Automatically Generated Runtime Checks for Design-Level Constraints

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Jochen Wuttke

under the supervision of
Mauro Pezzè

April 2010

Dissertation Committee

Antonio Carzaniga	Università della Svizzera Italiana, Switzerland
Holger Giese	Hasso Plattner Institute, Germany
Onn Shehory	IBM Haifa Research Labs, Israel
Michal Young	University of Oregon, OR, USA

Dissertation accepted on 12 April 2010

Mauro Pezzè
Research Advisor
Università della Svizzera Italiana, Switzerland

Michele Lanza
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Jochen Wuttke
Lugano, 12 April 2010

Abstract

In recent years, component- and service-orientation has gained importance as the new paradigm in software engineering, and it has introduced the challenge of dynamic component look-up and binding into the validation and verification process. This introduces a new class of inherently dynamic properties that have to be verified when the system is running in its target environment rather than at development time. Additionally, the separation of interface specification and the implementation combined with uses of Application Programming Interfaces (API) in contexts not envisioned by their developers often lead to subtle faults and consequent failures that are hard to diagnose. This problem is exacerbated when components or services are bound dynamically during deployment, and hence are not known during development and testing. One important implication of this is that it significantly limits the assurance of quality and reliability that testing at development time can give. Consequently, more quality assurance has to take place after the deployment of systems at runtime.

Most runtime verification techniques rely on monitoring state and behavior of systems and reasoning over the measured data to verify whether or not pre-defined properties are maintained. This dissertation is about runtime monitoring for structural design-level properties of component-based software systems. Such properties arise for example when a framework imposes requirements that must be met by all components, including components developed by third parties, connected within the framework. The working hypothesis is that there are useful design-level properties, violations of which lead to failures that exhibit enough commonalities to distinguish them from failures caused by other defects, and that these clusters of failures can be exploited to define reusable runtime monitoring mechanisms.

Based on this hypothesis, this dissertation's first contribution is a set of design-level properties that can be identified by clearly distinct classes of failures. Using these failure classes as guideline, the second contribution is defining templates for runtime monitors. These templates are similar in spirit to design patterns, even though their implementation is solving a monitoring, rather than a software design problem. The third contribution is a specification language to express the identified properties in design models, and a tool processing such models to automatically generate effective runtime monitors for the specified properties. The applicability of the identified properties and the effectiveness of the generated monitors is assessed in several case studies.

Acknowledgements

Many people contributed to this dissertation in different ways. My foremost thanks go to my advisor Mauro Pezzè, who not only supported and advised me during the research for this dissertation, but also taught me a lot about life as an academic. To the members of my dissertation committee, Antonio Carzaniga, Holger Giese, Onn Shehory and Michal Young: Thank you for the deep and insightful discussions and comments about my work. From Mehdi Jazayeri and Alexander Wolf I learned more than they might think, even if I did not always heed their advice. All of you gave me the guidance and support that made this thesis possible. Thank you for that.

Finally, to my friends here, Vaide and Christian, Anna and Alexander, Alessandra, Cyrus, Mircea, Philippe, Adina, Marco – you turned Lugano from a small town into a place worth living in. For this I am more grateful than I could possibly express.

Contents

Chapter 1

Introduction

Increasingly complex and dynamic software systems pose new challenges for validation and verification. The increasing complexity of systems makes it harder to assure software correctness through development time testing and analysis. The increasing dynamicity makes it impossible to test all feasible system configurations at development time. Both these challenges suggest to add runtime verification mechanisms to software to continuously monitor its correctness and health. This thesis proposes property templates, an approach facilitating the automatic generation of runtime monitors for high-level structural constraints on systems.

Validation and verification are integral parts of all software engineering processes. Pezzè and Young [2007] discuss at length what kind of validation and verification is required at which stage of the development process, and conclude that ideally every artifact produced be accompanied by appropriate tests or validation and verification activities that ensure the quality of the artifact. But even the best validation and verification process cannot guarantee to find all faults in a software system, because in general the problem of software correctness is undecidable. Even though correctness is in general undecidable, small programs or functions may still be provably correct. However, the more complex software becomes, the less likely it is that its correctness can be assured by validation and verification.

Developing ways to deal with the complexity of software has been a driving force behind software engineering ever since the term was coined. Component- and service-based software engineering is one of the latest examples of new paradigms that have been developed explicitly to deal with the increasing complexity of software. Unfortunately, every new development technique or paradigm introduces new kinds of faults that pose challenges that validation and verification techniques designed for other approaches do not handle well. Component-based software, for example, introduced the challenge of dynamic component look-up and binding into the validation and ver-

ification process. Common wisdom backed by scientific research is that these new validation and verification challenges should be tackled as early as possible in the development cycle, because the cost of fixing detected problems increases exponentially in the later stages of software development [Boehm, 1981]. However, dynamic component look-up and binding is one example of inherently dynamic problems that have to be verified when the system is running in its deployed environment rather than at development time. In addition to purely dynamic problems, the increasing complexity of software makes it more likely that faults are not discovered during testing and analysis at development time, no matter how much effort is expended in this stage. This suggests to increase efforts in runtime verification to increase the likelihood that faults are discovered at some point during system lifetime.

1.1 Motivation

The paradigm of component based software engineering, that is the idea to construct software from components with a given functionality without having to know the details of the components implementation, serves as a good motivating example for the new challenges introduced by increasing software complexity and the development paradigms invented to deal with them. Component-based development implies much stronger correctness requirements on the individual components, because they must be usable in many contexts without change. It also poses new challenges for validating the composed systems. Two examples for classes of such components are the APIs of web applications such as Google Maps[1], Flickr[2], or Facebook[3], and standardized frameworks such as Java Server Pages [JSR245], Java Server Faces [JSR127], or Spring[4]. These examples show two important aspects of new validation and verification challenges introduced by component based software engineering. Web APIs are available to literally millions of developers who may use these APIs in contexts not even envisioned by the API developers. These new and unforeseen uses may reveal subtle faults that are difficult to detect. Frameworks like JSP and JSF are often defined and standardized purely via their interfaces, leaving the concrete implementation unspecified. This is considered a strength of the component-based approach, because developers using such frameworks only have to know and consider the defined interfaces, and should be able to swap concrete implementations of the framework when necessary. However, in practice this separation between interface and implementation, combined with the complete hiding of the implementation from the component client, can introduce subtle problems that are very hard to pinpoint when their effects cross component boundaries. Some of the case studies in Chapter 7 show concrete examples

[1] `http://maps.google.com`
[2] `http://www.flickr.com`
[3] `http://www.facebook.com`
[4] `http://www.springsource.org`

of such problems.

These fundamental new challenges of component-based software engineering are exacerbated by the increasing trend to dynamic composition and service binding. Web services are the current, most popular example for this dynamic binding. The basic idea is to define the composition of service-based applications through syntactic interface requirements that are assumed to reflect the semantics of the service. Service composition engines then instantiate these service applications by looking up services matching the requirements in some kind of service registry, and connect the different services according to the specification. This means that the composition of concrete service implementations is not known until the application is initialized by the execution engine, and hence validation and verification at development time are restricted to the services available at development time. Augmenting systems with runtime checks that are independent of the bound components and can detect problems in any configuration is one way of alleviating the problems introduced by dynamic component bindings. Some of the case studies in Chapter 7 show how this can be achieved within a given component framework.

The three novel properties of use in unpredicted contexts, unknown implementations, and dynamic composition have important implications for the validation and verification of systems exhibiting at least one of these properties. The most fundamental implication, and the one that lies at the core of this thesis, is that all these properties reduce the assurance of correctness that verification at development time can give. Almost by definition, if a component developer does not foresee a certain usage scenario, he will not build a test case for it, hence verification of that scenario is omitted. Different component implementations may meet the interface specifications, but often certain aspects of possible implementations are not defined by the specification. Exception handling is one example for unspecified behavior that will show up in the case studies later, and these unspecified parts of the behavior lead to gaps in what developers can test for, unless they know which concrete implementations of components they use. The same is true for dynamically bound systems. In addition, dynamic binding, that is looking up services in a registry every time the application is started, means that reproducible testing is impossible, since there are no guarantees that the look-up will return the same service every time.

If verification at development time is impractical or impossible, an obvious solution is to integrate verification mechanisms into systems themselves, and thus constantly verify them while they are deployed. A lot of research has gone into runtime monitoring and verification of non-functional properties such as minimal performance levels, but work on systematic runtime constraint verification approaches that are closer to the goals of classic testing is also increasing. Chapter 2 gives a detailed overview of the developments in this area. In this dissertation, I particularly focus on developing runtime monitoring and verification techniques for design-level structural constraints on software systems. Violations of such properties represent system failures, and I also

study the connection of such failures to their underlying faults.

1.2 Terminology

This thesis is about detecting unwanted behaviors in running software. When talking about such unwanted behavior, it is often useful to distinguish clearly between the cause for bad behavior, how it is represented in the programs internal state, and how it becomes obvious at system boundaries. In the literature these concepts are usually termed *fault*, *error* and *failure*. However, their definitions vary a little in the literature. The two main works that provide precise definitions of these terms are the taxonomy and terminology paper by Avizienis et al. [2004], and the IEEE Glossary of Software Engineering Terminology [IEEE, 1990]. The following three definitions attempt to give a precise meaning to how I use these terms in this thesis.

