller was designed (first constraint in (6.7)). Moreover, under the same assumption, the time to converge to the desired solution is known. Indeed, let $e(0)$ be the initial error $w - s(0)$, then $e(k) = \alpha^k e(0)$, according to the exponential decay. If one assumes the system converged when the error $e(k) \leq \varepsilon$, then in nominal conditions this happens when:

$$k \geq \log_\alpha \frac{\varepsilon}{e(0)}. \tag{6.8}$$

If the system is no more in nominal conditions, i.e., if $\hat{\mathbf{r}}(k)$ deviates from its nominal value $r(k)$, the proposed solution is robust whenever a solution is found for the optimization problem. In fact, in such a case, the first constraint of (6.7) holds.

To prove this assertion, let first of all consider the ideal case $s(k) = \hat{s}(k)$, that is no noise is affecting the measurement of $s(k)$. Suppose now that there is an additive term, due to a difference in the estimation of $\mathbf{r}(k) = \hat{\mathbf{r}}(k) + \Delta\mathbf{r}(k)$, and therefore $s(k) = \hat{s}(k) + \Delta s(k)$. The error norm becomes $||w - \hat{s}(k) - \Delta s(k)||$ and the following equations hold

$$\begin{array}{c} ||w - \hat{s}(k) - \Delta s(k)|| \leq ||w - \hat{s}(k)|| + ||\Delta s(k)|| \\ ||w - \hat{s}(k)|| \leq \alpha ||w - s(k-1)|| \end{array} . \tag{6.9}$$

Notice that the second equation of (6.9) comes from the existence of a solution for the optimization problem. As a consequence, in the presence of a solution the stability still holds, while the convergence time equation does not hold anymore. However, if

$$||\Delta s(k)|| < (1 - \alpha)||w - s(k-1)|| \tag{6.10}$$

the error norm is guaranteed to diminish, although not at the (unfeasible) desired rate, determined in Equation 6.8.

Some words deserve to be spent on the role of α. The closer α is to 0, the faster is the system convergence. However, when the error is closer to zero there could be oscillations when changes occur, as testified by the inequality (6.10). On the other hand, when α approximates 1 the system convergence is slow, and though oscillations could potentially still occur before the error approaches zero, they are less intense.

Notice however, that in the previous analysis the triangle inequality has been used to quantify the upper bound of the norm of the difference as the sum of the norms (Equation (6.9)). This is definitely a coarse over-bounding, and makes the proposed assessment quite conservative. This approach is however shared by numerous robustness-related control-theoretical methodologies, and historically accepted in place of easier but less robust criteria. The interested reader can find a discussion in [153].

The situation in which the optimization problem has no feasible solutions can be easily identified [43][5] and used to trigger the intervention of a higher level controller (or even a human operator) because there is no control assignment that can further reduce the error. For the experiments presented in Section 6.1.1, the employed solver has the further feature of going *as close as possible* to the unfeasible constraint, therefore reaching the optimum value that is reachable in the system conditions.

Although a complete treatment of the matter is deferred to future work, an first way to automatically deal with unfeasibility can employ reliability estimates to recompute the set point as the feasible value nearest to the desired one. Indeed, this would intuitively correspond to the solution of the following optimization problem:

$$\begin{cases} w^*(k) = & min\{\bar{s}(k) - f(\mathbf{r}, \mathbf{c}(k))\} \\ \text{subject to:} & \mathbf{r} = \hat{\mathbf{r}} \end{cases} \tag{6.11}$$

The solution of (6.11) could identify the best feasible solution and actuate a graceful degradation of system's reliability, while waiting for higher level intervention.

[5]Mathematical solvers available off the shelf such as Maple, Mathematica, Matlab, or Cplex automatically detect the unfeasibility of the optimization problem.

6.4 Experimental Evaluation

The controller for the example system of Section 6.1.1 has been implemented accordingly to Equations (6.6) and (6.6), where the analytic expression for $s(k)$ has been derived from Equation (6.1) and the following cost function is minimized:

$$J(\mathbf{c}) = (J_{1a}c_{1a} + J_{1b}c_{1b} + J_5c_5)^2 \tag{6.12}$$

where J_{1a}, J_{1b} and J_5 are equal to one, therefore assuming all costs are equals. In the case study, the first constraint of Equation (6.7) is considered with an equal sign, supposing the requirement for the system to expose exactly the desired reliability.

For all the experiments, α has been set to .5 after manual inspection of the controller performance.

Changing Set Point and Service Reliabilities. In the first experiment, the reliability required over time is changed to show how the controller reacts to changes in the desired value. The simulation is divided into four different slots, each having 25 time units. During the first slot, the reliability requirement is set to 0.7, while in the following one it is 0.8. In the third slot, the desired reliability is 0.5 and it increases to 0.6 in the last slot. All these numbers are feasible, considering the reliabilities of the involved services.

Figure 6.4 depicts the overall system reliability (s) over time. A detail showing the exponential error decay has been magnified to simplify its observation. Figure 6.5 shows the corresponding control signals, c_{1a} is represented with a dashed line, while c_{1b} is the continuous curve; the dashed dotted line shows how c_5 changes over time.

Perturbations to the nominal model were added in the form of disturbances to the services reliabilities r_i, in order to show the controller convergence previously formally proved. The expressions of the services reliabilities are shown in Equation (6.13):

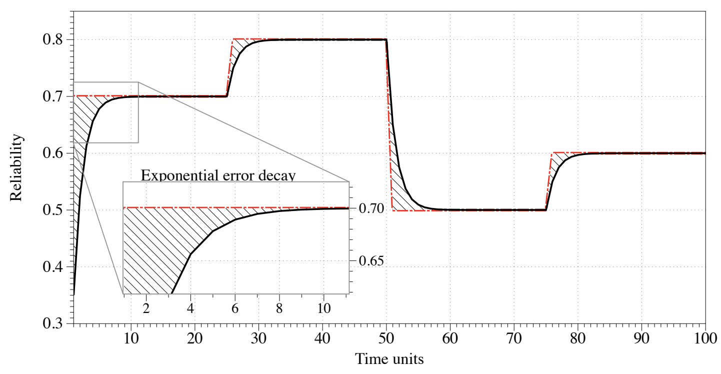


Figure 6.4: Reliability of the system: set point (dashed red) and achieved value (solid).

$$\begin{cases} r_0(k) & = & 0.95 + 0.02stp(k-25) - \\ & & 0.20stp(k-50) + 0.10stp(k-75) \\ r_2(k) & = & 0.95 + 0.02stp(k-20) - \\ & & 0.20stp(k-70) + 0.15stp(k-85) \\ r_3(k) & = & 0.95 + 0.02stp(k-15) - \\ & & 0.97stp(k-55) + 0.50stp(k-65) \\ r_5(k) & = & 0.95 \\ r_8(k) & = & 0.95 + 0.05stp(k-95) \end{cases} \tag{6.13}$$

where $stp(\cdot)$ represents the step function[6] and k counts the time units since start. The changes in the reliabilities are introduced to test the control system ability to respond to external variations. Notice that r_3 goes to zero in the time interval $55 \leq k \leq 65$ therefore accounting for the complete failure of the internal non-iterative filter. The control system sharply counteracts the failure of the *internal direct filter*, changing the control variables as can be noted in Figure 6.5 at time $k = 55$.

This experiment allows us to test the response to both transient behaviors, e.g., small variations of the operating conditions, and to

[6] $stp(x) = 1$ if $x \geq 0$ and 0 otherwise.

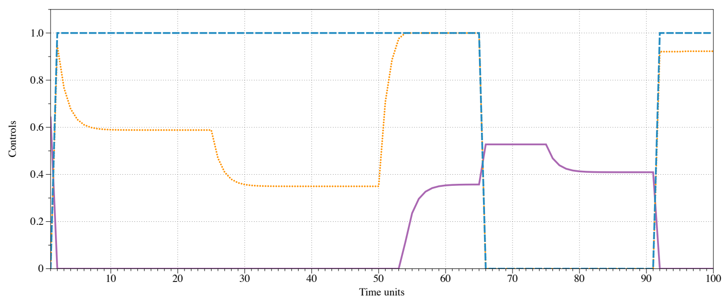


FIGURE 6.5: Control variables of the system: c_{1a} solid, c_{1b} dotted and c_5 dashed dotted.

changes of the operative scenario, e.g., the complete failure of a service.

Noisy Service Reliabilities. Adding a significant white noise in the range $\pm 5\%$ to all the service reliabilities r_i, the output of the process is essentially unchanged (Figure 6.6) though the controller has to carry the burden of promptly counteract the noisy measures $\hat{\mathbf{r}}$, as shown in Figure 6.7.

Notice, in general that convenient estimators for $\hat{\mathbf{r}}$ have also the effect of a low-pass filter (since they usually focus on the average of a set of samples [73]), thus they can compensate to some extent the presence of fast dynamics, such as white noise [184], reducing the effort of the controller.

Nosy Estimation of Global Reliability. For the sake of completeness, this paragraph shows the case where s cannot be perfectly measured. This situation may be either a collateral effect of noisy service reliability estimations or a side effect of numerical approximation in the analytical form of $f(\hat{\mathbf{r}}, \mathbf{c})$.

To empirically show the robustness of the controller to the presence of noise in $\hat{s}$, an artificial white noise bounded in $\pm 1\%$ has been added to the estimation $\hat{s}$ of Figure 6.4. The resulting global relia-

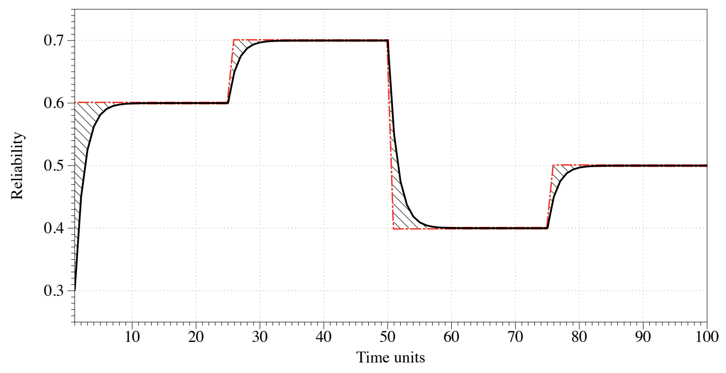


Figure 6.6: Reliability of the system with ±5% white noise on $\hat{\mathbf{r}}$: set point (dashed red) and achieved value (solid).

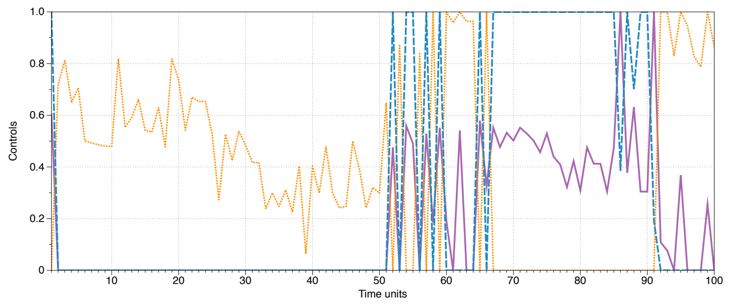


Figure 6.7: Control variables of the system with ±5% white noise on $\hat{\mathbf{r}}$: c_{1a} solid, c_{1b} dotted and c_5 dashed dotted.

bility is shown in Figure 6.8 and the corresponding control signals in Figure 6.9.

As shown in the figures, basing its actions on inaccurate information, the controller cannot perfectly follow the desired set point. On the other hand, the resulting global reliability is remarkably contained around its target value.

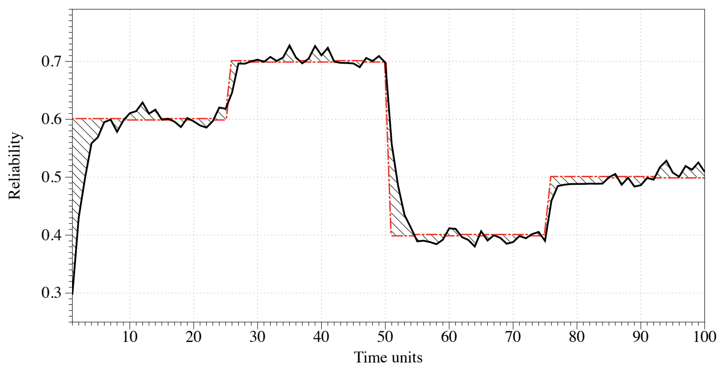


Figure 6.8: Reliability of the system with $\pm 1\%$ white noise on $\hat{s}$: set point (dashed red) and achieved value (solid).

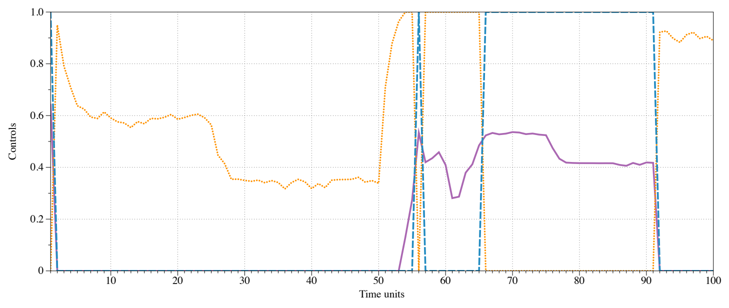


Figure 6.9: Control variables of the system with $\pm 1\%$ white noise on $\hat{s}$: c_{1a} solid, c_{1b} dotted and c_5 dashed dotted.