Definition 1.1. A system or component *failure* is the event that occurs when the system behavior, represented by externally visible system state, deviates from the system specification.

Definition 1.2. An *error* is a system state that *may* lead to a failure.

Definition 1.3. A *fault* is the cause of an error.

The difference between *error* and *failure* is subtle, yet sometimes relevant to a technical discussion: a failure is visible in the external system state, for example through an incorrect return value or a system crash. An error, on the other hand, represents an incorrect system state that has not yet become visible externally, and may never do so. Avizienis et al. [2004] emphasize this difference by distinguishing between *internal* and *external* system or component state. Even though this distinction often helps thinking clearly about a problem, practically it is often difficult to determine the boundaries of components or systems, and hence to define what comprises the internal and what comprises the external state. For example, when using an object-oriented language to implement a system, one could consider a class to be a component, and therefore any incorrect state that becomes visible to the users of the class should be considered a failure. If, on the other hand, one considers the library containing that class as a component, and the class is used only internally, then an incorrect state of that class should be considered an error instead. Hence, when viewing a system at different levels of abstraction, errors may become failures and vice versa. For clarity, in this dissertation I use the terminology as defined above, which is taken from Avizienis et al. [2004], but without making a strong distinction between error and failure.

My research focuses on failures that occur at the level of component integration or higher. At this level it is often difficult and unnecessary to precisely determine the fault that is causing the failure. More often it is sufficient to determine the component or the operation that contains the fault, to avoid their use or wrap them with additional

checks and transformations to mask the fault. Hence, precise fault localization is out of the scope of this thesis. However, when discussing the studies in Chapter 4 the understanding that faults are the causes of failures is required.

1.3 Research Hypothesis and Contributions

Component- and service-based architectures aim to alleviate the problem of system complexity by abstraction. With these approaches, overall system functionality is constructed from small, distinct units, components, or services. This separation of system structure from system functionality introduces an additional level of abstraction, simplifying the *composition* of systems, but complicating system *validation* and *verification*. Integration testing as a sub-discipline of software testing addresses the problems arising from componentized software construction. However, the introduction of dynamic composition and runtime adaptability limits the possibilities of integration and system testing before deployment, and highlights the need for continuous verification even at runtime.

The additional abstraction layer introduced by components suggests that integration testing, at development time as well as at runtime, focus on system properties that are appropriately expressed at this level. In Chapter 2 I survey existing testing and monitoring techniques for various types of properties. These techniques, mostly stemming from academic research, usually focus on very narrow domains and problems. They work well for those narrow problems, but the problem of structural and functional system-level and design-level properties has not been addressed by any of them. Such properties arise for example when frameworks connecting components impose requirements that must be maintained by all components, and are a typical case where runtime verification can be a valuable addition to development time testing and analysis techniques.

It must be noted here, and it is discussed in detail in Chapter 2, that new levels of abstraction to make system design easier are usually *not* accompanied by matching techniques that also lift constraint specification and checking to this higher level of abstraction. Not doing this, new techniques ignore the opportunity to specify constraints regarding entities on lower abstraction levels, but with a “global”, system-wide perspective.

In this thesis I address the question how such properties can be specified at a high level of abstraction and can be monitored effectively and efficiently at a lower level of abstraction. Based on observations during software studies and my own development activity, I formed the following research hypothesis:

> Many system failures that are related to high-level properties of the system have similar causes and exhibit common symptoms. These commonalities define equivalence classes that can be exploited to define general runtime detection mechanisms for the failures in each class, and to drive tools that automatically generate and deploy these runtime detectors into software systems.

With this hypothesis I make two fundamental claims: (1) I postulate the existence of classes of failures that are equivalent with regard to their causes and symptoms, and (2) I claim that these classes are sufficiently homogeneous to allow automatic generation of runtime detectors for each class. In this thesis I am not interested in failure classes that can and should be detected at development time by unit tests and other testing activities at the component level, even though the examples in later chapters show that the techniques developed here can also be useful during the earlier stages of development. Only failures that violate more abstract, higher-level system properties are of interest here.

For illustration, consider constraints on components or classes that require global information to be checked. Typical examples would be heap invariants that make claims over all objects on the heap. Taking into account the concept of *scope* it becomes clear that such properties cannot be checked or specified within the scope of a single component or class, because the state information required to check such constraints is not well localized, but rather spread across different scopes, and violations of such properties can be attributed to any class or component being part of the constraint. To contrast this with non-global constraints, consider class invariants and method pre- and post-conditions. These are constraints that are well localised, working within a limited scope and their violations are closely tied to the class they are specified in.

The two claims I make with my hypothesis are closely related and their relationship is circular to some extent. One needs a homogeneous set of failures to define a general runtime checker for it, and the definition of a set being *sufficiently homogeneous* could be based on whether or not it is possible to define a general runtime checker that works for all elements in the set.

To provide evidence for the validity of my hypothesis, I first collected failure and bug reports for several medium sized software systems and clustered those with similar symptoms and causes, ignoring failures that clearly could be caught by unit tests. These clusters of failures form the foundation of the contributions presented in this thesis. These classes of failures provide the framework within which the development of generic detectors takes places, and they also form the foundation of the modeling language used to drive the tool support for the technique associated with failure classes. The technical solution presented in this work consists of a UML profile that

allows the expression of properties, reflecting the failure classes identified in the software studies, and a tool generating runtime checking code from annotated models. The main contributions of this thesis are:

- Identifying the specification of high-level properties that regard lower level entities as an open, so far untapped area of research, and defining a framework for such specifications.
- Facilitating the automatic generation of runtime checks for high level properties specified in this framework. This is achieved by:
  - Identification and classification of failure classes.
  - Definition and evaluation of general runtime checkers capable of detecting these failure classes.
- A catalog of constraints that exemplify how the framework can be used.
- Experimental validation of the approach through the development of a specification language and prototype tool. Both, language and tool, support developers in the use of the failure classes to automatically enhance their software systems with runtime checks.

There are two distinct aspects to these contributions. One is the *existence* claim that failure classes exist, and the other is that they can be used to generate *useful* runtime checks. Evaluating such different claims naturally is also different. In general, it is hard to provide conclusive evidence for an existence claim over a newly introduced concept. Concrete evidence can only give a partial image, and with this particular concept, individual judgment of what constitutes a failure class is an important factor. However, the results of the studies discussed in Chapter 4 may convince the reader that there is sufficient merit to this claim. Similar considerations apply when assessing the usefulness of new techniques. When can a technique considered to be useful, which metrics are important, which are not? Chapter 7 discusses this issue in detail and then discusses case studies that show concrete cases where the technique developed in this thesis applies.

1.4 Thesis Organization

The remainder of this thesis is structure as follows. Chapter 2 discussed the State of the Art in runtime monitoring, identifies dimensions along which monitoring approaches can be distinguished, and discusses modeling as foundation for any automated approach. Chapter 3 gives an overview of the approach developed for defining and using property templates. Chapters 4-7 are the chapters detailing the core contribution of

this thesis. Chapter 4 presents the studies conducted to identify failure classes, summarizes their results and defines precise semantics for the failure classes identified. Chapter 5 discusses how the semantics of the failure classes map to concrete constraint checks, and how they can be inserted into existing systems. Chapter 6 documents the architecture and implementation of the current tool prototype and outlines the decision considerations that led to the solution developed for this thesis. Chapter 7 presents a set of case studies that demonstrate the effectiveness of the runtime checks defined in Chapter 5, and also demonstrate that there are clear cases where the approach is applicable to solve real-world problems. Finally, Chapter 8 concludes this thesis with a summary of the presented work, and an outlook on possible future developments and applications of the approach. The appendices provide detailed documentation of the tool and present the data collected during the software studies.

Chapter 2

State of the Art

Runtime checking of system properties is a diverse field, but most techniques have two things in common: they deal with specific kinds of properties, and they focus on one given level of abstraction for monitoring. This chapter gives an overview of current runtime monitoring and constraint checking techniques, classifies the employed properties and monitoring techniques, and identifies the gap in the current state of the art into which this thesis fits: The specification and runtime verification of system-level structural properties via low-level detection mechanisms such as method- and class-invariants.

This thesis is about **Runtime Checking for functional and structural properties of software systems**. The contributions made here touch on both, the definition of *properties* and the development of runtime checking mechanisms. Runtime checking conceptually involves two aspects: monitoring the runtime state and behavior of a system, and checking if the monitored data indicates violations of pre-defined properties. These properties must be captured in specifications or models that are formal enough to derive runtime checks from them.

Properties can be classified across various dimensions. The most common dimension in software engineering is probably the distinction between functional and non-functional properties [Ghezzi et al., 2002; Sommerville, 2006]. Functional specifications generally describe *what* a program is intended to do. Program statements that violate these specifications are typically called *faults*, and program states reached when executing faults are called *errors* or *failures* depending on their visibility to other program components [Avizienis et al., 2004; IEEE, 1990]. Non-functional properties are often statements about overall system behavior, such as runtime performance, resource consumption, or global constraints on program state and behavior that cannot be attributed to a single unit of functionality. Section 2.1 reviews work on classifying properties, defining fault and failure taxonomies, draws parallels to the work presented in

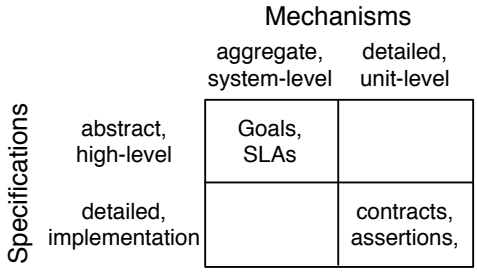


Figure 2.1. Comparison of the focus of existing monitoring techniques.

Chapter 4 and discusses important differences that distinguish the contribution of this thesis.

Runtime Checking of properties is subject to a classification directly related to the properties it is checking. Low-level functional properties may be checked by inline assertions, while high-level quality attributes such as average response time of service requests require more sophisticated infrastructure for measuring and evaluating results. This classification directly ties the abstraction level of the property to the level at which monitoring happens. Figure 2.1 illustrates this in a matrix that also shows the gaps left by this classification. Section 2.2 reviews work that led to this classification and discusses approaches towards filling the gaps.

Even though many practical approaches to runtime checking do not separate the concepts of *monitoring*, that is collecting relevant runtime data, and *checking* for property violations, more theoretical approaches usually draw a clear line between the (usually very hard) *oracle problem*, and the problem of collecting runtime data, which is often unjustly considered to be only a question of engineering. Section 2.2 reviews work on runtime monitoring, constraint checking, oracle generation and related topics that pertain to the contributions of Chapter 5.

Specifications, and in particular *models*, are the foundation of many automated techniques. *Model-based* techniques, a form of specification-based techniques, have received a large amount of attention in the last decade. Strong focus has been on developing sufficient techniques to support the entire software engineering process with a single set of consistent models. This idea of an integrated process with a well-defined set of models is important, because the types of models available generally have a strong impact on what automated techniques can theoretically and practically do with a given specification. Because of the importance of models, modeling itself is still an important research problem. Section 2.3 overviews *models* as a form of specification, and then reviews different modeling approaches, including approaches relying on automatic inference of models from development artifacts such as execution traces of the finished program. The discussion, also related to the needs of tool support,

relates strongly to Chapter 6, which discusses the framework and tool developed to validate the core hypothesis of this dissertation.

Tool support is always desirable when an engineering task involves tedious and repetitive activities. Transforming software specifications into programs is a task that falls into this category, and transforming functional and non-functional specifications into (runtime) checks capable of detecting violations of the specifications is an important, yet often neglected part of this task. Section 2.4 discusses existing tools for transforming specifications, identifies limitations and derives requirements for improving tool support.

2.1 Fault Taxonomies and Failure Hierarchies

There is a wide array of work in software testing, oracle and test-case generation, runtime monitoring and debugging that attempts to improve the effectiveness of individual techniques by restricting them to small, well-defined sub-problems such as specific kinds of specifications, failures or faults. Fault and failure taxonomies have been developed to divide the overall problems into smaller, more manageable pieces, and in the end, to place completed work in the larger context.

One widely used and cited fault taxonomy is that by Avizienis et al. [2004]. At the highest level of abstraction, they group faults into three partially overlapping sets: *development faults*, *physical faults* and *interaction faults*. These groups include faults that have been introduced during development either intentionally or by accident, faults that affect hardware, and faults that are due to factors outside the system, respectively. These groups are then further refined along different dimensions including whether or not faults are internal to the system, if they have been introduced by humans, etc. However, the taxonomy is not fine-grained enough to distinguish between different types of software faults within one of the categories. For example there are clearly distinct kinds of accidental, internal software faults, but the taxonomy cannot distinguish those. Such a distinction, on the other hand, seems indispensable to provide a concrete mapping between fault classes and failures (or errors) that are caused by these faults. To the best of my knowledge, no such work exists at the present time, and it is an open question if such a mapping is generally possible, or if any mapping must be language, platform or domain specific.

If detected failures are to be useful for diagnosing the underlying fault, there must be a mapping between faults or fault classes to failures or failure classes. The classification of failure domains provided by Avizienis et al. [2004] does not fulfill this requirement. It identifies 31 interesting classes of faults, but only five failure domains. Podgurski et al. [2003] present a technique to automatically classify failures, and they define a 1:1 mapping from faults to failures, distinguishing failure classes directly by the faults causing them such that all failures caused by the same fault are in one failure class. This choice of failure classes makes mapping from failures to faults easy, how-

ever the results of Podgurski et al. [2003] are based on statistical analysis and it is not clear whether the failures in the clusters can be described by commonalities that can be identified with elements of system design, that is if they violate the same *property*.

A method that ties fault analysis more strongly to detailed system design is *specification-based testing*. The fundamental idea in specification-based testing is to use formal specifications of acceptable system states and control flow. These specifications, practically always boolean specifications, can be used to derive fault conditions and test cases. The RELAY Model of Richardson and Thompson [1993] makes use of logical fault conditions in specifications derived by others in earlier work, for example Morell [1990], to derive conditions that guarantee to trigger specific fault types if they exist in the control flow path of generated test cases. Kuhn [1999] and Kapoor and Bowen [2007] refine the concept of fault-based testing and explore the limits of the error detection capability of techniques based on boolean specifications.

Both, the taxonomy of Avizienis et al. [2004] and the fault classes developed for specification-based testing are attempts to build taxonomies of common system faults and failure modes, and to establish how faults and failures are related. While Avizienis et al. [2004] take a high-level systems view, specification-based methods approach the same problem from the bottom up with specifications very close to the structure of the code implementing the system. While both are important contributions, they are also incomplete with regard to the needs of the work presented in this thesis. The classification of Avizienis et al. [2004] is too abstract and coarse grained to allow a useful mapping between failure domains and faults. On the other hand, the fault classes developed for the boolean specifications in specification-based testing regard code-level implementation faults such as switched operators and missing conditions, but are very hard to relate to higher level concepts like those identified by Avizienis et al. [2004]. The work presented here combines both of the above concepts. It establishes a clear connection between faults and failures, while at the same time associated specifications relate to system-level concepts rather than implementation details.

Besides work that aims to develop generally useful taxonomies, there is a large body of work that aims to solve the problem of error or failure detection for small, well-defined sets of errors and faults. The next section discusses the most relevant work in that area and provides a rough taxonomy of the properties being considered by the techniques discussed.

2.2 Runtime Monitoring, Constraint Checking, Oracles

The purpose of runtime monitoring is to collect data about a running system, and to analyze this data to detect conditions that require actions by system managers or automatic mechanisms to adapt the system. It is important to distinguish runtime monitoring from the means of failure detection that are usually applied in software testing,

even though technically most testing techniques also detect problems at runtime. The most important conceptual difference between testing and runtime monitoring is that in testing the system typically is executed in a controlled environment with carefully selected inputs, while runtime monitoring happens in production systems that are subjected to all inputs the system users can come up with [Pezzè and Young, 2007]. This difference implies several challenges in runtime monitoring that do not exist to this extent in traditional testing. First, while test oracles can be very specific to the test cases executed, oracles that determine problems at runtime must be capable of dealing with the entire range of inputs and outputs. Second, the execution of a production system cannot be controlled to fit a particular test case, rather the runtime checks must fit into normal execution. And third, performance is important in deployed systems, hence runtime monitors must not induce significant overhead. Besides these three major differences between runtime monitoring and testing, it is also helpful to take an overview of the existing literature on runtime monitoring to assess which kind of system information the community is mostly interested in. The overview in the following subsections will show that also the major goals of runtime monitoring and constraint checking differ from those of most testing techniques.

When discussing runtime monitoring or constraint checking it is helpful to consider three related, yet distinct aspects: first is the property that is to be monitored, for example system performance relevant to Service Level Agreement (SLA) guarantees, second are the techniques employed to monitor the system metrics and parameters relevant for a given property, and third are decision procedures or oracles determining whether or not the desired property is violated. Metrics are the connection between the first two aspects. Data pertaining to metrics is collected by runtime monitors, and then collated into aggregates that measure the degree to which goals are achieved or properties of the system are maintained. However, the low-level system data relevant to specific higher-level metrics and monitoring goals is often unknown, subject to intensive debate and an object of intensive research (for examples see [Chen et al., 2007; deGrandis and Valetto, 2009; Heo and Abdelzaher, 2009; Jiang et al., 2009]), and is of little relevance to the topic of this thesis. Hence, this section does not discuss detailed metrics beyond what is necessary for the discussion of system goals and properties, and their relation to monitoring techniques. The third topic, *oracles*, is more complex. When the monitored properties are related to system performance, then simple (numerical) thresholds marking the boundary between acceptable and unacceptable performance are sufficient. Properties relating to system structure or functionality require more complex decision procedures.

The remainder of this section reviews these issues in this order: first it discusses common topics in various fields of research that employ a form of runtime monitoring, then it discusses the metrics and measuring techniques used there, and concludes with a short discussion of the oracle problems involved in the different techniques.