Changing Cost Function. One may also consider the cost of a service as a time varying parameter of the control system. For example, an experiment can be conducted with a different cost function for the iterative filter: J_{1b}, becomes 100 in the interval $70 \leq k \leq 90$. Figure 6.10 shows the control variables in this case. Notice that the reliability set point is attained, obtaining the same results shown in Figure

6.4. This test shows that the system is able to attain the set point specification and to minimize the cost of the overall solution, also in the presence of different operating conditions; for example changes of service costs.

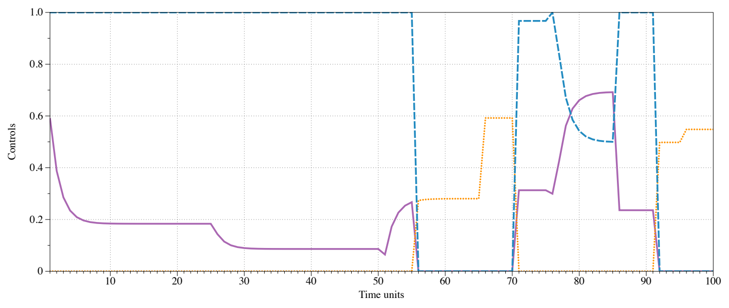


Figure 6.10: Control variables of the system: c_{1a} dashed, c_{1b} solid and c_5 dashed dotted.

Notice that, intuitively, Figures 6.5 and 6.10 are equal except for the mentioned interval where costs are modified. In Figure 6.10 the control system changes the transition probabilities (to make the software system perform as in Figure 6.4) according to the differences in the cost function for the diverse software stages. The overall cost is therefore minimized.

6.5 Extension to R-PCTL and Limitations

The generalization of the control methodology presented in this chapter to the entire flat fragments of PCTL and R-PCTL is straightforward. For example, the expected cumulated cost function before absorption can be controlled by computing the closed-form expression $f(\cdot)$ corresponding to the property $\mathbb{R}_{=?}(\diamond\, \phi)$ through WM and using it in place of s in Equations (6.6) and (6.7). The boundary conditions have to be adjusted accordingly, by adding the constraints on the values of reward parameters, if any. The set of control parameters can then involve both transition probabilities and state rewards, providing a significantly more general application scope.

For example, in [70] the cost of a state describes the deployment cost for the corresponding service. In the assumption of a cloud computing environment, such a cost may assume only a finite set of values, corresponding to the available alternative configurations. Each choice has also a cost decided by the platform provider that can be used to define the cost function $J(\cdot)$.

Nested formulae. On the other hand, this methodology shows severe limitations due to the irregularity of the expression produced by the WM. Indeed, it is no longer a rational polynomial formula but it comes with a set of if-then rules for the run-time update of the parameters α and β (cf. Section 4.2.2.3). This complex shape is in general not suitable for numerical evaluation because an assignment of the control parameters decided on the current operative point may change the values of the α and β invalidating its preliminary assumptions. Hence, despite the validity of its abstract construction, the controller may be practically unsuitable for self-adaptive software.

Limitations. There are two main limitations for the control strategy proposed in this chapter:

- *Structural changes*. The synthesis of the control law is grounded on the WM approach. For this reason, if the structure of the Markov process changes, a new closed-form expression must be computed. This limitation can be mitigated by adding, at design-time, extra parameters accounting for the disconnection of a certain state (i.e. making it unreachable from the initial one) or for creating or deleting certain extra transitions.

Furthermore, a finite number of "spare" states may be introduced in the Markov model to be connected at run time to support a structural change. The price of such a patch is an increased complexity of both the design-time partial evaluation and the runtime control. Leaving these special cases aside, adding an arbitrary number of states remains unfeasible.

- *Optimization time.* Despite its robustness and generality, the implicit control law defined in this chapter involves the solution of an optimization problem. The time required for each step depends mostly on the form of the cost function $J(\cdot)$. Though efficient routines are available for numerical optimization, and considering also that in usual operating conditions only slight variations on the control parameters are expected, the optimization phase may be too slow for quickly adapting systems. This issue is in general domain-specific and there is not a general way out.

- *Contrasting goals.* If a single controller has to ensure multiple properties, the boundary conditions of the optimization problem in Equation (6.7) should be adapted by imposing the exponential decreasing of the error for each of the properties. This operation is trivial in case of non contrasting properties, while could make the optimization unfeasible if two ore more properties are dependent such that decreasing the error of the first leads to an increase on the error of the second. This issue is matter of future work, but a first intuition is that a single comprehensive error function might be studied, whose reduction leads to the satisfaction of multiple goals.

The main strength of this control approach is its generality for all the control problems where 1) the software behavior can be represented by a discrete time Markov process, and 2) the controlled quantitative property can be expressed as a flat (R-)PCTL formula.

In the next chapter, the special case of reliability-driven dynamic binding will be discussed. Though it can be on a first hand handled via the control strategy of this chapter, it will be shown how its characteristics can be exploited to overcome the limitations concerning structural changes and optimization time.

6.6 Related Work

As introduced in Chapter 1, software systems are increasingly required to be self-adaptive. If certain requirements change, they must be able to adapt their behavior accordingly. Moreover, they should be able to detect changes in the surrounding environment they interact with and automatically adapt to prevent or manage violations of their requirements, both functional and nonfunctional.

Looking at the literature on self-adaptive software systems, two observations emerge. First, in the past decades, several areas of Software Engineering have been designing self-adaptive systems. Second, the research community has recently become aware that self-adaptation must become a first-class concern of software engineering methodologies.

Dynamically adjusting strategies for database buffering [89], load-balancing algorithms [195], caching strategies for operating systems [2], graphical user interfaces that adapt to different devices or usages [77], or network routing algorithms [56, 64] are all examples of (self-)adaptive systems. In more recent years, compiler-level advancements have been developed to support adaptive implementations for performance [8, 187] or power [14, 180], and low level architectures are dynamically adjusted and targeted [30, 109, 125, 182].

The main step toward maturation of broad-scope methodologies for self-adaptive software likely came from the effort to reconcile its different enabling disciplines [177]. Besides Software Engineering, Artificial Intelligence and Control Theory are currently playing a crucial role for self-* systems, paving the way for new integrated research fields.

In spite of the analogies between the control systems and self-adaptation, the basic paradigms of control have rarely a straightforward application in software engineering practice [120]. Nonetheless, control theory is capturing an increasing interest from the software engineering community that looks at self-management as a means to meet QoS requirements despite environmental changes and fluctuations of external phenomena [58, 100]. Examples of this trend can be seen in research on control of web servers [117, 145], data centers and clusters management [65, 126], and operating systems [40, 115, 123, 146, 158].

Self-management techniques are also prominent in industry; e.g., companies like IBM [105] (see projects like the IBM Touchpoint

Simulator, the K42 Operating System [123]), Oracle (Oracle Automatic Workload Repository [160]), and Intel (Intel RAS Technologies for Enterprise [108]).

Several approaches in literature try to generalize the use of feedback loops in software engineering [35,154,177]. To the best of the author's knowledge, control of software through a (Markov) model capturing behavioral aspects related to non-functional properties was not explored previously. Moreover, the approach described in this chapter is general for any flat R-PCTL property and can be applied to any system whose behavior can be abstracted by a discrete time Markov process.

Concerning the control of Markov processes, from a control theoretical perspective, most of the approaches in literature cover only special cases. A general control approach for Continuous Time Markov Chains (CTMCs) has been proposed by Brockett [34]. The goal of the controller is to set the value of controllable transition rates of CTMC in order to control the value of selected transition rates, through minimizing a quadratic cost function over the controls.

Reliability-Driven Dynamic Binding 7

There are many different styles of composition. I characterize them always as Mozart versus Beethoven. When Mozart began to write at that time he had the composition ready in his mind. He wrote the manuscript and it was 'aus einem Guss' (casted as one). And it was also written very beautiful. Beethoven was an indecisive and a tinkerer and wrote down before he had the composition ready and plastered parts over to change them. There was a certain place where he plastered over nine times and one did remove that carefully to see what happened and it turned out the last version was the same as the first one.

Edsger Dijkstra

Polymorphism and dynamic binding are fundamental mechanisms to support self-adaptation in Service Oriented Architectures (SOA) [21]. The first allows for the same interface to be provided by different, interchangeable implementations. The second removes the quest for design-time decisions on the binding between abstract operations of a workflow and the concrete implementations that will carry out the execution.

Designing an adaptive SOA application requires a way to (re-)define the bindings at runtime. Indeed, the most convenient bindings usually depend on: 1) software requirements, 2) environmental conditions, and 3) the actual quality of the suitable alternative implementations. All of these factors may change at run time, with all the concerns about uncertainty and unpredictability already stated in previous chapters[1].

In this chapter the focus is posed on the definition of a *per–request* dynamic binding strategy, based on control theory and driven by the satisfaction of a reliability requirement. This strategy is envisioned to overcome the main limitation of Markov models, that is, to allow the dynamic addition of an arbitrarily large number of new implementations at runtime, still keeping a low overhead for the selection process.

In Section 7.1 the problem is formalized for a two-alternatives scenario, defining a convenient modeling strategy for control purpose in Sections 7.1.1 and 7.1.1.1, then a procedure to automatically synthesize the controller in Section 7.1.2, and an auto-tuning strategy to configure it in Section 7.1.3. The core procedure will be validated experimentally in Section 7.1.4. In Section 7.2 the approach is extended to the case of n alternatives. Finally, Section 7.3 presents a few prototype implementations and Section 7.4 concludes the chapter discussing some related researches.

[1]State of the art technologies supporting dynamic binding at runtime are thoroughly discussed in [21, 57].

7.1 Two-Alternatives Online Dynamic Binding

This section explains how dynamic binding can be translated into a discrete-time feedback control problem, by going through the typical control synthesis approach. The problem is first of all formalized as the dynamic decision of directing requests to one out of two possible alternatives (*two-alternatives dynamic binding*), with the goal of following a reliability set-point $r(k) = \bar{r}(k)$[2], where $r(k)$ represents the reliability provided the abstract operation at time k[3].

To achieve this goal, first, a dynamic model of the uncontrolled system is written and described in Section 7.1.1. Subsequently, a regulator is designed to fulfill the required goals, as shown in Section 7.1.2. The analytical formulation of the controller allows rigorous convergence analysis to be performed on the closed-loop system. A further result of this modeling and synthesis process is the formal proof that the controller can avoid oscillations, biases and unnecessary supply of extra quality – that would be costly – by convenient tuning of its control parameters that will be discussed along the chapter.

7.1.1 The Modeling Paradigm

The two alternatives dynamic binding problem can be formalized, on a first order approximation, by the DTMC in Figure 7.1, where requests enter the system through the initial node n_i, at rate w_i, and are then re-routed to different nodes to be served, s_1 and s_2 respectively. Each service node s_j has a success probability p_{s_j}, thus a failure one $p_{f_j} = 1 - p_{s_j}$. The control objective is to continuously adapt the probability p_1 of routing to s_1 (thus also the probability $p_2 = 1 - p_1$ of routing to s_2) to match overall reliability goal. Nodes n_f and n_s respectively represent the failure and the success state. Notice that in the notation of Figure 7.1, actual implementations are identified by labels s, while abstract states are represented by labels n.

[2]A controller designed to follow a set point is also called *regulator*.

[3]Notice that a controller able to get as close as possible to a set point $\bar{r}$ can also be used to satisfy requirements of the type $max\{r(k)\}$ and $r(k) \geq \bar{r}(k)$ by imposing the set points $r(k) = 1$ and $r(k) = \bar{r}(k) + \varepsilon$, respectively (where ε is an arbitrary small positive number).

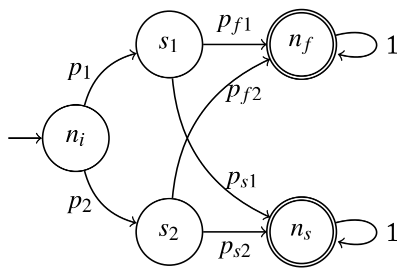


Figure 7.1: DTMC formalization of the basic dynamic binding problem.

The controller is supposed to act periodically, at a fixed time step (or *sampling period*) whose length is T_s time units[4]. The choice of T_s allows to properly formalize the problem in the discrete-time domain, that is, introducing a time index k that counts the controller interventions, and interpreting any quantity $x(k)$ "at (discrete) time k", for every quantity x, as the value of x in the (continuous) time span from $k \cdot T_s$ to $(k+1) \cdot T_s$, when a new value will become available.

Dealing with Service Time. Extending the modeling formalism of Figure 7.1, to each node j, whether abstract or concrete, is associated a request queue. Indeed, in many real-life dynamic-binding applications the processing time of a node may be non negligible in presence of high loads. The assumption made in this chapter is that each node has a maximum throughput per time step and a request queue to store the requests not yet served.