2.2.1 Goals, Properties, and Problems

The motivation for and the problems addressed by runtime monitoring come from different areas. Traditional fault tolerance techniques have to detect failures of system components to mask failures or to recover from them. In the domain of traditional fault tolerance, *failure* typically means that a process crashed or can no longer respond for other reasons. Work on failure detectors in this domain is more of theoretical interest since a result by Fischer et al. [1985] shows that consensus (a fundamental concept in distributed systems) is not possible in the most general case when a single process fails. Even though there is a lot of work trying to weaken the preconditions for consensus as much as possible (e. g. [Chandra and Toueg, 1996]), the fundamental problem of actually detecting crashed processes is one that has received relatively little attention.

Fault tolerance techniques dealing with other types of failures, explicitly including failures due to faults in software, usually exploit forms of redundancy, such as N-version programming [Avizienis, 1985], similar forms of diversity for program data [Ammann and Knight, 1988], or deploying the same software in slightly different environments [Bruschi et al., 2007]. Rather than having monitors and oracles that contain some form of domain knowledge about expected results, such approaches vote on the outcome of computations. This idea of diversity, and hence redundancy in program text and data is conceptually clean and appealing. However, Brilliant et al. [1990] show that these techniques suffer from surprisingly high correlations between faults and errors in the different versions, and hence provide less added fault tolerance than expected.

Managing computing systems to optimize resource usage, minimize cost, or achieve other optimization goals is a second, large area that requires extensive information about the runtime behavior of systems. In recent years, the automatic management of large computing systems has been named *autonomic computing* by Horn [2001], and many other authors, for example Kephart and Chess [2003] and Lin et al. [2005], extend the idea and give more precise definitions of concepts, goals and challenges. Even though most of the recent work in systems management has been fit into the autonomic computing framework, collecting and analyzing runtime information is also useful for human-driven management and controlling tasks.

System management can have various goals and the monitoring objectives have to be tailored to these goals. Figure 2.2 shows a breakdown of the high-level objectives of system management most commonly found in the literature. Techniques focusing on *SLA guarantees* care about system behavior at the interface between service providers and clients. And even though much of the research literature discusses techniques that in principle would be applicable to a number of metrics, by far the most common guarantees offered by service providers are maximum response time [Breitgand et al., 2005; Chen et al., 2007; Raimondi et al., 2008], throughput [Bhat et al., 2006], or a combination of both expressed in combined utility functions [Walsh et al., 2004].

Systems management from the perspective of the owner or maintainer of comput-

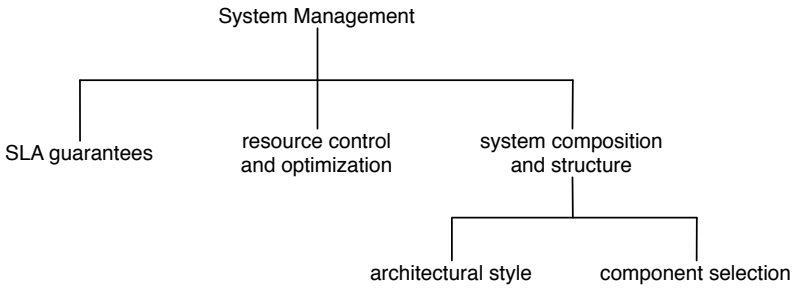


Figure 2.2. System Management Objectives

ing systems and supporting infrastructure is usually not only concerned with maintaining SLAs with clients, but also to optimize resource usage, reduce power consumption, and optimize the overall cost of running a computing system [Bahati et al., 2007; Bhat et al., 2006]. This concern, together with the protection of SLAs, constitutes one main focus of the literature on autonomic computing and systems management. One important factor that facilitates this work is that it depends on the measurement of non-functional metrics, most of which are readily available to probes that can be added to computing systems from the outside. This is in contrast to a large class of the fault tolerance techniques discussed above, which implicitly or explicitly attempt to mask the effects of functional faults.

The third aspect of system management that has received a lot of attention, in particular in the area of autonomic computing, is ensuring and maintaining architectural constraints on the system composition. As there are many views on what constitutes a system architecture, there are as many approaches to maintaining that particular architecture. For example, Garlan and Schmerl [2002] consider changing the number of clients and servers in a simple client-server architecture to constitute changes to the architecture, and these changes may be subject to constraints. Sykes et al. [2008] consider changes in the behavior of software, in their case changing the behavior of a robot from path-finding to specific task completion, as an adaptation best achieved at the architectural level. Even more so, Kramer and Magee [2007] consider the general problem of system self-management to be a problem of managing the systems architectures. Consequently, they propose to monitor the architectural composition of systems. However, even though with ArchJava [Aldrich et al., 2002], Rapide [Luckham et al., 1995], and Wright [Allen, 1997] there are a number of dedicated architecture description languages that often integrate directly with programming languages implementing the system, the system models described by these languages are typically used to statically analyze and generate parts of the system, but not to support runtime monitoring of the system to ensure that the constraints expressed in the models are maintained. One thing that is common to all approaches discussed above is that the constraints to be verified with the collected data are simple. Typically they are numer-

ical thresholds on metrics or the type of component connectors, and do not require sophisticated specifications and reasoning.

More complex, concrete approaches to monitoring structural constraints on system architectures often rely on simple models using the Unified Modeling Language [OMG, 2007] or similar semi-formal languages. Hein et al. [2007] utilize dynamic generation of UML models from actual system configurations to check constraints written in the Object Constraint Language [OCL, 2005] on these models. In a similar approach, Dzidek et al. [2005] use OCL annotations to UML class diagrams to generate runtime monitors for the OCL constraints. These approaches that not only monitor component compositions, but check constraints at a lower, programming language level, for example relations between objects, start blurring the boundary between non-functional and functional constraints. Heap invariants, as discussed by Jackson [2003], are a good example for those, and some of the UML and OCL based approaches discussed above move in the same direction. Moving closer to the programming language level usually also means a qualitative change in the kinds of metrics being expressed and checked. While higher-level monitoring focuses on non-functional metrics, typically geared towards performance and reliability assessments, lower-level checks usually focus on the functionality of the systems. Figure 2.1 is a graphical representation of this distinction. It also shows that there are open areas that have not yet been addressed well by research, namely how to relate high-level, goal-oriented specifications to concrete runtime monitors of low-level system behavior, and how low-level specifications, such as contracts, relate to system-level aggregate metrics or properties.

The techniques discussed above all have in common that the level of abstraction of the specifications is the same or nearly the same as the level of abstraction at which the constraints have to be checked. The conceptual contribution of this thesis is a framework for thinking about and specifying constraints at abstraction levels higher than the level at which the constraints must be checked. In principle, this enables and eases the specification of constraints that were difficult to specify before. For example, the possibility to connect high-level specifications with low-level checks allows the specification of global functional and structural constraints that previously could not be specified easily, because the specification mechanisms required the constraints to be localized, denying access to global information.

2.2.2 Constraints, Metrics and Assertions

This subsection follows the same order as the previous: it first reviews techniques related to monitoring data relevant for system management, and then it explores techniques that are closer to structural and functional properties. In both cases, the discussion focuses on the expressed constraints and how they translate to metrics or assertions to be checked.

As discussed above, system management is generally concerned with conflicting goals: on one hand system performance is to be maximized, while on the other hand

cost, for example in terms of energy consumption and required hardware, is to be minimized. The discussion here mostly ignores this conflict. However, it has one important implication when analyzing different approaches to monitoring for system management: Depending on the trade-off chosen by the authors of any given approach, different system metrics and evaluation methods may be given preference, leading to quite different results for what superficially seems to be the same problem.

It has become established wisdom that performance and utilization management is far from straightforward, and that manual translation of performance objectives into system-level thresholds is often not efficient when the number of monitored metrics is large [Chen et al., 2007]. Recently a lot of research has gone into the question which system metrics do actually predict system performance, and which do not. For example, Jiang et al. [2009] and Heo and Abdelzaher [2009] compute statistical correlation models of many low-level system metrics to determine which metrics correlate to management goals, and when changes to the correlations indicate problems in the system. Usually the best fit is achieved by using multiple metrics, which leads to difficulties when trying to define thresholds determining when a management action has to be taken. Walsh et al. [2004] show that computing utility functions, which could be considered an aggregate metric, can help in simplifying the problem of threshold setting, and deGrandis and Valetto [2009] propose a way to derive appropriate utility functions automatically.

Measuring low-level metrics and determining which of them correlate to selected performance goals is relevant from a system management point of view. Another line of inquiry takes the point of view of measuring and attempting to assure SLAs. Doing this allows one to measure relevant high-level metrics, for example response time or throughput in terms of transactions per second, and relate them to the explicitly stated thresholds in the SLA. Raimondi et al. [2008] present a very simple example of this, where they only monitor the response time of client requests. Apart from being fairly simple and not capturing more concepts useful for real-world SLAs, any such approach leaves the question open what to do if an SLA violation is detected. If no correlation model (as those discussed above) exists for the relation between the SLA-relevant metric and the system metrics that can be controlled by management, it remains an open question how to deal with the cause of SLA violations.