In system-theoretical terms each queue is called a *storage*, and the values of all storages at time k (i.e., the number ν_j of requests in each queue j) form the controlled system's *state vector* $\mathbf{n}(k)$[5]. As for Chapter 6, any quantity that is variable and exogenous for the controlled

[4]The duration of T_s depends on each specific system; if a change in the reliability of the implementations is expected to change (or to be monitored) no faster than each second, T_s should be of about the same order of magnitude. The lower bound value for T_s is the time required to enforce the control decision, that is, after T_s the effect of the control action must be measurable.

[5]Notice that in this approximation, only the length of the queue is considered. Possible priority strategies are not encompassed at this stage.

system is either a *control variable*, if the controller can prescribe its value (e.g. $p_1(k)$ in Figure 7.1), or a *disturbance* if the controller can measure or estimate it, but not prescribe its value (e.g. the input rate $w_i(k)$ and the reliabilities of the two alternatives $p_{s_1}(k)$ and $p_{s_2}(k)$). Control variables and disturbances constitute the input of the controlled system.

On a first hand, suppose that p_{s1} and p_{s2} are "moderately varying with sporadic steps", i.e., that their value undergoes, in each control step of duration T_s, only small variations around a nominal value, while from time to time – but sporadically with respect to the control steps – there may be a large and abrupt variation. The former assumption is quite natural in reliability assessment. Indeed, during its regular operation a service may be subject to slow variations due, for example, to fluctuation in the incoming workload. An abrupt change is instead related to the exceptional situations such as a crash in the software or the failure of hardware or network resources.

If the values p_{s_1} and p_{s_2} do not depend on the time, the controlled system is classified in control theory terms as *time-invariant*. If they can vary during time, as will be the case for dynamic binding, the system is said *time-varying*. In particular, in this chapter it is assumed that p_{s_1} and p_{s_2} are estimated by the services' success and failure rates. Finally, each node j is supposed to have a maximum throughput of t_{mj} requests per control period of length T_s, as already informally anticipated.

Under these assumptions, the dynamic model of the system is defined in the next section.

7.1.1.1 Dynamic Model

In this section a dynamic model of the controlled system is defined.

The first step on the path is to write its *state equations*, i.e., to express the state at time k as a function of the state and the input at time $k-1$. In Chapter 6 it was done starting from the DTMC. Here the dynamic model is extended to consider also the queuing mechanism induced by the throughput saturation.

Denoting by m the number of nodes in the chain – in Figure 7.1 $m = 5$ – the state equations are:

$$\begin{array}{rcl} \mathbf{n}(k) & = & \mathbf{n}(k-1) - \mathbf{r}(k-1) \\ & & +\mathbf{P}(k-1) \cdot \mathbf{r}(k-1) + \mathbf{w}(k-1) \\ \mathbf{r}(k) & = & \min\{\mathbf{t_m}, \mathbf{n}(k)\} \end{array} \qquad (7.1)$$

where bold symbols denote vectors. Each element of $\mathbf{w}$ represents the number of requests entering the corresponding node in the control step: for simplicity assume there is only one entry point at node n_i, i.e. $\mathbf{w} = [w_i 0 0 0 0]'$, but the model already takes into account the possibility of having more than one (by setting the corresponding input rates in $\mathbf{w}$). Vector $\mathbf{n} = [\nu_i\, \nu_1\, \nu_2\, \nu_s\, \nu_f]'$ is the state, while $\mathbf{r}$ represents the number of requests actually served by each node at time k, that is the minimum between those pending in the queue and $\mathbf{t_m}$ – the vector of maximum node throughputs. Each node is supposed to have a possibly different maximum throughput, taking into account the differences in the implementations and capacity of each component of the chain. Finally, $\mathbf{P}$ is the transition matrix of the chain. For the DTMC of Figure 7.1:

$$\mathscr{P}(k) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ p_1(k) & 0 & 0 & 0 & 0 \\ 1-p_1(k) & 0 & 0 & 0 & 0 \\ 0 & 1-p_{s1} & 1-p_{s2} & 1 & 0 \\ 0 & p_{s1} & p_{s2} & 0 & 1 \end{bmatrix} \tag{7.2}$$

where p_{s1} and p_{s2} are reported as constant. In other words, nominal values are assumed for the probability of success and failure of the service nodes, for example based on service level agreements. This assumption will be relaxed later in this section.

The second step toward the dynamic model is to write the *output equation*. As in Chapter 6, this equality relates, instantaneously, the quantities needed to produce the metric(s) of interest – here, reliability – to the system state (and, in general, its input).

The raw output quantities are the number of failures and successes, respectively ν_f and ν_s, measured up to the k-th time step. Thus defining $\mathbf{y} = [\nu_f\, \nu_s]'$, the output equation is:

$$\mathbf{y}(k) = \mathbf{C}\mathbf{n}(k) = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{n}(k) \tag{7.3}$$

Equations (7.1) and (7.3) defines now a *time-invariant nonlinear model* that will be identified by $\mathscr{M}$ from now on. $\mathscr{M}$ is not linear in both the state and the input vectors because of the saturation of the throughputs and to an input-by-state product – the term $\mathscr{P} \cdot \mathbf{r}$ in (7.1).

Finally it comes the *measurement dynamics*, i.e., the output-to-metric relation cascaded to the system model. The purpose of such measurement system ($\mathscr{M}_m$) is to estimate system's reliability (in this chapter identified by the symbol q) from the output $\mathbf{y}$ of the controlled system:

$$q(k) = \frac{\nu_s(k) - \nu_s(k-1)}{\nu_s(k) + n_f(k) - \nu_s(k-1) - \nu_f(k-1)}, \tag{7.4}$$

Intuitively, Equation (7.4) measures system's reliability as the ratio of the number of successful requests to the total requests served, in the last time step.

The requirement investigated in this chapter is the continuous assurance of the desired reliability. If, for example, the goal was to assure the average reliability in a longer time window, the measure in Equation (7.4) can be adapted by shifting the temporal index $k-1$ further in the past.

The state equation of $\mathscr{M}_m$ is:

$$\mathbf{x}_m(k) = \mathbf{u}_m(k-1), \tag{7.5}$$

where the input $\mathbf{u}_m$ and the state $\mathbf{x}_m$ are both given by the output $\mathbf{y} = [\nu_f\ \nu_s]'$ of the controlled system. The output of the measurement system is defined by Equation (7.4). Notice that $\mathscr{M}_m$ is nonlinear because of the output relation only.

7.1.1.2 Linearization

In Section 7.1.1 it has been assumed that system's reliability may be subject to significant variations only sporadically, while most of the time it fluctuates or varies slowly around an equilibrium. In control theory terms, this problem can be cast into the framework of *control in the proximity of an equilibrium*, indicating that the system needs to be brought to the equilibrium when an abrupt change occurs, and subsequently kept in its proximity.

Within such framework, it is possible to analyze the equilibria of the system for constant inputs and construct a linearized model valid in the proximity of a generic equilibrium. Then, a controller suitable for any equilibrium is devised. More precisely, a strategy is devised to obtained a different controller *parameterization* for any equilibrium .

Observing that $\mathscr{M}$ and $\mathscr{M}_m$ are cascaded, it is convenient to treat them separately, and then join the results. Starting with $\mathscr{M}$, Equation (7.1) can be written in the form:

$$\mathbf{n}(k) = \Phi(\mathbf{n}(k-1), \mathbf{u}(k-1), \mathbf{d}(k-1)) \tag{7.6}$$

where $\mathbf{u} = p_1$ is the control input and $\mathbf{d} = w_i$ the disturbance. Assuming to receive constant inputs $\overline{\mathbf{u}}$ and $\overline{\mathbf{d}}$, the corresponding equilibrium states $\overline{\mathbf{n}}$ are the solutions $\overline{\mathbf{n}}$ of the following equation:

$$\overline{\mathbf{n}} = \Phi\left(\overline{\mathbf{n}}, \overline{\mathbf{u}}, \overline{\mathbf{d}}\right) \tag{7.7}$$

that, specialized in the case of $\mathscr{M}$, becomes:

$$(\mathscr{P}(\overline{\mathbf{u}}) - \mathbf{I}) \min\{\mathbf{t_m}, \overline{\mathbf{n}}\} + [w_i 0 \cdots 0]' = \mathbf{0} \tag{7.8}$$

where $\mathbf{I}$ represents the identity matrix.

Matrix $\mathscr{P}(\overline{\mathbf{u}}) - \mathbf{I}$ is structurally singular, and it can be verified that no equilibrium exists. This is correct, as the absorbing nodes in Figure 7.1 apparently accumulate (served) requests indefinitely.

If however the accumulation is neglected, by removing the self-loops on success and failure states, the matrix $\mathscr{P}$ in Equation (7.8) can be replaced by the following reduced transition matrix $\mathscr{P}_r$:

$$\mathscr{P}_r = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ p_1 & 0 & 0 & 0 & 0 \\ 1-p_1 & 0 & 0 & 0 & 0 \\ 0 & 1-p_{s1} & 1-p_{s2} & 0 & 0 \\ 0 & p_{s1} & p_{s2} & 0 & 0 \end{bmatrix}, \tag{7.9}$$

Then an equilibrium can always be obtained from (7.8)[6]:

$$\begin{aligned} \overline{\mathbf{n}} &= [\overline{v}_i \, \overline{v}_1 \, \overline{v}_2 \, \overline{v}_s \, \overline{v}_f]' = (I - P_r(\overline{\mathbf{u}}))^{-1} \overline{\mathbf{d}} \\ &= \begin{bmatrix} 1 \\ p_1 \\ 1-p_1 \\ p_1(1-p_{s1}) + (1-p_1)(1-p_{s2}) \\ p_1 p_{s1} + (1-p_1) p_{s2} \end{bmatrix} \overline{w}_i \end{aligned} \tag{7.10}$$

[6]The reader might have noticed the similarity between matrix $\mathscr{P}_r$ and the matrix Q describing the transition probabilities among transient states of a DTMC. The underlying purpose of the two approaches is indeed the same, that is, intuitively, manage separately the transient and the absorbing states.

Notice that Equation (7.10) represents a valid equilibrium under the assumption that $\mathbf{r} = \mathbf{n}$ (or, equivalently, $\mathbf{t_m} \geq \mathbf{n}$). This means that the system is actually able to satisfy all the incoming requests. The latter assumption can be violated only for a limited time, otherwise there cannot be any equilibrium, because one or more queues will grow indefinitely. Notice that the equilibrium (7.10) computed on $\mathscr{P}_r$ is valid also for the original system, under the same assumptions, by just interpreting v_s and v_f as the successful and failed requests in the last period, thus (re-)defining the reliability measure in the same period as:

$$q(k) = \frac{v_s(k)}{v_s(k) + v_f(k)} \tag{7.11}$$

Defining $\delta\mathbf{n} = \mathbf{n} - \overline{\mathbf{n}}$, $\delta\mathbf{u} = \mathbf{u} - \overline{\mathbf{u}}$, $\delta\mathbf{d} = \mathbf{d} - \overline{\mathbf{d}}$ and $\delta\mathbf{y} = \mathbf{y} - \overline{\mathbf{y}}$, the linearized model of the system based on $\mathscr{P}_r$ is:

$$\left\{ \begin{array}{rcl} \delta\mathbf{n}(k) & = & \mathbf{A}\delta\mathbf{n}(k-1) + \\ & & \mathbf{B}_u\delta\mathbf{u}(k-1) + \mathbf{B}_d\delta\mathbf{d}(k-1) \\ \delta\mathbf{y}(k) & = & \mathbf{C}\delta\mathbf{n}(k) \end{array} \right. \tag{7.12}$$

where:

$$\begin{gathered} \mathbf{A} = \left.\frac{\partial\Phi_r}{\partial\mathbf{n}}\right|_{\overline{\mathbf{n}},\overline{\mathbf{u}},\overline{\mathbf{d}}}, \quad \mathbf{B}_u = \left.\frac{\partial\Phi_r}{\partial\mathbf{u}}\right|_{\overline{\mathbf{n}},\overline{\mathbf{u}},\overline{\mathbf{d}}}, \\ \mathbf{B}_d = \left.\frac{\partial\Phi_r}{\partial\mathbf{d}}\right|_{\overline{\mathbf{n}},\overline{\mathbf{u}},\overline{\mathbf{d}}} \end{gathered} \tag{7.13}$$

are respectively the Jacobian matrices of Φ_r (defined Φ in Equation (7.6) but where $\mathscr{P}_r$ replaces $\mathscr{P}$) with respect to $\mathbf{n}$, $\mathbf{u}$, and $\mathbf{d}$, computed at the equilibrium, and $\mathbf{C}$ has been defined in (7.3).