For all performance related techniques discussed above, the constraints are simple numerical thresholds that should not be exceeded, the monitored data are system metrics like response time, and the assertions to be checked are simple, since the constraints only require a direct numerical comparison between threshold and measured metric. Apart from the techniques above that focus on system performance measurements and management, there are two other classes of techniques worth mentioning here:

Techniques that monitor functional aspects typically occur in the context of fault tolerance schemes, and at least partially rely on techniques such as quorums and con-

sensus for determining problems, rather than precise oracles [Avizienis, 1985; Ammann and Knight, 1988]. Other techniques, for example RX by Qin et al. [2005] and micro-reboot by Candea et al. [2004], use exceptions as the implicit monitors and oracles. Interestingly, none of these techniques consider monitoring of functional properties in more detail than exceptions raised. There are even techniques that preemptively effect changes to the system to avoid exceptions completely, and using heuristics rather than monitoring, sidestep the problem of failure detection [Goldstein et al., 2007].

Techniques that Froihofer et al. [2007] summarize as *constraint validation approaches* generally are approaches that are based on specifications and derive some form of executable constraint check, for example assertions, from these specifications. Properties checked by these constraints include but are not limited to structural constraints, heap invariants and classical method pre- and post-conditions. Interesting constraint specification and checking frameworks not covered by Froihofer et al. [2007] include those by Kiviluoma et al. [2006], Gorbovitski et al. [2008], and Chen and Roşu [2007]. All these approaches have in common that they attempt a general framework for the specification of constraints and the generation of aspect-like runtime checks. They differ mostly in the level of abstraction they target and the type of constraints they are interested in. Chen and Roşu [2007] discuss very low-level functional properties, for example simple two-step object protocols, as the properties their approach can handle well. They express properties in different logics, for example Linear Temporal Logic (LTL), enclosed in comments in the code, and hence localized to specific classes and code. Gorbovitski et al. [2008] employ a very simple and very powerful specification mechanism to express constraints over sets of objects. These sets are typically "all instances of a given type" or similar, and the constraints in their examples are invariants over the objects' state. This approach is much more expressive than the approach presented here, but requires specifications that have detailed knowledge of the implementation and are more complex to write. Kiviluoma et al. [2006] present a UML-based approach to specify behavioral constraints at the architectural level. The examples shown are behavior protocols for component interactions in a system architecture. They employ techniques similar to the approach presented in Chapter 3: UML profiles to annotate system models, and automatic code generation from such annotated models to monitor the maintenance of the constraints. The main difference between all these approaches and the approach proposed here is that they require developers to specify detailed constraints rather than applying patterns that encapsulate the semantic detail of the constraints.

Wang and Shen [2007] describe an approach to detect violations of constraints intrinsic in UML class diagrams, that is they use the specification and check that the implemented system maintains the constraints represented by this specification, rather than formulating additional constraints. They instrument Java programs with calls to a monitoring infrastructure that uses assertion-like statements to check if the running system maintains invariants like association multiplicities. Dzidek et al. [2005] trans-

late OCL constraints added to UML class models into aspects that encode assertions matching the OCL constraints. Stirewalt and Rugaber [2005] present an approach to not only check, but to *enforce* OCL invariants at runtime. They use specified invariants to generate wrappers around the classes that the invariants refer to. These wrappers notify all classes involved in the constraints about state changes, and suitably update the related objects. Raimondi et al. [2008] combine the idea of specification patterns with the SLAng language to specify certain types of timing constraints that typically appear in the Service Level Agreements (SLA) of service-oriented applications. They translate typical patterns into timed automata and execute the automata together with the services to determine if events violate the SLA. These constraints are non-functional constraints that are closely tied to specific component functionality, and hence can be matched to some degree to structural specifications. Clark [2009] proposes a diagrammatic approach to express heap invariants, and suggests that diagrams can serve as the input for validation and verification. Clark's work indicates interesting research directions, but is in a preliminary stage, and is not yet supported by evidence. The already cited Jackson [2003] uses Alloy [Jackson, 2000] models to express constraints on object models, and discusses how these represent heap invariants.

These approaches show different ways to utilize models and constraints expressed over these models to generate runtime monitors. The approach by Stirewalt and Rugaber has a strong impact on the development and deployment of applications. Since their monitors deliberately produce side-effects, these effects have to be taken into account while developing systems. Hence the approach is not applicable to third-party components. The approach by Raimondi et al. shows how extensions to standard modeling languages can be used for application or task specific monitoring. The work makes strong assumptions about the checked properties, which effectively limits its applicability to timing constraints. The monitoring approaches closest to the one developed in this thesis are those by Wang and Shen and by Dzidek et al. Wang and Shen's approach monitors the maintenance of constraints implicit in UML models, while the approach here targets properties that constrain UML models, but cannot be expressed in UML models themselves. Further, Wang and Shen make strong assumptions about the models and their implementations. These assumptions make the approach hard to generalize and difficult to apply to third-party components. The approach by Dzidek et al. aims to monitor OCL constraints, while the design constraints developed in Chapter 5 cannot be expressed in OCL.

Heap invariants constrain the shape and structure of the object graph on the heap, which is inherently a runtime feature. Type systems, and in particular ownership types, represent programming language based static constructs that restrict object graphs. Clarke et al. [1998] originally developed ownership types to control aliasing in programs, and hence to make them accessible to static verification. In this earliest form, ownership types enforce a tree structure on the object graph, and hence facilitate traversing it for verification. Later work extended this idea to control object contain-

ment [Clarke et al., 2001; Clarke and Wrigstad, 2002; Dietl and Müller, 2005], and even to control shared data access in concurrent programs [Boyapati et al., 2002].

Reference and state uniqueness or immutability of objects are another type of heap invariant that has been addressed by work in the programming languages community, and like ownership types, usually hinges on the type system of programming languages [Boyland and Retert, 2005; Unkel and Lam, 2008; Zibin et al., 2007]. Like most type system based work, the goal is to ensure statically that undesirable heap configurations cannot occur. However, especially the rigid and inflexible requirement of ownership types that the object graph be a tree highlights the fact that often static approaches are too inflexible, and consequently led to efforts to weaken the guarantees of these approaches to gain more flexibility.

Protocol verification aims to ensure that communication protocols between components of a system are not violated. Formal work by Cengarle and Knapp [2005], followed by a practical verification approach by Knapp and Wuttke [2006] assures that behavior protocols are maintained at the specification level. Gan et al. [2007] build on this work and transform the finite state machines defined by Knapp and Wuttke into simple runtime monitors that check that actually occurring call sequences match the specification.

The approach of this thesis is more flexible than the static approaches discussed above, because it allows one to express the desired constraints only on objects where they are desired, while ownership types constrain all objects, and it does not require specifications that are formal and complete enough to allow static verification. Rather, individual specifications are translated into runtime checks that are capable of detecting violations of the desired, object or component-based invariant, without affecting the rest of the system. Additionally, lifting the specification of desired properties to the design level rather than the programming language level allows one to express properties that are not directly tied to entities on the programming language level, but rather conceptual entities, or to express whole program invariants.

2.2.3 Oracles

In general, the constraints discussed above are invariants. Checking these invariants at runtime requires automatic oracles, that is programs that can decide whether or not each constraint holds without human involvement. For the properties and techniques discussed above, these oracles can be fairly simple, like threshold checks for performance metrics, or rather complex when constraints have to assert on algorithm correctness.

Some specification approaches express the oracles directly. For example, contracts express pre- and post-conditions over program variables within the current scope, and hence these conditions can be evaluated directly. Other approaches, for example the heap invariants in the different UML and OCL based approaches, require some more work before oracles can be derived from the constraints. In the current state of the

art, this work has to be put in by developers, either explicitly by writing contracts, or implicitly by hard-coding translations in tools that process specifications and generate the monitoring code and wrappers. This is a non-trivial task, and one obvious generalization is that the more abstract a specification is, the more work is required to derive concrete oracles from it.

Hoare [1969] introduced the concept of axiomatic specifications for procedures, aimed at enabling formal proofs of program correctness. Meyer [1988] transformed this concept into *contracts*, which enable programs to check themselves for correctness. This idea of embedding pre- and post-conditions to enable basic correctness checks, even though being far from enabling automated correctness *proofs*, had significant impact over the years. Academic and industrial research both explored the usefulness and applicability of assertions, their limitations and benefits [Voas and Miller, 1994; Rosenblum, 1995; Ciupa et al., 2008; Polikarpova et al., 2009]. The theoretical appeal of axiomatic and checkable specifications is obvious, and assessments of the positive impact of consistently used assertions on software quality, for example as presented by Rosenblum [1995], led to continued academic interest and the development of powerful assertion languages and specification systems for most mainstream programming languages. Not only do most modern languages include facilities to conveniently express simple assertions within their code, but often more complex specification languages have been added to emulate the idea of contracts. Three prominent examples are, Eiffel which incorporates at the language level all concepts presented by Meyer [1988], the Java Modeling Language (JML), developed at the University of Central Florida by Leavens et al. [1999], and Spec#, developed at Microsoft Research by Barnett et al. [2004].

The limitations of assertions and contract-based specifications have also been explored. Voas and Miller [1994] provide evidence that finding the best assertions and the best location for assertions in the code is not trivial, and often not intuitive. Ciupa et al. [2008] extend this line of inquiry and in a large experiment using random testing come to the conclusion that even though contracts are successful in revealing faults, it is often the case that the fault lies in the contract, that is the contract does not express the intention of the annotated procedure, rather than the code implementing the procedure. Hence, these results emphasize the question of how to derive correct and useful assertions that express the programmer's, or better, the designer's intent.

2.3 Specification and Modeling Approaches

System design and properties that are "maintained" only in the mind of developers are of little use for documenting and communicating these concepts. *Documentation* implies that this information is written down, and formal *specifications*, or more re-

cently *models*[1], are one way of capturing a lot, but not all of the information about system design that should be documented. When one intends to use this information in tools that aid implementing, testing, validating or verifying the software, the form in which this information is written down has to be "machine readable". Code in a programming language or assembly is an example of a machine readable specification. However, the definition of *specification* in IEEE [1990, p. 69] requires a specification to be "a document that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or component." And code in common programming languages does not meet all these requirements. Program code (normally) is a complete and precise specification of program behavior, but it is not verifiable, because it does not identify requirements and system design. Hence, system code must be augmented with other forms of documentation to obtain a complete system specification.

All specification and modeling approaches I am aware of have similar limitations. The rest of this section first gives a synthetic overview of specification approaches classified according to their main purpose and features. This general discussion is followed by a more detailed analysis of persistent weaknesses in existing approaches, the identification of the particular problem addressed by the work in this thesis, and a comparison to the most closely related work. For the rest of this chapter, I will use the terms *specification* and *model* synonymously.

2.3.1 Synthesis

To give an overview of the main purposes and features of existing specification approaches, it helps to classify each approach and language along relevant dimensions that distinguish important features or delineate restrictions on the approaches. Important dimensions are:

Open vs. closed world

In logic, the open-world vs. closed-world assumption determine the truth value of statements that cannot be logically derived from known facts. In a closed world, all such statements are false, while in an open world, their value remains unknown. Similarly, a closed model describes the system exactly, there are no unknown elements, only what is in the model is allowed to be in the system. In contrast, an open model only describes parts of the system, and leaves the rest unspecified.

The fundamental distinction between viewing a specification with the open- or the closed-world assumption is that under the open-world assumption a specification represents a finite constraint on the possibly infinite space of possible systems that can be built. Effectively, there are still infinitely many systems that meet the specification,

[1] It might be interesting to note that in the IEEE glossary of software engineering terminology from 1990 the word *model* does not appear.

and those are allowed to have any behavior, structure or features not mentioned in the specification, as long as no part of the specification is violated. In contrast, under the closed-world assumption, a specification defines not only the explicit constraints on the infinite number of possible systems, but implicitly forbids all systems that exhibit more behavior than specified.

In many ways, this is an almost philosophical distinction, however, it has some important implications for the interpretation and verification of specifications. Closed-world specifications by definition are complete, that means all components, features, or behaviors of the system are explicitly specified. This makes closed-world specifications more amenable to verification, and by implication allows one to make stronger statements about the correctness of implementations, than with open-world specifications.

Defined vs. learned models

Models of systems can be either defined by developers, or can be learned or inferred from system artifacts such as code, modules, execution traces, or whatever else contains relevant information. This implies a fundamental difference between defined and learned models: Defined models are specifications in the sense of the definition, and as such express the intention of the developers, but not what actual system implementations do. Learned models, on the other hand, derive from attempts to discover the existing system structure and the behaviors possible within this structure, but it is impossible to recover intent, requirements, or design decisions from the final product.

As a consequence, learned models cannot be used for verification tasks directly, and instead their main uses of are (1) recovery of models of legacy systems [Kazman and Carrière, 1999; Wiggerts, 1997], (2) using learned models to drive test generation techniques, or to compare them for consistency with defined models to identify faults [Polikarpova et al., 2009; Schuler et al., 2009], (3) using changes in dynamically changing learned models as oracles for failure detection [Baah et al., 2006; Jiang et al., 2009; Mariani and Pezzè, 2005; Lorenzoli et al., 2008].

Dynamic vs. static models

Most classical specification techniques assume that systems are first specified and then built according to that specification. In the very early days of software engineering, changes to specifications after system completion were considered undesirable, and hence infrequent. Brooks [1975] disabused the community of that idea in his classic book *The Mythical Man Month*. At about the same time Parnas [1976] developed the idea of developing software families, and building software to facilitate changes in the functionality of members of the family [Parnas, 1979]. From this early work, software evolution and runtime adaptation have developed into major research fields.

Both fields embrace the idea that software is continuously changing, and thus, the software's specifications must also change continuously. While software evolution focuses on evolving systems as requirements change over relatively long periods of time, runtime adaption focuses on changes to running systems to optimize them for their current execution environment and meet current user needs. Changing systems in a controlled manner can benefit from models that describe the existing system precisely, and describe which changes are allowed. Having systems manage their own change automatically at runtime *requires* such models [Oreizy et al., 1998], and the ability to change the model of the system, that is the representation of the current system structure and state, together with the system itself.

This ability to change at runtime is the fundamental difference between dynamic and static models. As the other two dimensions along which specification approaches can be classified, this has implications on the possible structure, content and use of the specifications.

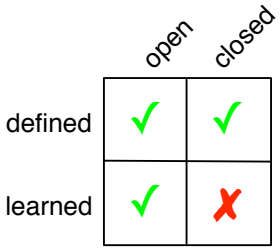
Discussion

Thanks to its generality, all specification approaches fit into this schema. Usually, approaches explicitly address problems of a particular scope, for example learning dynamic models from program executions, but that does not preclude their classification along the other dimensions. However, some classification combinations are impossible, or at least of doubtful utility. The matrices in Figures 2.3(a)-2.3(c) show which combinations are possible and sensible, which are impossible, and which are of dubious use.

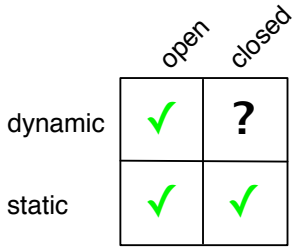
While defined models may be either open or closed, non-trivial learned models are always open, because there is no way to ensure that any learning method discovers the complete model. The same goes for the distinction between static and dynamic models. A model that never changes may be either open or closed, depending on how it is developed. Open dynamic models should be easy to realize, however, dynamic closed models require that every change to the model reflects all changes to the system in precise detail. Since most runtime models are relatively high-level abstractions of systems, it is doubtful whether having dynamic-closed models is practical. A similar problem occurs with defined-dynamic models. *Defined* models are created by developers independent of the system and represent a specification of it. Following this definition, a defined model can only be dynamic if change is a first-class entity in the model, and the model thus represents a language describing all possible system configurations reachable at runtime. Again, the practicality of this is doubtful.

2.3.2 Analysis

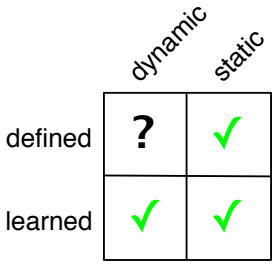
One thing that becomes directly obvious from the discussion of the open- and closed-world assumptions is that any kind of specification can (or maybe should) be under-



(a) Combinations for defining/learning models with open- and closed-world assumptions.



(b) Meaningful and doubtful combinations between dynamic/static and open/closed models.



(c) Meaningful and doubtful combinations of defined and learned static/dynamic models.

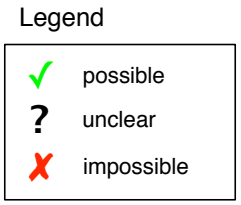


Figure 2.3. Matrices showing which combinations of open/closed, static/dynamic, and defined/learned modeling techniques are possible.

stood as a constraint on the infinite set of possible models, system implementations, or system states. Authors that explicitly acknowledge this notion include Jackson [2003], who considers object models, that is models showing allowed configurations of data structures at runtime, as invariants on the program heap. On the other hand side, modeling languages like UML, which, like any other specification language, expresses constraints on the model space, is augmented by OCL as an explicit constraint language to further constrain UML models. This distinction between modeling languages and constraint languages on models expressed in those modeling languages is not strictly necessary and is to some degree arbitrary. What becomes a first class entity in the modeling language and which concepts are factored out into a separate constraint language is essentially a matter of choice.

The second aspect that has to be considered to decide which concepts become first-class entities in a modeling language is which aspects of systems the language is intended to model. For example, software architecture, modularization, class hier-

archies, behavioral specifications and non-functional specifications all talk about different concepts, and no language can include all these concepts and remain usable. Hence most languages restrict their scope and thus the number of concepts they must include. For example, *Architecture Description Languages* (ADL) like ArchJava focus on the structural concepts of *components* and *connectors*, but have no means to specify behavior. UML sequence diagrams or Message Sequence Charts [ITU, 2004] define allowed communications between components, but have no concepts to express component structures and interconnections. Goal modeling languages like KAOS by Lamsweerde [2001] or Tropos by Giorgini et al. [2005] focus on eliciting and structuring non-functional requirements, but not on how to build systems meeting these requirements.

Property templates, as defined in this thesis, are constraints on *defined, static, open* component or class models. Since the types of constraints expressed by property templates are either heap invariants or method invariants, the models and constraint specifications do not need to be dynamic, since invariants cover all possible dynamically possible configurations. And the specifications utilized by the LuMiNous approach are defined and open for practical reasons. Generating runtime checks for specific constraints does not require complete or closed models, so open, partial models do fine, which is also more practical, because defining complete, closed models of real-world systems is an almost impossibly huge task. The work that is closely related to this thesis in terms of modeling and specification approaches is related not so much by the modeling language used, as by the goal and the types of models expressed. That is, approaches that express structural or heap invariants, rather than contracts.