Matrix $\mathbf{A}$, as it would be expected, equals $\mathscr{P}_r$, while:

$$\begin{array}{rcl} \mathbf{B}_u & = & \left[\begin{array}{ccccc} 0 & \overline{v}_1 & -\overline{v}_1 & 0 & 0 \end{array}\right]' \\ \mathbf{B}_d & = & \left[\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \end{array}\right]' \end{array} \tag{7.14}$$

Concerning $\mathscr{M}_m$, the re-definition of q provided in Equation (7.11) makes the measurement system simply algebraic. In other words there is no longer the need to keep an internal state, as required to compute q according to Equation (7.4), since the output is instantaneously and algebraically related to the input.

Furthermore, any constant input $\overline{\mathbf{u}}_m$ yields an equilibrium output $\overline{q} = \overline{v}_s/(\overline{v}_s + \overline{v}_f)$. Then, following the same procedure used for the output equation of $\mathscr{M}$, the linearized (algebraic) model of $\mathscr{M}_m$ is defined as:

$$\delta q(k) = \mathbf{D}_m \delta \mathbf{u}_m(k) \tag{7.15}$$

where $\delta \mathbf{u}_m = \mathbf{u}_m - \overline{\mathbf{u}}_m$, $\delta q = q - \overline{q}$, and:

$$\mathbf{D}_m = \begin{bmatrix} \frac{\overline{v}_f}{(\overline{v}_s+\overline{v}_f)^2} & -\frac{\overline{v}_s}{(\overline{v}_s+\overline{v}_f)^2} \end{bmatrix}. \tag{7.16}$$

Putting all the pieces of the cascade system together, a complete linearized dynamic model can be defined as:

$$\begin{cases} \delta \mathbf{n}(k) & = & \mathbf{A}\delta \mathbf{n}(k-1) \\ & & +\mathbf{B}_u \delta \mathbf{u}(k-1) + \mathbf{B}_d \delta \mathbf{d}(k-1) \\ \delta q(k) & = & \mathbf{C}_m \delta \mathbf{n}(k) \end{cases} \tag{7.17}$$

where:

$$\begin{array}{rcl} \mathbf{C}_m & = & \mathbf{D}_m \mathbf{C} \\ & = & \begin{bmatrix} 0 & 0 & 0 & \frac{\overline{v}_f}{(\overline{v}_s+\overline{v}_f)^2} & -\frac{\overline{v}_s}{(\overline{v}_s+\overline{v}_f)^2} \end{bmatrix}. \end{array} \tag{7.18}$$

Applying the Z-transform to converts the discrete-time domain of (7.17) into a frequency domain, the transfer function from δp_1 to δq is readily computed as:

$$\begin{array}{rcl} P(z) & = & \mathbf{C}_m (z\mathbf{I} - \mathbf{A})^{-1} \mathbf{B}_u \\ & = & \overline{v}_1 \frac{\overline{v}_s(p_{s2} - p_{s1}) - \overline{v}_f(p_{f2} - p_{f1})}{(\overline{v}_s + \overline{v}_f)^2} \frac{1}{z^2} \end{array} \tag{7.19}$$

Considering the dependencies among probability values, Equation (7.19) can be simplified to:

$$P(z) = \frac{p_{s2} - p_{s1}}{z^2} \tag{7.20}$$

Equation (7.20) reveals some information relevant to control. First, the *gain* of the system is the difference of the service nodes' success probabilities, thus (correctly) zero if they are equal, since in that case no routing action can alter the overall reliability. Second, and most important, the structure of the controlled dynamics is invariantly that of a two-steps delay, allowing for a simple control law as the one that will be introduced in the following sections. On the other hand, since the *sign* of the controlled system's gain can change, most likely no single controller parameterization will be suitable for all situations, and an on-line estimation of service nodes success probabilities will be exploited to adapt the controller when needed. Notice however that, in general, the only estimation needed to make the controlled system eventually meet its goal concerns the sign of the transfer function; of course "eventually" is not enough and the efficiency of the controller will be taken into account while designing it in the next section.

7.1.2 Control Synthesis

Concerning the system in Figure 7.1, the goal of the controller is to set the value of $p_1(k)$ in order to obtain $q(k) = \bar{q}(k)$.

Based on equation (7.20), established control theory results allow to state that zero steady-state error and a high degree of stability can be achieved by a PI (Proportional plus Integral) controller [61], whose dynamic model is:

$$\begin{array}{rcl} u_i(k) & = & u_i(k-1) + a(1-b) \cdot e(k-1) \\ p_1(k) & = & u_i(k) + a \cdot e(k) \end{array} \tag{7.21}$$

where $e(k) = \bar{q}(k) - q(k)$ is the error[7].

In Equation (7.21), two parameters govern the behavior of the controller, namely the coefficients a and b. The value of these two controller parameters defines the performance of the controller in terms of effectiveness, time to converge, robustness, stability, and overshooting avoidance.

The Jury criterion [25] reveals that after a convenient assignment of a and b, asymptotic stability of the closed-loop system composed of (7.20) and (7.21) holds for any value d of the difference $p_{s2} - p_{s1}$, of course $d \in (-1, 1)$, such that:

[7]The reader interested in an extensive introduction on PI(D) control can refer e.g. to [12] and the vast bibliography provided therein.

$$\begin{cases} \dfrac{1-abd}{1+abd} & > \quad 0 \\ \dfrac{(a^2b^2d^2-abd+ad-1)(a^2b^2d^2+abd-ad-1)}{(1-abd1)(1+abd)} & > \quad 0 \\ \dfrac{ad(1-b)(abd+ad+2)(a^2b^2d^2-abd+ad-1)}{a^2b^2d^2+abd-ad-1} & > \quad 0 \end{cases} \tag{7.22}$$

Studying (7.22) it can be noticed that for a wide range of values for a and b, stability is preserved under the sole condition $ad > 0$, thus that even relevant estimation errors for d do not produce disrupting effects if at least the sign is caught. Of course, this is not true for control *performance*: for example, the time required to recover from a disturbance can degrade significantly if the estimation of d is not good enough.

The values of a and b are specific for each equilibrium and allow the controller to keep system reliability in the proximity of the goal. If the equilibriium changes, because for example of abrupt changes in the reliability of one or both alternatives, a new value for the controller parameter must be set.

In common practice of control application, setting the values of a and b is referred to as the *tuning* of the controller. Tuning can be done manually, deciding the value of parameters on the base of their role in the PI. A first approach to define the values of the controller parameters a and b could be the identification of their relation to d and some performance specifications. However this requires a deeper knowledge of this family of controller, ad usually some experience. These skills maybe limiting in most software development contexts, therefore in the next section a procedure to automatically determine the most appropriate values of a and b is discussed.

7.1.3 Auto-Tuning

In this section an *auto-tuning* mechanism is introduced, whose high-level block diagram appears in Figure 7.2. The purpose of the auto-tuning mechanism is to automatically update the controller parameters to the current conditions. This means that if the reliabilities of the different services radically change, the parameters of the controller need to be tuned accordingly.

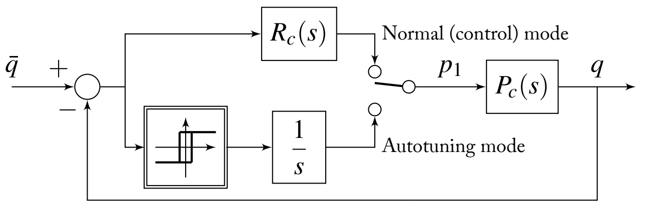


FIGURE 7.2: Block diagram for relay-based (PI) auto-tuning.

To define the auto-tuning procedure, the discrete-time transfer function (7.20) is first re-interpreted as continuous-time and sampled at period T_s, yielding:

$$P_c(z) = d\,e^{-2T_s s}, \quad R_c(s) = K\left(1 + \frac{1}{sT_i}\right) \tag{7.23}$$

where s is the Laplace transform complex variable, d is the difference defined above, and:

$$a = K\left(1 + \frac{T_s}{T_i}\right), \quad b = \frac{T_i}{T_i + T_s}. \tag{7.24}$$

The applied methodology is based on relay feedback, see e.g. [138] for background material. More in detail, by replacing the feedback controller with a relay cascaded to an integrator, the point of the frequency response $P_c(j\omega)$ – where j is the imaginary unit and ω the frequency – with phase $-90°$ is easily found from the characteristics (frequency and amplitude) of the sustained oscillation induced on the controlled variable. This technique, commonly referred to as *relay feedback identification*, is known to provide useful auto-tuning information rapidly and with a very modest system upset.

Once the mentioned frequency response point is determined, assuming from control specifications a desired phase margin $\overline{\varphi}_m$ (in degrees), the parameters of R_c in (7.23) are obtained by solving the complex equation:

$$R_c(j\overline{\omega}) \cdot \overline{P}_\omega\, e^{-j90°} = e^{j(180° - \overline{\varphi}_m)} \tag{7.25}$$

where $\overline{\omega}$ is the oscillation frequency, and $\overline{P}_\omega$ the correspondingly measured frequency response magnitude, see e.g. [139].

In general, the desired phase margin for the controller, $\overline{\varphi}_m$, is in the interval $(0,90]^\circ$. Lower values privilege response speed versus absence of oscillations and degree of stability, while higher values do the reverse. By experience, 60° has proved to be a reasonable default value in the conducted experiments.

When parameters of the continuous-time controller $R_c(s)$ are identified, the values of a and b can be derived by Equation (7.24).

In order to simplify the specification of a phase margin for non-specialist users, it could be simpler to define a "desire knob" whose value is between 0 and 1. 0 corresponds to the request of minimum time for both the response to desired reliability variations and the rejection of disturbances at the possible cost of oscillatory transients and diminished stability degree; 1 calls for maximum stability and transients' smoothness, at the potential cost of response time. The value of the knob can then be used to select the actual phase margin in a reasonable range, such as 40° to 80°, by linear interpolation ($40 \cdot x + 40$ for all the knob values $x \in [0,1]$).

7.1.4 Control Validation

Before the implementation of the controller in a real software system, a simulation campaign was conducted to provide empirical support of its effectiveness in a controlled environment.

The results of one of the simulations from the campaign are reported in Figures 7.3 to 7.6. The MATLAB simulator is started asking for 10000 simulation steps, each node can serve maximum 100 requests per step and the system is required to continuously provide a reliability of 0.9. The initial failure probability of the first service is 0.4 while the one for the second service is 0.1.

A few variations and disturbances are injected into the system:

- to see how the controller reacts to changes in the *set point*, the simulation time has been divided into three intervals, and during the second interval the requested reliability has been diminished to 0.8.
- the *reliability of the two services* have been changed four times, at regular intervals, to simulate different variability scenarios; for example, the complete failure of the first service has been simulated between time units 2000 and 4000. The complete pattern of changes is shown in Figure 7.5.

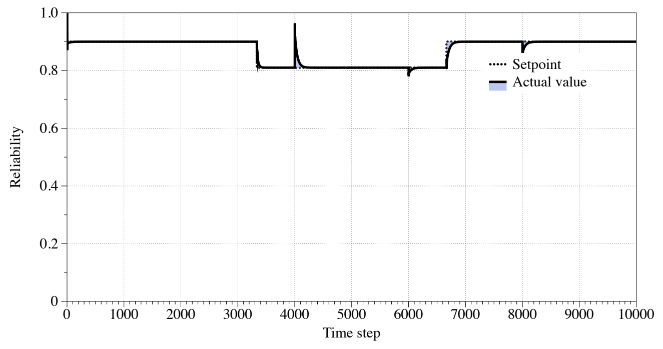


Figure 7.3: Simulation: overall reliability.

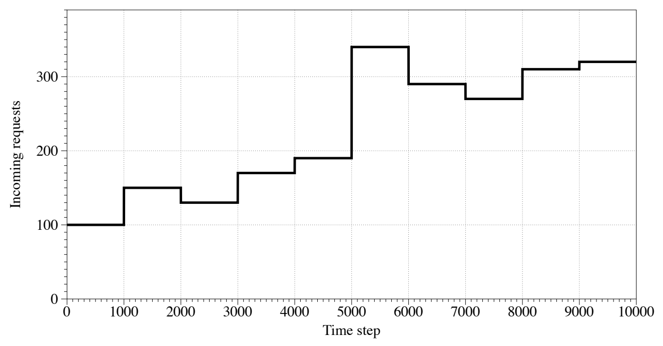


Figure 7.4: Simulation: incoming requests.

- the *number of requests* entering the system is changed according to a predefined pattern, to see the reaction of the system to different loads.

In most of the cases the controller has been able to compensate disturbances in a completely transparent way. The largest divergence between the required behavior and the one obtained can be seen in Figure 7.3 at time unit 4000: when the first service node is back to its normal operations, there is a spike in reliability that is immediately compensated by the control action shown in Figure 7.6.

The MATLAB implementation that has been used to perform

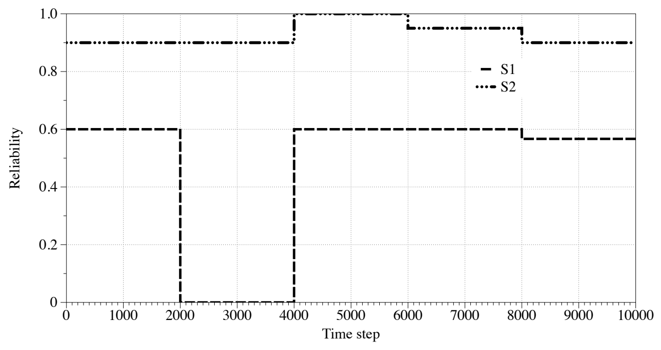


Figure 7.5: Simulation: services reliabilities.

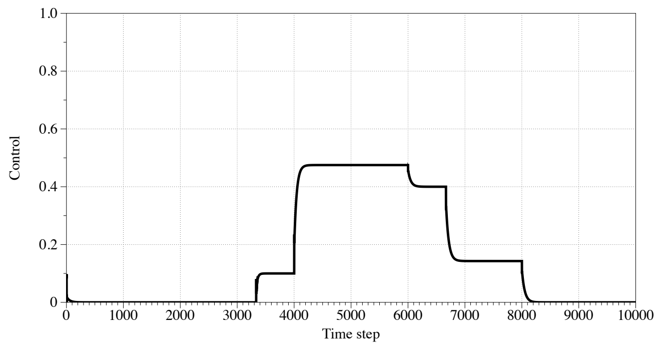


Figure 7.6: Simulation: control.

the experiments will be discussed later in Section 7.3.

7.2 Extending to *n* Alternatives

Thanks to its inherent modularity, the control approach presented in the previous section for the two-alternatives case can be extended to the *n*-alternatives case in a natural manner. In 1976 D. Knuth proved that every multinomial distribution can be equivalently reproduced by conveniently combining binary probabilistic choices [118]. In a similar fashion, to build a *n*-alternatives selector, it suffices to apply the scheme of Section 7.1.2 hierarchically in order to build a binary selection tree whose leaves are the *n* implementations that can be targets of the binding and internal nodes are two-alternatives selectors. This leads to a structure like the one shown in Figure 7.7, composed of controllers like the one devised for the two-alternatives case, where all the controllers try to follow the same set point corresponding to the global target reliability.

Notice that the "intermediate" nodes can be considered as fictitious, since they are only used to compute the probabilities of routing from the input node to the possible targets.

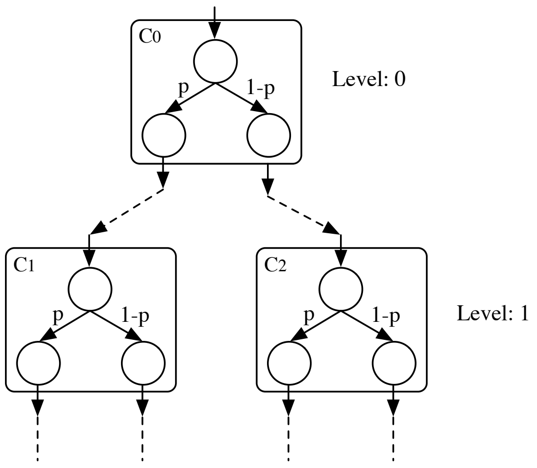


FIGURE 7.7: *n*-alternatives binding structure.

The composition of PI controllers to create a hierarchical control structure requires careful setting of the parameters for the different control elements. To understand the issue, assume the structure of

Figure 7.7 is implemented and both arrows exiting from C_1 are connected to concrete executors, S_1 and S_2. These executors have their own reliabilities, respectively r_1 and r_2. The reliability r_{C_1} provided by C_1 can be computed as $p_1 \cdot r_1 + (1 - p_1) \cdot r_2$, where p_1 is the value produced by the controller. When r_{C_1} does not meet the reliability requirement, the controller C_1 can only adjust the value of p_1 in order to get as close as possible to the target. This adjustment typically takes a few time steps to be completed, depending on the configuration of the controller (i.e., the values of K and T_i).

Suppose now that the system is running and satisfies the overall reliability target. Consider a scenario where r_1 decreases sharply, for example due to a complete failure of the service S_1. Assuming that all two-alternatives controllers adopt the same time-step to query their siblings, the violation of the requirement due to S_1's failure is almost immediately propagated upwards from C_1 and therefore it is perceived by both C_1 and C_0, instantaneously triggering their reactions. Simultaneous changes in the decision of C_0 and C_1 could interfere with one another, delaying the solution and possibly introducing oscillations in the global reliability of the system.

A better solution for the problem is to allow the controller that is closer to the source of the violation to react first. In this case, if possible, C_1 would compensate the failure of S_1 by redirecting the load to S_2. If the compensation is not possible, it would still provide the reliability value closest to the set point that can be obtained at that level in the tree. Only at this point, if still needed, the intervention of the higher level controller C_0 should be triggered.

This scenario naturally generalizes to more complex selection trees and can be solved applying a *multirate* control strategy. The term means that for each level in the hierarchy, the corresponding controllers act with different time periods, i.e., at different rates. Precisely, higher-level nodes in the routing tree would intervene slower with respect to lower ones, and their control period would just need to be changed accordingly. To simplify the design of n-alternatives selectors, the PI controllers can still share the parameters, introducing a further "scaling" factor, identified with the integer parameter τ_{T_s}. This means that each tree level exerts its control action every τ_{T_s} steps with respect to the lower one. τ_{T_s} can be interpreted as the number of time steps required by a controller node to stabilize the control signal and therefore the reliability of the correspondent part of the tree. Multirate systems are an established research field in control theory, and powerful analysis and synthesis techniques are available [9]. In this case, the use of such a strategy allows to avoid the issues generated by mutual interference of the controllers.

The response to a step variation of a five alternatives binder is shown in Figure 7.8, where the short convergence time required to meet the goals can be visually appreciated.

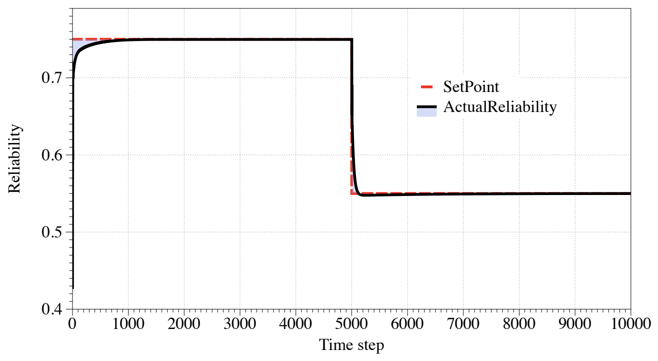


FIGURE 7.8: Step response of a 5-alternatives binder.

Concluding, the selection problem among n different alternatives scales easily even for a large number of alternatives. The selection algorithm just requires to assign to each request a token in $[0, 1]$ an then traverse the selection tree by comparing the token with the control variable of each controller. The temporal complexity of binding a request is thus $O(\log(n))$. Furthermore, the control decision for each controller has to be updated at most every time step (for the controllers closest to the leaves) and is accomplished in constant time by simply 6 floating point operations (cf. Equation (7.21)).

The main drawback for the n-alternatives case concerns the applicability of an auto-tuning strategy, since it involves in this case to manage a multivariable system, whose components interact one another. Experiments have shown that the auto-tuning procedure devised for the two-alternatives case is not replicable *as is* in the n-alternatives context, and more advanced synthesis techniques need to be devised. The presented *preliminary* solution based on multirate control already shows that the system can work satisfactorily also with “hand made” tuning, thereby proving that the extension is practically feasible, and the tuning problem is well posed from the system-theoretical standpoint. As for now, the problem stands however open, and is being addressed with methodologies specifically aimed at multivariable control. Also, more advanced adaptation mechanisms are being studied, grounded on well established control-theoretical methods such as that proposed in [137].

7.3 Implementation

To validate the dynamic-binding approach described in this chapter, the proposed algorithms have been implemented in three different platforms. The produced artifacts can be downloaded from [69].

A Matlab implementation is available for simulation purposes[8]. Numerical mathematic programming is an established instrument for control experts to study controller's performance by simulating disturbances and process dynamics.

The second implementation is based on the Spring Framework [112]. Spring is considered one of the most complete lightweight container for J2EE applications. Its Aspect Oriented Programming (AOP) functionalities have been exploited to show how a this dynamic-binding methodology can be integrated in real life applications with a low impact on development organization. Monitoring and control can be defined as a specific aspect of the application, requiring no changes on the existing code. Indeed, an *around* advice of AOP allows for performing custom behavior before and after the invocation of an existing method, resulting in a natural environment to engraft monitoring and control mechanisms. The impact on performance is definitely low thanks to the complete integration of the framework in Spring (further details in [112] or on the Spring Framework website [1]). A running instance of the Spring implementation is also accessible from [69], with a web-interface to ease the demonstration.

Finally, to show how to build an n-alternatives selector in Java, a simple prototype has been implemented which automatically arranges the available executors in a balanced binary tree and applies the control laws presented in Section 7.2.

Notice that, though it is theoretically possible to obtain effective control with any binary tree, the choice of a balanced (or almost balanced) tree proved in the conducted experiments to be easier to configure and more efficient. Indeed the tree is set up by applying to each control node a sampling time of $\tau_{T_s}^{d}$, where $d \geq 0$ is the distance from the leaves.

The Matlab and the Spring implementations support the autotuning procedure described in Section 7.1.

[8]An equivalent implementation is also available for Scilab, which is a widely used open-source Matlab counterpart [178].

7.4 Related Work

Dynamic binding for Web services is emblematic of many situations in which multiple implementations for the same abstract operation are available and the actual execution of incoming requests has to be delegated to one of them. The selection criteria are usually based on cost, QoS, or both.

Most of the current approaches address this problem through mathematical optimization, where different qualities are traded-off, looking for an optimal, or at least satisfactory, solution [11,24].

Some approaches allow the formulation of an optimization problem for each operation in order to select the best candidate with respect to a local objective function [10, 197]. Local approaches are usually efficient because in most of the practical cases the candidates for each single operation are not in a large number. On the other hand, most of the QoS requirements are expressed at application level, thus shrinking the scope down to single operations my produce sub-optimal solutions with respect to the global system, or, conversely, they may overshoot producing (possibly costly) better-than-required solutions. Other approaches allow for managing the optimization of a global objective, considering the entire control space at once [110, 198]. The immediate negative effect is in terms of complexity: considering in a single optimization all the possible alternative bindings of each operation leads to a combinatorial explosion. In practice, these approaches are either unfeasible for even small cases or too complex to support binding control. Besides the growth in the exploration space, the non linearities of the global problem may be untreatable with standard mathematical procedures and may require the adoption of soft computing techniques, such as genetic algorithms [39], which are not always suitable for run time use and rarely provide formal guarantees of their effectiveness.

Some recent approaches combine both local and global techniques to so to improve the performance of global search by feeding in locally optimal bindings of all or part of the operation level selections (e.g. [5,142]).

Most of the optimization-based approaches consider multiple QoS metrics simultaneously. The control theoretical dynamic-binding strategy presented in this chapter has been design for reliability requirements (with the generality provided by the domain-specific notions of *success* and *failure*). Nonetheless it is successfully being extended to other properties and experimental evaluation is already in progress.

Comparing with Chapter 6, that controller was capable of trading off reliability and costs by solving an optimization problem. Further properties can also be embedded in the control problem, allowing to manage different quantitative properties at the same time. The complexity of the adaptation mechanism did not depend on the number of states of the Markov process but only on the type of objective function and on the number model parameters. Such an optimization problem could be complex enough to make certain systems loose the ability to timely adapt when their requirements are violated. Another limitation of the controller of Chapter 6, inherited by the WM approach, consists in the inability of adding an arbitrary number of new states to the Markov model without re-computing the closed form expression for the optimization. The adoption of a simpler controller proposed in this chapter overcomes these two limitations and allows for timely adaptations even on low-end or mobile devices. For the proposed controller is also possible a formal assessment of its effectiveness (cfr. Section 7.1).

Finally, concerning the application of control theory to achieve continuous QoS assurance, in the close field of load-balancing, a comparison between optimization based and control-theory based techniques has been performed in [196]. Though the MIMO controllers used to balance the load among DB2 instances in [196] was more complex than a PI, the effectiveness of the feedback loop overwhelmed the optimization-based techniques, particularly in the situation of highly variable loads, where efficient continuous adjustments leaded to a smother performance curve, with reduced outliers and faster convergence time.

Part V

Finale

Conclusions and Future Work 8

Clarity, above all, has been my aim. I prefer a clear statement subsequently disproved to a misty dictum capable of some profound interpretation which can be welcomed as a 'great thought.' It is not by 'great thoughts,' but by careful and detailed analysis, that the kind of technical philosophy which I value can be advanced.

Bertrand Russell

In this chapter conclusive remarks and future research directions are summarized for each of the main contributions.

Run Time Efficient Probabilistic Model-Checking.
The problem of efficient run time model-checking of quantitative properties on D-MRMs system abstractions has been discussed in Chapter 4.

According to the WorkingMom paradigm, the verification problem has been split in two phases to be carried out at design time and at run time, respectively. The role of the design time phase is to partially evaluate the problem, leaving at run time a residual task as simple as evaluating a closed form expression.

Besides a theoretical complexity assessment of the design time partial evaluation algorithms, an implementation has been empirically compared with other related approaches, showing a significant improvement with respect to known results, and a reasonable execution time, even on general purpose hardware.