2.4 Tool-driven Techniques

Identifying and specifying constraints is only the first step towards having runtime checks verifying these constraints. The second step is the creation of the actual monitors and oracles that verify the constraints. However, often the mapping between specifications and the necessary oracles and monitoring code is not trivial and requires considerable knowledge of the underlying platform and language. Automating this generation makes techniques reusable by average rather than expert programmers and removes this transformation as a source for errors. Codifying expert knowledge in transformations also eases reuse and sharing of this knowledge. Assuming a suitable tool infrastructure also allows a community of developers to contribute to the maintenance and expansion of available transformations.

Additionally, if specifications are language and platform independent, then tools provide the necessary separation between specification and implementation platform, allowing the same specification to be applied in different contexts. The potential benefits directly lead to some basic requirements constraint validation techniques and tools should fulfill:

- Specification languages should be general and useful to the entire development process rather than specific to particular tasks.
- Specification languages and tools should be easy to learn and use, and should not incur excessive additional work overhead.
- Tools should be customizable and extensible to allow constant evolution of transformations and properties.
- Where possible, specification languages and properties should be independent from implementation platforms and languages.

Most practical approaches have to make trade-offs between these requirements, and the approaches and tools discussed above all meet some of the requirements well, while not considering others. Most of the approaches discussed before make use of tools to generate oracles and to integrate them into the implementation of the system, or at least partially automate this process. However, most fall short on extensibility and expressiveness, because they rely on the expressiveness of standard UML. The approach and tool discussed in the remaining chapters of this thesis have been designed and built to meet these requirements better than competing approaches. However, some trade-offs were still necessary, and those will be discussed in the respective chapters.

2.5 Summary

Runtime verification touches on various fields. Properties to be verified have to capture the intentions of the developers, and should relate to common faults and failures. Existing taxonomies are domain specific and have relatively small overlap among them. Further, the types of properties and failure classifications available in the discussed literature is not sufficient for the work developed in this thesis. On the other hand, in their respective sub-areas other surveys are much more detailed than is necessary to discuss the contributions of this thesis [Delgado et al., 2004].

Most existing techniques that focus on runtime monitoring aim at systems management under performance and economic considerations. The metrics and specifications employed in this area are hardly useful for verifying functional and structural properties of systems. Work on runtime verification or enforcement of structural constraints is more strongly inspired by traditional testing techniques and relies on specifications of heap invariants in the form of class or object models, and contracts.

The recent focus on model-based techniques in testing and system development in practice implies that many new techniques are based on UML models or at least UML-like models. The advantage of this focus on UML is that in principle UML models are useful during the entire system development process. However, most approaches that add their own semantics or properties to the existing specification syntax do so in an

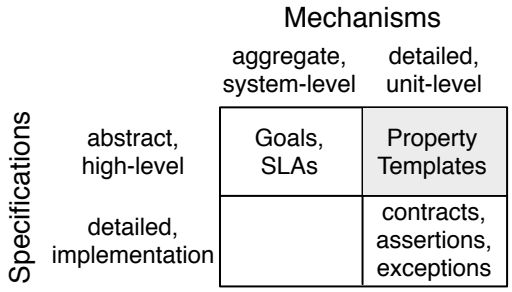


Figure 2.4. Relation of this thesis to other work.

ad-hoc manner; the idea of integrating individual techniques into a complete model-driven process is lost, and despite the use of UML as the foundation, techniques are effectively stand-alone.

The work presented in this thesis goes beyond this state of the art in several ways. It identifies classes of failures and faults, and by detailed analysis of their characteristics defines mapping between the fault and failure classes. That supplants the too abstract failure taxonomies as the foundation for specification and monitoring techniques. Further, the focus of the studies is on design-level properties that map to clearly identifiable error conditions. Properties of this kind, most prominently heap invariants and component protocols, have so far been mostly the domain of static verification techniques such as model checking or symbolic execution. Figure 2.4 shows where these properties fall in the matrix of property vs check abstraction level. The tool developed as part of the validation of this thesis also takes the goal of a complete model-driven process into account and keeps additions to standard modeling languages transparent, extensible and easy to use.

Chapter 3

The LuMiNous Approach

Runtime monitoring has been used for profiling, system management, support of fault tolerance and more. The previous chapter identified dimensions along which runtime monitoring approaches can be classified, discussed existing work within this framework and identified gaps in that work that remain to be filled. The comparison in Figure 2.4 identified techniques that specify properties at the abstraction level of system design, but has to check them at the statement or method level rather than the module level in the implementation as one such gap. In this chapter I propose a systematic approach, called LuMiNous, that fits into this gap. The approach exploits the results postulated by the hypothesis that violations of many design-level properties map to well-defined classes of failures, and that these classes of failures can be exploited to create runtime monitors. The approach outlined here also pays particular attention to make the results usable in practice. Chapters 4 and 5 then show that this hypothesis holds, and hence confirm the foundation for the approach.

This chapter discusses the overall approach that underlies the work in the following chapters. The approach hinges on two key principles: simplicity of specification, and a maximum of automation. It achieves simplicity of specification by defining a method for developers to analyze system requirements and to create and annotate system models. For simplicity, an easy to understand, straightforward language is preferable to a complex formal notation as long as it is capable of expressing the desired properties. Hence, the annotations introducing the properties are simple tags with few parameters, and they are easy to place in almost any kind of specification. The drive for automation is motivated by two orthogonal factors: first, the complexity of runtime monitors for design-level properties, and second, to pave the way for utilizing the results of this thesis in software development.

Design-level properties by definition affect substantial parts of software systems,

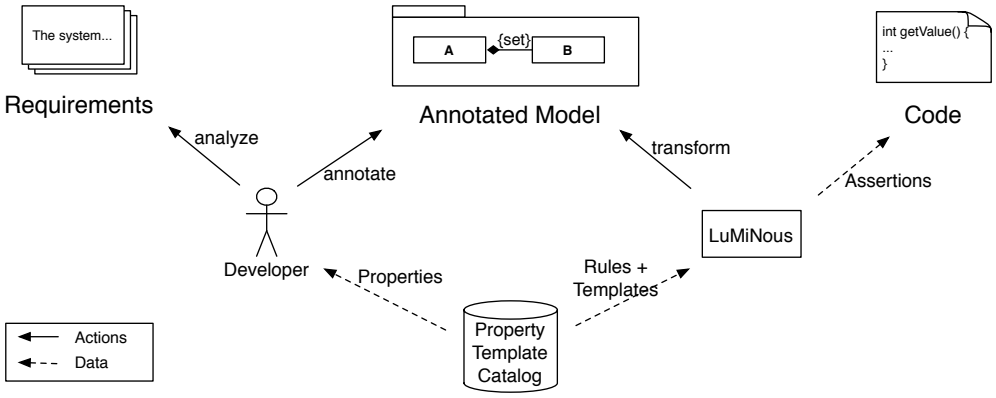


Figure 3.1. Overview of the LuMiNous approach

which suggests that runtime monitors checking for such properties will have to monitor many parts of the system rather than being well localized. The studies leading to the definition of failure classes in Chapter 4 and the case studies in Chapter 7 confirm that assertions for high-level properties are often distributed across substantial parts of the system. Creating and adding *all* assertions in the right places is therefore tedious and error-prone.

Utilizing the results of this thesis means enabling developers to create high-quality runtime monitors with effort that is commensurate with the added value of the monitors. This directly motivates automation, since automation reduces the effort developers have to put into the task of creating monitors, and likely increases the quality of the resulting monitors. Automation of course requires a tool to process the annotated specifications, and to generate runtime monitors from them.

The central element in this approach, and the foundation for tool support, is the property template catalog. *Property templates* are the major research contribution of this thesis that facilitates the automatic generation of runtime monitors. Each property template represents a design-level property, for example global state invariants or simple temporal properties, and code templates for runtime monitors capable of detecting violations of that property. Hence, implicit in the assertions of the runtime monitor, property templates also contain well defined classes of failures that signal property violations. Developers use this catalog to identify properties in their requirements, a model transformation tool uses the catalog of properties and matching templates to generate runtime monitors and to automatically insert them into the target system.

Figure 3.1 illustrates the components and activities of the approach. The approach focuses on two aspects: (1) Deriving and specifying system-level constraints, and (2) automatically translating constraints into runtime monitors, which are assertions in the prototype implementation of the tool in Chapter 6. The central building block is the repository of formalized failures classes, called *Property Templates*. Developers

systematically analyze the system requirements to identify properties in their system that match those in the catalog, which is described in detail in Chapter 4, and create and annotate a system model with these properties. This annotated model is the input for the model transformation tool that generates the runtime monitors. The generated code derives from the templates documented in Chapter 5. These templates consist of monitoring infrastructure that gathers the necessary data, and oracles that determine when a property violation occurred.