The resulting closed form polynomial expression can be efficiently evaluated by replacing the model parameters with their actual values, with no need to execute complex mathematical routines at run time. This enables the run time phase to be performed even on low power mobile devices.

Several improvement directions are currently under investigation:

- The experimental assessment presented in Section 4.6 shows that the number of states and number of parameters do not satisfactorily characterize the actual performance of the equation-based partial evaluation algorithms. This is clear by looking at the difference between maximum and average execution time, which is an evidence of the large variance of the experimental outcomes. This suggests that the dependency of design-time performance on the topology of the D-MRM should be further investigated.

- Matrix-based approaches are naturally parallelizable for high performance processing environments. An experimental campaign would be needed to assess the actual application scope of similar implementation.

- The current implementation is being enhanced by a set of pre-processing techniques aiming at state space reduction for the input models [15,17,116]. Other divide-et-imperat heuristics are also being adapted from [45] in order to split the partial evaluation problem in smaller, possibly parallelizable, subproblems.

- The application of the WM paradigm is currently under investigation for the analysis of Continuous Time Markov Models, which support a more expressive analysis of the execution time of system.

- The WM is also supporting other related researches such as [82] in the field of Software Product Lines, and [152] to speed up the Monte Carlo analysis of parametric DTMC models.

Sensitivity Analysis at Run Time.
The generation of closed-form expressions corresponding to the probability of satisfying a quantitative property has been exploited in Section 4.4 for the efficient sensitivity analysis with respect to the model

parameters. The results of the analysis can support both the improvement of the system and the diagnosis of failures, and can be brought at run time.

Some applications of this research are currently under investigation:

- The use of sensitivity results to support adaptation planning. Sensitivity is currently used at design time to rank the conditions that make a system violate its quantitative requirements. On the other hand, in a number of practical situations sensitivity is by itself a key driver to decide basic counter actions [55]. This suggests the possibility to define sensitivity-based adaptation strategies to be carried out at run time.

- Since the sensitivity corresponds to the gradient of the system quality with respect to model parameters, it can in principle be used to design optimal controllers based on a gradient-descent approach [168]. These controller are usually more efficient than the one proposed in Chapter 6, though the gradient descent strategy is used to get stuck into local optima.

Syntax-Driven Incremental Quantitative Analysis.
A framework for the definition of incremental analysis procedures has been introduced for software artifacts. The analysis is driven by the syntactic structure of the software and encoded as the synthesis of semantic attributes. Incrementality is then automatically achieved by coupling the evaluation of semantic attributes with an incremental parsing technique. Syntax-driven incremental analysis is investigated in Chapter 5.

Chapter 5 introduced SiDECAR, a framework for the definition of verification procedures, which are automatically enhanced with incrementality by the framework itself.

SiDECAR supports verification procedures encoded as synthesis of semantic attributes associated with a formal language. The attributes are evaluated by traversing the syntax tree that reflects the structure of the software system. By exploiting incremental parsing and attributes evaluation techniques, SiDECAR reduces the complexity of the verification procedure in presence of changes, thus providing a speed-up in performance.

A procedure for quantitative analysis of structured programs has been defined, to show the effectiveness of SiDECAR for this task.

Future work will address the following main directions:

- Definition of verification procedures for more functional and quantitative properties. Safety and reachability analysis are already at a prototypal stage, while other properties are under investigation.

- Support for run-time changes of the language (and thus its grammar) in which the artifact to be verified is described. This capability would support the application of SiDECAR to more advanced adaptiveness scenarios.

- Support for changes in the properties to be verified, and still exploit the benefits of incremental verification. As in the case described in this thesis, only the model is allowed to change, not the property under analysis, though in adaptive systems also requirements may change.

- An efficient implementation is under development, which incorporates also the parallel parsing techniques of [19]. The implementation will also allow an empirical comparison with other verification techniques.

Software Control through Markov Models.
In Chapter 6, a general methodology has been defined for the control of tunable software whose behavior can be described by a discrete time Markov model.

The controller is systematically defined starting from the parametric D-MRM and a flat R-PCTL property defining the quality goal. The control strategy has been proved to be robust to sudden changes in the environment and to monitoring inaccuracy. Another significative strength of this approach lies in its application scope. Indeed, it is not tailored to the solution of a specific problem, but can handle a wide set of situations that can be represented by the probabilistic models of Chapter 2.

The main limitations of this approach are: 1) the support for changes in the model structure is limited because the generation of the dynamic model is built upon the WM approach, and 2) the complexity at run time depends on the cost function used to select the optimal control settings.

Future research directions include:

- Multi-objective control and unfeasibility: with the current controller it is naturally possible to consider multiple objectives in the control strategy by conveniently adjusting the optimization problem. The main issue appears when contrasting goals make the problem unfeasible. A constraint relaxation strategy is under validation that provides support for graceful degradation of the system quality when the required goals are unfeasible.

- Some local optimization strategies, as well as some special families of cost functions, are being investigated to improve the run time efficiency of the controller.

Reliability Driven Dynamic-Binding.
Dynamic-binding has been considered as a mean for a service to satisfy reliability requirements by continuously adjusting the choice among the available concrete implementations. Control-theoretical analysis and synthesis have been applied to prove effectiveness, stability, and scalability of the controller, and an auto-tuning procedure has been defined to set its configuration, even at run time. This topic is dealt with in Chapter 7.

The dynamic selection of the most suitable alternative in dynamic binding problems has been addressed in Chapter 7, with the goal of continuously providing a given target reliability.

First, the choice of dynamically binding a service request to one of two available alternatives has been addressed by means of control-theoretical analysis and synthesis. An auto-tuning procedure has been devised to automatically select the most suitable controller configuration even at run-time. Subsequently, the solution has been generalized to the selection among n alternatives.

Three different implementations have been used to validate the approach, as well as a simulation campaign, showing that the proposed control-theoretical approach is a feasible decision mechanism for the problem at hand.

Future research directions are:

- The current auto-tuning system is not effective when the goal is not feasible. A more robust strategy is under investigation that provides a reasonable behavior even when a sudden change in the environment shift the system in a situation where the re-tuning of the controller is required but the goal is not feasible.
- More quality properties can be accommodated within this framework by providing convenient measures of their current satisfaction. A more general framework for quality-driven dynamic-binding is being defined to allow for a broader applicability of this contribution.

Bibliography

[1] "Spring: The open source application framework for java." [Online]. Available: http://www.springsource.org

[2] A. Agarwal, *Analysis of cache performance for operating systems and multiprogramming*, ser. Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 1989.

[3] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya, "Archeopterix: An extendable tool for architecture optimization of aadl models," in *Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES '09. ICSE Workshop on*, may 2009, pp. 61—71.

[4] H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner-Fischer, and S. Leue, "Safety analysis of an airbag system using probabilistic fmea and probabilistic counterexamples," in *Quantitative Evaluation of Systems, 2009. QEST '09. Sixth International Conference on the*, sept 2009, pp. 299—308.

[5] M. Alrifai and T. Risse, "Combining global optimization with local selection for efficient qos-aware service composition," in *Proceedings of the 18th international conference on World wide web*. New York, NY, USA: ACM, 2009, pp. 881—890.

[6] S. C. Althoen and R. McLaughlin, "Gauss-Jordan reduction: A brief history," *The American Mathematical Monthly*, vol. 94, no. 2, pp. 130—142, 1987.

[7] S. Andova, H. Hermanns, and J. Katoen, "Discrete-time rewards model-checked," *Formal Modeling and Analysis of Timed Systems*, pp. 88—104, 2004.

[8] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: a language and compiler for algorithmic choice," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 38—49.

[9] M. Araki and K. Yamamoto, "Multivariable multirate sampled-data systems: State-space description, transfer characteristics, and nyquist criterion," *IEEE Transactions on Automatic Control*, vol. 31, no. 2, pp. 145—154, feb 1986.

[10] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 369—384, June 2007.

[11] D. Ardagna and R. Mirandola, "Per-flow optimal service selection for web services based processes," *Journal of Systems and Software*, vol. 83, no. 8, pp. 1512—1523, 2010.

[12] K. Åström and T. Hägglund, *Advanced PID Control*. Research Triangle Park, NC: ISA - The Instrumentation, Systems, and Automation Society, 2005.

[13] A. Aziz, V. Singhal, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli, "It usually works: The temporal logic of stochastic systems," in *Computer Aided Verification*. Springer, 1995, pp. 155—165.

[14] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 198—209.

[15] C. Baier and J. Katoen, *Principles of model checking*. The MIT Press, 2008.

[16] C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan, "Symbolic model checking for probabilistic processes," in *Automata, Languages and*

Programming, ser. Lecture Notes in Computer Science, P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, Eds. Springer, 1997, vol. 1256, pp. 430—440.

[17] C. Baier, P. D'Argenio, and M. Groesser, "Partial order reduction for probabilistic branching time," *Electronic Notes in Theoretical Computer Science*, vol. 153, no. 2, pp. 97–116, 2006.

[18] C. Baier and M. Kwiatkowska, "Model checking for a probabilistic branching time logic with fairness," *Distributed Computing*, vol. 11, pp. 125—155, 1998.

[19] A. Barenghi, S. Crespi Reghizzi, D. Mandrioli, V. Ponte, M. Pradella, and E. Viviani, "Practical parallel parsing for large texts," in *Proceedings of SLE 2012*, accepted for publication.

[20] L. Baresi, E. Di Nitto, and C. Ghezzi, "Toward open-world software: Issue and challenges," *Computer*, vol. 39, no. 10, pp. 36—43, Oct. 2006.

[21] ——, "Toward open-world software: Issue and challenges," *Computer*, vol. 39, no. 10, pp. 36—43, Oct 2006.

[22] S. Becker, "Model transformations in non-functional analysis," in *Formal Methods for Model-Driven Engineering*, ser. Lecture Notes in Computer Science, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds. Springer, 2012, vol. 7320, pp. 263—289.

[23] S. Becker, H. Koziolek, and R. Reussner, "Model-based performance prediction with the palladio component model," in *Proceedings of the 6th international workshop on Software and performance*, ser. WOSP '07. New York, NY, USA: ACM, 2007, pp. 54—65.

[24] N. Ben Mabrouk, S. Beauche, E. Kuznetsova, N. Georgantas, and V. Issarny, "Qos-aware service composition in dynamic service oriented environments," in *Middleware*, ser. Lecture Notes in Computer Science. Springer, 2009, vol. 5896, pp. 123—142.

[25] M. Benidir, "On the root distribution of general polynomials with respect to the unit circle," *Signal Processing*, vol. 53, no. 1, pp. 75—82, 1996.

[26] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst, "Synoptic: studying logged behavior with inferred models," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 448—451.

[27] A. Bianco and L. de Alfaro, "Model checking of probabilistic and nondeterministic systems," in *Foundations of Software Technology and Theoretical Computer Science*, ser. Lecture Notes in Computer Science, P. Thiagarajan, Ed. Springer, 1995, vol. 1026, pp. 499—513.

[28] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 118—149, 2003.

[29] P. Billingsley, "Statistical methods in markov chains," *The Annals of Mathematical Statistics*, vol. 32, no. 1, pp. 12—40, 1961.

[30] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 318—329.

[31] G. Blair, N. Bencomo, and R. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22—27, oct 2009.

[32] A. Bojanczyk, "Complexity of solving linear systems in different models of computation," *SIAM Journal on Numerical Analysis*, vol. 21, no. 3, pp. 591—603, 1984.

[33] D. Bosnacki, S. Edelkamp, and D. Sulewski, "Efficient probabilistic model checking on general purpose graphics processors," in *Model Checking Software*, ser. Lecture Notes in Computer Science, C. Pasareanu, Ed. Springer, 2009, vol. 5578, pp. 32—49.

[34] R. Brockett, "Optimal control of observable continuous time markov chains," in *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*. IEEE, 2008, pp. 4269—4274.

[35] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science. Springer, 2009, vol. 5525, pp. 48—70.

[36] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Commun. ACM*, vol. 55, no. 9, pp. 69—77, Sep. 2012.

[37] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic qos management and optimization in service-based systems," *IEEE Transactions on Software Engineering*, vol. 37, pp. 387—409, 2011.

[38] R. Calinescu and M. Kwiatkowska, "Using quantitative analysis to implement autonomic it systems," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE Computer Society, 2009, pp. 100—110.

[39] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "An approach for qos-aware service composition based on genetic algorithms," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 1069—1075.

[40] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski, "Performance and environment monitoring for continuous program optimization," *IBM Journal of Research and Development*, vol. 50, no. 2.3, pp. 239—248, march 2006.

[41] R. C. Cheung, "A user-oriented software reliability model," *IEEE Trans. Softw. Eng.*, vol. 6, no. 2, pp. 118—125, 1980.

[42] W. Ching and M. Ng, *Markov Chains: Models, Algorithms and Applications*, ser. International Series in Operations Research & Management Science. Springer, 2005.

[43] J. Chinneck, *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods*, ser. International Series in Operations Research and Management Science. Springer, 2008.

[44] A. Ciancone, A. Filieri, M. Drago, R. Mirandola, and V. Grassi, "Klapersuite: An integrated model-driven environment for reliability and performance analysis of component-based systems," in *Objects, Models, Components, Patterns*, ser. Lecture Notes in Computer Science. Springer, 2011, vol. 6705, pp. 99—114.

[45] F. Ciesinski, C. Baier, M. Grosser, and J. Klein, "Reduction techniques for model checking markov decision processes," in *Quantitative Evaluation of Systems, 2008. QEST '08. Fifth International Conference on*, Sept 2008, pp. 45—54.

[46] E. Clarke, D. Garlan, B. Krogh, R. Simmons, and J. Wing, "Formal verification of autonomous systems nasa intelligent systems program," 2001.

[47] E. Clarke, "Model checking," in *Foundations of Software Technology and Theoretical Computer Science*, ser. Lecture Notes in Computer Science, S. Ramesh and G. Sivakumar, Eds. Springer, 1997, vol. 1346, pp. 54—56.

[48] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu, "Learning assumptions for compositional verification," in *Proceedings of TACAS 2003*, ser. LNCS, vol. 2619. Springer, 2003, pp. 331—346.

[49] C. Conway, K. Namjoshi, D. Dams, and S. Edwards, "Incremental algorithms for inter-procedural analysis of safety properties," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer, 2005, vol. 3576, pp. 387—400.

[50] V. Cortellessa and V. Grassi, "A modeling approach to analyze the impact of error propagation on reliability of component-based systems," *Component-Based Software Engineering*, pp. 140—156, 2007.

[51] C. Courcoubetis and M. Yannakakis, "Verifying temporal properties of finite-state probabilistic programs," in *Foundations of Computer Science, 1988., 29th Annual Symposium on*, oct 1988, pp. 338—345.

[52] ——, "The complexity of probabilistic verification," *J. ACM*, vol. 42, no. 4, pp. 857—907, Jul. 1995.

[53] T. Davis, *Direct methods for sparse linear systems*. Society for Industrial Mathematics, 2006, vol. 2.

[54] C. Daws, "Symbolic and parametric model checking of discrete-time markov chains," in *Theoretical Aspects of Computing - ICTAC 2004*, ser. Lecture Notes in Computer Science, Z. Liu and K. Araki, Eds. Springer, 2005, vol. 3407, pp. 280—294.

[55] E. De Rocquigny, N. Devictor, and S. Tarantola, *Uncertainty in Industrial Practice: A Guide to Quantitative Uncertainty Management*. Wiley, 2008.

[56] G. Di Caro and M. Dorigo, "Mobile agents for adaptive routing," in *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, vol. 7, jan 1998, pp. 74—83.

[57] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl, "A journey to highly dynamic, self-adaptive service-based applications," *Automated Software Engineering*, vol. 15, pp. 313—341, 2008.

[58] Y. Diao, J. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung, "Self-managing systems: a control theory foundation," in *Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the*, april 2005, pp. 441—448.

[59] N. M. V. Dijk and M. L. Puterman, "Perturbation theory for markov reward processes with applications to queueing systems," *Advances in Applied Probability*, vol. 20, no. 1, pp. 79—98, 1988.

[60] S. Distefano, A. Filieri, C. Ghezzi, and R. Mirandola, "A compositional method for reliability analysis of workflows affected by multiple failure modes," in *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*. ACM, 2011, pp. 149—158.

[61] R. Dorf and R. Bishop, *Modern control systems*. Prentice Hall, 2008.

[62] J. Doyle, B. Francis, and A. Tannenbaum, *Feedback control theory*. Basingstoke, UK: MacMillan, 1992.

[63] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 7, pp. 1165—1178, july 2008.

[64] J. Duato, "On the design of deadlock-free adaptive routing algorithms for multicomputers: Design methodologies," in *PARLE '91 Parallel Architectures and Languages Europe*, ser. Lecture Notes in Computer Science. Springer, 1991, vol. 505, pp. 390—405.

[65] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck, "From data center resource allocation to control theory and back," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, july 2010, pp. 410—417.

[66] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification," in *Proceedings of the second workshop on Formal methods in software practice*, ser. FMSP '98. New York, NY, USA: ACM, 1998, pp. 7—15.

[67] W. Farr, "Software reliability modeling survey," *Handbook of software reliability engineering*, pp. 71—117, 1996.

[68] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, "Reliability analysis of component-based systems with multiple failure modes," *Component-Based Software Engineering*, pp. 1—20, 2010.

[69] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio, *Reliability-Driven Dynamic Binding Implementation*, Dipartimento di Elettronica e Informazione - Politecnico di Milano, 2011. [Online]. Available: http://filieri.dei.polimi.it/publications/2012-seams

[70] A. Filieri and C. Ghezzi, "Further steps towards efficient run-time verification: Handling probabilistic cost models," in *Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012 Formal Methods in*, june 2012, pp. 2—8.

[71] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, "Reliability analysis of component-based systems with multiple failure modes," in *Component-Based Software Engineering*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6092, pp. 1–20.

[72] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 341—350.

[73] ——, "A formal approach to adaptive software: continuous assurance of non-functional requirements," *Formal Aspects of Computing*, vol. 24, pp. 163–186, 2012.

[74] R. W. Floyd, "Syntactic analysis and operator precedence," *J. ACM*, vol. 10, no. 3, pp. 316—333, Jul 1963.

[75] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems, 6th Edition*. Pearson, 2009.

[76] L. Fribourg and A. Étienne, "An inverse method for policy-iteration based algorithms," in *Proceedings International Workshop on Verification of Infinite-State Systems INFINITY*, 2009, pp. 44—61.

[77] K. Z. Gajos, M. Czerwinski, D. S. Tan, and D. S. Weld, "Exploring the design space for adaptive graphical user interfaces," in *Proceedings of the working conference on Advanced visual interfaces*, ser. AVI '06. New York, NY, USA: ACM, 2006, pp. 201—208.

[78] K. Gallivan, M. Heath, E. Ng, J. Ortega, B. Peyton, R. Plemmons, C. Romine, A. Sameh, and R. Voigt, *Parallel Algorithms for Matrix Computations*, ser. Miscellaneous Bks. Society for Industrial and Applied Mathematics, 1987.

[79] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Quality prediction of service compositions through probabilistic model checking," *Quality of Software Architectures. Models and Architectures*, pp. 119–134, 2008.

[80] F. Gantmakher, *The Theory of Matrices*, ser. Chelsea Publishing Series. AMS Chelsea, 2000.

[81] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of software engineering*. Prentice Hall, 2003.

[82] C. Ghezzi and A. Sharifloo, "Verifying non-functional properties of software product lines: Towards an efficient approach using parametric model checking," in *Software Product Line Conference (SPLC), 2011 15th International*, Aug 2011, pp. 170—174.

[83] C. Ghezzi and D. Mandrioli, "Incremental parsing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 1, pp. 58–70, Jan. 1979.

[84] S. Gokhale and K. Trivedi, "Reliability prediction and sensitivity analysis based on software architecture," in *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, 2002, pp. 64—75.

[85] S. S. Gokhale, "Architecture-based software reliability analysis: Overview and limitations," *Dependable and Secure Computing, IEEE Transactions on*, vol. 4, no. 1, pp. 32—40, march 2007.

[86] R. Goldblatt, *Logics of time and computation*. Center for the Study of Language and Information, 1995, vol. 60, no. 1.

[87] K. Goseva-Popstojanova, M. Hamill, and R. Perugupalli, "Large empirical case study of architecture-based software reliability," in *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, nov. 2005, pp. 43—52.

[88] K. Goseva-Popstojanova and K. S. Trivedi, “Architecture-based approach to reliability assessment of software systems,” *Performance Evaluation*, vol. 45, no. 2–3, pp. 179—204, 2001.

[89] A. Gounaris, N. Paton, A. Fernandes, and R. Sakellariou, “Adaptive query processing: A survey,” in *Advances in Databases*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2405, pp. 882—940.

[90] V. Grassi and R. Mirandola, “Derivation of markov models for effectiveness analysis of adaptable software architectures for mobile computing,” *Mobile Computing, IEEE Transactions on*, vol. 2, no. 2, pp. 114—131, april-june 2003.

[91] H. Gruber and J. Johannsen, “Optimal lower bounds on regular expression size using communication complexity,” in *Foundations of Software Science and Computational Structures*, ser. Lecture Notes in Computer Science, R. Amadio, Ed. Springer, 2008, vol. 4962, pp. 273—286.

[92] D. Grune and C. J. H. Jacobs, *Parsing Techniques - a practical guide*, 2nd ed. Springer, 2008.

[93] L. Grunske, “Specification patterns for probabilistic quality properties,” in *Software Engineering, 2008. ICSE ’08. ACM/IEEE 30th International Conference on*, may 2008, pp. 31—40.

[94] Y. Guowei, M. Dwyer, and G. Rothermel, “Regression model checking,” in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, Sept 2009, pp. 115—124.

[95] E. Hahn, H. Hermanns, and L. Zhang, “Probabilistic reachability for parametric markov models,” *Model Checking Software*, pp. 88—106, 2009.

[96] E. M. Hahn, T. Han, and L. Zhang, “Synthesis for pctl in parametric markov decision processes,” in *Proceedings of the 3rd International Symposium - NASA Formal Methods*, 2011, pp. 146—161.

[97] H. Hansson and B. Jonsson, “A logic for reasoning about time and reliability,” *Formal aspects of computing*, vol. 6, no. 5, pp. 512—535, 1994.

[98] B. Haverkort and K. Trivedi, “Specification techniques for markov reward models,” *Discrete Event Dynamic Systems*, vol. 3, pp. 219—247, 1993.

[99] B. Haverkort and A. Meeuwissen, “Sensitivity and uncertainty analysis of markov-reward models,” *Reliability, IEEE Transactions on*, vol. 44, no. 1, pp. 147—154, mar 1995.

[100] J. Hellerstein, S. Parekh, Y. Diao, and D. Tilbury, *Feedback control of computing systems*, ser. Wiley interscience publication. John Wiley & Sons, 2004.

[101] J. L. Hellerstein, V. Morrison, and E. Eilebrecht, “Applying control theory in the real world: experience with building a controller for the .net thread pool,” *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 3, pp. 38—42, jan 2010.

[102] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. Sanvido, “Extreme model checking,” in *Verification: Theory and Practice*, ser. Lecture Notes in Computer Science. Springer, 2004, vol. 2772, pp. 180—181.

[103] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet, “Approximate probabilistic model checking,” in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2004, pp. 307—329.

[104] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to automata theory, languages, and computation*. Addison-wesley, 2007.

[105] IBM Inc., “IBM autonomic computing website,” http://www.research.ibm.com/autonomic, 2009.

[106] R. Iman and J. Helton, “An investigation of uncertainty and sensitivity analysis techniques for computer models,” *Risk Analysis*, vol. 8, no. 1, pp. 71–90, 2006.

[107] A. Immonen and E. Niemel, “Survey of reliability and availability prediction methods from the viewpoint of software architecture,” *Software and Systems Modeling*, vol. 7, no. 1, pp. 49—65, 2008.

[108] Intel Inc., “Reliability, availability, and serviceability for the always-on enterprise,” www.intel.com/assets/pdf/whitepaper/ras.pdf, 2005.

[109] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, “Core fusion: accommodating software diversity in chip multiprocessors,” *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 186—197, 2007.

[110] M. Jaeger, G. Mühl, and S. Golze, “Qos-aware composition of web services: An evaluation of selection algorithms,” in *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, ser. Lecture Notes in Computer Science. Springer, 2005, vol. 3760, pp. 646—661.

[111] D. Jansen, J.-P. Katoen, M. Oldenkamp, M. Stoelinga, and I. Zapreev, “How fast and fat is your probabilistic model checker? an experimental performance comparison,” in *Hardware and Software: Verification and Testing*, ser. Lecture Notes in Computer Science, K. Yorav, Ed. Springer, 2008, vol. 4899, pp. 69—85.

[112] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, and D. Kopylenko, *Professional Java Development with the Spring Framework*. Birmingham, UK, UK: Wrox Press Ltd., 2005.

[113] C. B. Jones, “Tentative steps toward a development method for interfering programs,” *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 4, pp. 596—619, Oct 1983.

[114] W. Jones, J. Hudepohl, T. Khoshgoftaar, and E. Allen, “Application of a usage profile in software quality models,” in *Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on*, 1999, pp. 148 –157.

[115] C. Karamanolis, M. Karlsson, and X. Zhu, “Designing controllable computer systems,” in *Proceedings of the 10th conference on Hot Topics in Operating Systems*. Berkeley, CA, USA: USENIX Association, 2005, pp. 9—15.

[116] J.-P. Katoen, T. Kemna, I. Zapreev, and D. Jansen, “Bisimulation minimisation mostly speeds up probabilistic model checking,” in *Tools and Algorithms for the Construction and*

Analysis of Systems, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4424, pp. 87—101.

[117] M. Kihl, A. Robertsson, M. Andersson, and B. Wittenmark, “Control-theoretic analysis of admission control mechanisms for web server systems,” *World Wide Web*, vol. 11, pp. 93—116, 2008.

[118] D. Knuth and A. Yao, *Algorithms and Complexity: New Directions and Recent Results*. Academic Press, 1976, ch. The complexity of nonuniform random number generation.

[119] D. E. Knuth, “Semantics of context-free languages,” *Theory of Computing Systems*, vol. 2, pp. 127—145, 1968.

[120] M. M. Kokar, K. Baclawski, and Y. A. Eracar, “Control theory-based foundations of self-controlling software,” *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 37—45, May 1999.

[121] H. Koziolek, B. Schlich, and C. Bilich, “A large-scale industrial case study on architecture-based software reliability analysis,” in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, nov. 2010, pp. 279—288.

[122] H. Koziolek, B. Schlich, C. Bilich, R. Weiss, S. Becker, K. Krogmann, M. Trifu, R. Mirandola, and A. Koziolek, “An industrial case study on quality impact prediction for evolving service-oriented software,” in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE ’11. New York, NY, USA: ACM, 2011, pp. 776—785.

[123] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig, “K42: building a complete operating system,” *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 133—145, apr 2006.

[124] S. Krishnamurthi and K. Fisler, “Foundations of incremental aspect model-checking,” *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 2, Apr. 2007.

[125] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Processor power reduction via single-isa heterogeneous multi-core architectures," *Computer Architecture Letters*, vol. 2, no. 1, 2003.

[126] D. Kusic and N. Kandasamy, "Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems," *Cluster Computing*, vol. 10, pp. 395—408, 2007.

[127] M. Kwiatkowska, "Model checking for probability and time: from theory to practice," in *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, june 2003, pp. 351—360.

[128] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 2.0: a tool for probabilistic model checking," in *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, sept 2004, pp. 322—323.

[129] ——, "Advances and challenges of probabilistic model checking," in *Communication, Control, and Computing (Allerton), 2010 48th Annual Allerton Conference on*, oct 2010, pp. 1691—1698.

[130] M. Kwiatkowska, D. Parker, and H. Qu, "Incremental quantitative verification for markov decision processes," in *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, june 2011, pp. 359—370.

[131] M. Kwiatkowska, "Quantitative verification: models techniques and tools," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE '07. ACM, 2007, pp. 449—458.

[132] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic model checking," in *Formal Methods for Performance Evaluation*, ser. Lecture Notes in Computer Science, M. Bernardo and J. Hillston, Eds. Springer, 2007, vol. 4486, pp. 220—270.

[133] ——, "Using probabilistic model checking in systems biology," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 4, pp. 14—21, Mar. 2008.

[134] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu, "Assume-guarantee verification for probabilistic systems," in *Proc. of TACAS 2010*, ser. LNCS, vol. 6015. Springer, 2010, pp. 23–37.

[135] M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma, "On incremental quantitative verification for probabilistic systems," in *Proceedings of the High-order workshop on automated runtime verification and debugging*, Manchester, Royaume-Uni, 2012.

[136] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan, "Incremental state-space exploration for programs with dynamically allocated data," in *Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 291—300.

[137] A. Leva, C. Maffezzoni, and R. Scattolini, "Self-tuning PI-PID regulators for stable systems with varying delay," *Automatica*, vol. 30, no. 7, pp. 1171—1183, 1994.

[138] A. Leva, "Pid autotuning algorithm based on relay feedback," *Control Theory and Applications, IEE Proceedings D*, vol. 140, no. 5, pp. 328—338, Sept 1993.

[139] A. Leva, S. Negro, and A. V. Papadopoulos, "PI/PID autotuning with contextual model parametrisation," *Journal of Process Control*, vol. 20, no. 4, pp. 452—463, 2010.

[140] W. Levine, *The control handbook*. CRC Press, 2005.

[141] F. Lewis and V. Syrmos, *Optimal Control*, 2nd ed. John Wiley & Sons, 2004.

[142] Q. Liang, X. Wu, and H. Chuin Lau, "Optimizing service systems based on application-level qos," *IEEE Transactions on Services Computing*, vol. 2, no. 2, pp. 108—121, Apr 2009.

[143] B. Littlewood, "A reliability model for markov structured software," *SIGPLAN Not.*, vol. 10, no. 6, pp. 204—207, Apr. 1975.

[144] ——, "A reliability model for systems with markov structure," *Applied Statistics*, pp. 172–177, 1975.

[145] C. Lu, Y. Lu, T. Abdelzaher, J. Stankovic, and S. Son, "Feedback control architecture and design methodology for service delay guarantees in web servers," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 17, no. 9, pp. 1014—1027, sept 2006.

[146] M. Maggio, H. Hoffmann, M. Santambrogio, A. Agarwal, and A. Leva, "Controlling software applications via resource allocation within the heartbeats framework," in *Decision and Control (CDC), 2010 49th IEEE Conference on*, dec 2010, pp. 3736—3741.

[147] S. Malik and S. Arora, *Mathematical Analysis*. Wiley, 1992.

[148] A. Martens, D. Ardagna, H. Koziolek, R. Mirandola, and R. Reussner, "A hybrid approach for multi-attribute qos optimisation in component based software systems," *Research into Practice–Reality and Gaps*, pp. 84—101, 2010.

[149] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng, "Composing adaptive software," *Computer*, vol. 37, no. 7, pp. 56—64, July 2004.

[150] I. Meedeniya, B. Buhnova, A. Aleti, and L. Grunske, "Architecture-driven reliability and energy optimization for complex embedded systems," in *Research into Practice – Reality and Gaps*, ser. Lecture Notes in Computer Science, G. Heineman, J. Kofron, and F. Plasil, Eds. Springer, 2010, vol. 6093, pp. 52—67.

[151] I. Meedeniya and L. Grunske, "An efficient method for architecture-based reliability evaluation for evolving systems with changing parameters," in *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*, 2010, pp. 229—238.

[152] I. Meedeniya, I. Moser, A. Aleti, and L. Grunske, "Architecture-based reliability evaluation under uncertainty," in *Proceedings of the joint ACM SIGSOFT conference – QoSA*

and ACM SIGSOFT symposium, ser. QoSA-ISARCS '11. New York, NY, USA: ACM, 2011, pp. 85—94.

[153] M. Morari and E. Zafiriou, *Robust process control*. Upper Saddle River, NJ: Prentice Hall, 1989.

[154] H. Muller, H. Kienle, and U. Stege, "Autonomic computing now you see it, now you don't," in *Software Engineering*, ser. Lecture Notes in Computer Science. Springer, 2009, vol. 5413, pp. 32—54.

[155] P. Naur and B. Randell, "Software engineering report of a conference sponsored by the nato science committee garmisch germany 7th-11th october 1968," 1969.

[156] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta, "Using probabilistic model checking for dynamic power management," *Formal Aspects of Computing*, vol. 17, pp. 160–176, 2005.

[157] J. Norris, *Markov Chains*, ser. Cambridge Series on Statistical and Probabilistic Mathematics. Cambridge University Press, 1998, no. no. 2008.

[158] S. Oberthür, C. Böke, and B. Griese, "Dynamic online reconfiguration for customizable and self-optimizing operating systems," in *Proceedings of the 5th ACM international conference on Embedded software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 335—338.

[159] A. O'Dwyer, *Handbook of PI And PID Controller Tuning Rules*, 2nd ed. Imperial College Press, 2006.

[160] Oracle Corp., "Automatic Workload Repository (AWR) in Oracle Database 10g," http://www.oracle-base.com/articles/10g/AutomaticWorkloadRepository10g.php.

[161] D. Parker, "Implementation of symbolic model checking for probabilistic systems," Ph.D. dissertation, University of Birmingham, aug 2002.

[162] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053—1058, Dec. 1972.

[163] W. Pestman, *Mathematical Statistics*, ser. De Gruyter Textbook. De Gruyter, 2009.

[164] M. Pezzè and M. Young, *Software testing and analysis: process, principles, and techniques*. Wiley, 2008.

[165] H. Pham, *Software reliability*. Wiley Online Library, 1999.

[166] ——, “Software reliability and cost models: Perspectives, comparison, and practice,” *European Journal of Operational Research*, vol. 149, no. 3, pp. 475 – 489, 2003.

[167] T. B. Pinkerton, “Program behavior and control in virtual storage computer systems,” Ph.D. dissertation, Ann Arbor, MI, USA, 1968, aAI6813378.

[168] E. Polak, “An historical survey of computational methods in optimal control,” *SIAM review*, vol. 15, no. 2, pp. 553—584, 1973.

[169] Q-ImPrESS Consortium. (2011) Q-impress research project (eu fp7-215013). [Online]. Available: http://http://www.q-impress.eu

[170] A. Quarteroni, R. Sacco, and F. Saleri, *Numerical mathematics*. Springer Verlag, 2007, vol. 37.

[171] F. Rabhi and S. Gorlatch, *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[172] O. Raiha, “A survey on search-based software design,” *Computer Science Review*, vol. 4, no. 4, pp. 203—249, 2010.

[173] C. V. Ramamoorthy, “The analytic design of a dynamic look ahead and program segmenting system for multiprogrammed computers,” in *Proceedings of the 1966 21st national conference*. New York, NY, USA: ACM, 1966, pp. 229—239.

[174] S. C. Reghizzi and D. Mandrioli, “Operator precedence and the visibly pushdown property,” *Journal of Computer and System Sciences*, vol. 78, no. 6, pp. 1837—1867, 2012.

[175] S. Ross, *Stochastic Processes*. Wiley New York, 1996.

[176] Y. Saad, *Iterative methods for sparse linear systems*. Society for Industrial Mathematics, 2003.

[177] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 1—42, may 2009.

[178] Scilab Consortium, *Scilab: The free software for numerical computation*, Scilab Consortium, Digiteo, Paris, France, 2011. [Online]. Available: http://www.scilab.org

[179] O. Sokolsky and S. Smolka, "Incremental model checking in the modal mu-calculus," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer, 1994, vol. 818, pp. 351—363.

[180] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger, "Eon: a language and runtime system for perpetual systems," in *Proceedings of the 5th international conference on Embedded networked sensor systems*, ser. SenSys '07. New York, NY, USA: ACM, 2007, pp. 161—174.

[181] M. Springer, *The algebra of random variables*, ser. Probability and Statistics Series. Wiley, 1979.

[182] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS*, 2009, pp. 253—264.

[183] Q. Sun, G. Dai, and W. Pan, "LPV model and its application in web server performance control," in *CSSE*, vol. 3. Washington, DC, USA: IEEE Computer Society, 2008, pp. 486–489.

[184] J. Tan, *Fundamentals of Analog and Digital Signal Processing*. AuthorHouse, 2008.

[185] M. Tanelli, D. Ardagna, and M. Lovera, "LPV model identification for power management of web service systems," in *IEE MSC*. Boston, MA: IEEE Control Systems Society, 2008, pp. 1171–1176.

[186] H. Taylor and S. Karlin, *An introduction to stochastic modeling*. Academic Press (Boston), 1994.

[187] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, "A framework for adaptive algorithm selection in STAPL," in *ACM PPoPP*. New York, NY, USA: ACM, 2005, pp. 277—288.

[188] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. J. Wiley and Sons, 2009.

[189] M. Y. Vardi, "Automatic verification of probabilistic concurrent finite state programs," in *Foundations of Computer Science, 1985., 26th Annual Symposium on*, oct. 1985, pp. 327—338.

[190] S. Wadekar and S. Gokhale, "Exploring cost and reliability tradeoffs in architectural alternatives using a genetic algorithm," in *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, 1999, pp. 104—113.

[191] W.-L. Wang and M.-H. Chen, "Heterogeneous software reliability modeling," in *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, 2002, pp. 41—52.

[192] W.-L. Wang, D. Pan, and M.-H. Chen, "Architecture-based software reliability modeling," *Journal of Systems and Software*, vol. 79, no. 1, pp. 132—146, 2006.

[193] D. Woit, "Specifying component interactions for modular reliability estimation," in *Proc. 1st International Software Quality Week Europe (QWE'97)*, 1997.

[194] M. Woodside, D. Petriu, D. Petriu, H. Shen, T. Israr, and J. Merseguer, "Performance by unified model analysis (puma)," in *Proceedings of the 5th international workshop on Software and performance*. ACM, 2005, pp. 1—12.

[195] C. Xu and F. Lau, *Load Balancing in Parallel Computers: Theory and Practice*, ser. Kluwer international series in engineering and computer science. Springer, 1996.

[196] D. Yixin, W. W. Chai, J. Hellerstein, A. Storm, M. Surenda, S. Lightstone, S. Parekh, C. Garcia-Arellano, M. Carroll, C. Lee, and J. Colaco, "Comparative studies of load balancing with control and optimization techniques," in *Proceedings of the American Control Conference*, Jun 2005, pp. 1484—1490.

[197] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient algorithms for web services selection with end-to-end qos constraints," *ACM Transactions on the Web*, vol. 1, May 2007.

[198] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 311—327, May 2004.

[199] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin, "What does control theory bring to systems research?" *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 62—69, 2009.

List of Figures

List of Tables