The first step, requirements analysis, is a task for developers, because it is difficult to automate. It requires understanding the semantics of requirements specifications, which are usually provided in natural language or informal notations that do not lend themselves well to automatic analysis. Often, decisions whether or not property templates apply to a given situation also have to consider the developers intentions of how to build the system. From a practical perspective, intuitive and yet well-defined semantics for property templates and simple annotation languages facilitate this step for developers. This is why the annotation language defined in Chapter 6 is not a fully fledged assertion language like JML or Spec#. Rather, it is a set of concrete annotations with well-defined semantics that readily translate into effective code-level assertions. This has the advantage that model annotations are very simple and can easily be placed. On the other hand, this limits the expressiveness of annotations to exactly the predefined set in the catalog.

The second step, monitor generation, is fully automated. The simplicity of the model annotations paired with the inherent complexity of bridging the semantic gap between system- and design-level models on one side, and the implementation on the other side, means that the automatic transformation must address this complexity. Fortunately, this same simplicity suggests a solution. It is feasible to define re-usable templates for runtime monitors matching the limited number of well-defined annotations. Using such templates, the annotations are translated into the necessary sets of assertions, and inserted in all relevant locations in the code. Like in every model-based technique, for the automatic assertion generation to work, there must be a semantic link between the annotated model and the system implementation. This also means that the model and the code must evolve together when the system changes. Keeping models and code synchronized is a well-researched problem, but still provides significant challenges. The tool discussed in Chapter 6 alleviates the problem of keeping models and system implementation synchronized by requiring only partial models of the system. The model has to contain only the entities relevant to each annotation, and assertion generation can proceed. This does not remove the need of keeping model and code synchronized, but it reduces the work to synchronizing a much smaller model to code changes.

To realize the complete approach, research as well as practical engineering challenges must be tackled. First, property templates must be identified, formalized, and brought into a format usable by software developers to create models as well as by

model transformation tools to generate code. Property templates are identified by empirical studies of software requirements and typical failures occurring in systems. This is mostly a research challenge addressing the feasibility of the approach, but the empirical research can be supported by feedback from practitioners. Second, the methods associated with the use of the developed property templates, that is the annotation language and transformation tool, must be easily usable, and the code must not only be effective in detecting property violations, but also efficient enough to be useful in real projects. This is by and large an engineering challenge to utilize existing technologies and build easily usable additional tools where needed. Empirical evaluation of the runtime performance of the monitors can lead to improvements in performance or trade-offs between precision and performance.

The technical chapters in the remainder of the thesis discuss the solutions found for these challenges. Chapter 4 discusses the research leading to the identification of property templates, and a formalization of their semantics, ultimately leading to catalog of reusable properties for developers to draw from during requirements analysis. Chapter 5 shows how the formal semantics are translated into effective and efficient code templates. The set of templates shown in this chapter is a concrete realization of the automatic translation step for Java applications, utilizing AspectJ as a tool. The complexity of mapping from the design level to the code level is hidden in the templates, since AspectJ performs the matching of monitoring code to locations where it has to be executed. Finally, Chapter 6 gives a short account of the design and implementation of the transformation tool used for the experiments discussed in Chapter 7. The overall practicality of the approach is also assessed in Chapter 7.

Chapter 4

Classes of Runtime Failures

The approach outlined in the previous chapter relies on property templates. Conceptually, property templates are software patterns that combine common system- and design-level properties with typical error patterns. This chapter discusses research that derives these patterns from studies of existing software systems. This research studies both, software requirements specifications and failure reports logged in issue tracking systems to establish links between properties in the requirements, and the failures of systems. The results of these studies confirm the hypothesis about the existence of clusters of failures associated with design-level properties. This chapter first motivates the research in more detail, then discusses the research methodology used to identify candidate properties, and then defines the formal semantics of these properties.

To be useful in the approach discussed previously, property templates must relate at least two factors: error patterns, that are patterns in the system state or behavior that signal that something is wrong, and more abstract, intuitive properties that are being violated when such patterns occur. For example, components or frameworks may require some initialization before they can be used, and an error pattern that identifies a violation of the requirement might be that a service is requested from a component before this initialization happened. Ideally, a third factor is related to the first two. This factor is that error patterns are due to faults of a similar type. It is part of the research question in this chapter to determine if there is such a connection between error patterns and faults. But if there is such a connection, the detection abilities of runtime monitors can be strengthened by also considering these fault patterns.

The studies in this chapter focus on both aspects, identifying recurring abstract semantics in the requirements specifications, like the example of *initialization* above, and identifying recurring error and failure patterns in the issue reports. Candidates for property templates are those patterns where there is sufficiently strong correlation

between error patterns and meaning. Precisely defining when parts of specifications constitute a pattern, when reported failures and errors constitute a pattern, and when the two are sufficiently correlated is difficult, because these definitions depend on human judgment. Most high-level requirements specifications are written in informal or semi-formal natural language, and with the current state of the art in information retrieval and natural language processing it is impossible to automatically analyze such specifications to extract their precise meaning. Similarly, determining when identified errors and failures constitute a pattern also depends on human judgment. There are approaches to automatic anomaly detection that may be capable of detecting some patterns, like component misuse [Mariani and Pezzè, 2005] or deviations from "normal" behavior [Baah et al., 2006]. However, they are limited by the assumption that "normal" behavior can be derived from the system under analysis. A lot of information about failures and faults is stored, again in natural language, in issue tracking systems, and this information is not accessible to automated analysis, even though some inroads have been made [Zimmermann et al., 2008]. Hence, the information that allows clustering and separating specifications and problems must be processed by humans, and thus decisions are always influenced by human judgment, and a more precise formal definition of when something becomes a pattern is of little help. In Section 4.1, recurring elements in specifications or similarities in failure reports are considered a pattern when they occur around five times and in several applications.

Dwyer et al. [1999] present research that is based on a similar idea. They collect patterns in temporal logic specifications for later reuse, and Cobleigh et al. [2006] build tool support for developers based an these results. However, their work differs from the work here in significant points. First, in temporal logic specifications, many general properties, for example liveness and safety, are already known. Dwyer et al. [1999] do not attempt to identify more such general properties, but simply use well known ones and then classify identified specification patterns as those and refinements of them. Second, temporal logic specifications have a completely different scope. They generally are used for automated verification of system behavior. Due to the abstractions necessary, this currently limits their applicability to component communication and coordination, but state related properties are very often still not feasible to verify. This makes automated verification via temporal logic specifications complementary to the runtime monitoring of state-based properties defined here.

On the other hand, Chapter 2 shows that much of the work on runtime monitoring focuses on low-level performance metrics or contract-like pre- and post-conditions for methods. It also identified a gap in the current state of the art. The hypothesis of this dissertation is that it is possible to close this gap by raising the abstraction level of state-based constraints to that of system architecture and design. Well defined methods working at this abstraction level give developers new tools to improve the quality and reliability of systems. The research presented in this chapter is the first step towards achieving that goal.

4.1 Research Methodology

The desired outcome of the research described in this chapter are clusters of software errors and failures that are grouped by their similarity in terms of their erroneous behavior and their relation to more general properties of the system. Hence, the research must first identify candidates for such general properties, then collect and cluster reported software errors and failures, and then see if the clusters correlate with the properties. A later step, required to facilitate automatic error and failure detection, is to define templates for runtime checks for each cluster, so that tools can generate the checks based on knowledge of where a property associated with the cluster should hold. This last step is discussed in detail in Chapter 5.

Since there are two clearly separable aspects that define error classes, I have designed the study in two steps. The first step is the analysis of requirements, such as end-user documentation or API specifications, to identify recurrent properties that are often used. For example, API specifications often refer to patterns for component initialization and mutability. More complex examples that can be found in end user documentation are requirements referring to domain specific input languages, like special regular expressions for searches. The second step consists of understanding and clustering problem reports for the applications and software systems I studied in the first step. Whenever clusters can be mapped to one or more property, I established an important link between the high-level requirements and the code implementing those requirements.

To have a consistent set of inputs for our studies we selected applications where specifications, code, and issue reports are available. For practical reasons I limit the scope of the study to applications developed in Java. The projects hosted by the Apache Foundation[1] provide a rich source for many types of applications from large servers like Tomcat, through various frameworks, to highly optimized libraries like Lucene. All of these projects have reached a level of maturity and quality that they are used in many successful commercial application scenarios.

To the best of my knowledge there is no mining tool that is able to process natural language documents or bug databases and extract sets of reports based on semantic criteria of the text content. Podgurski et al. [2003] present an automated AI based approach to automatically classify execution traces reported as failing, and in a companion report Francis et al. [2004] analyze in detail how precise the derived classifications are. However, using only execution traces does not capture the semantics of what is going wrong, and the precision reported by Francis et al. [2004] is not high enough to base the second part of the study on this approach. After a study of the existing work in repository mining, of how bugs are reported, and how symptoms and fixes are discussed, I came to the conclusion that a manual analysis of the reports was the only way to obtain reliable information about the semantic content of bug reports. Unfor-

[1] `http://www.apache.org`

tunately, the large amount of time required for manual analysis limits the number and size of studies the could be done within the scope of this thesis.

4.1.1 Requirements Analysis

The goal of the requirements analysis during the studies was twofold: (1) to determine recurring patterns that imply constraints on possible implementations in the specifications, and (2) to determine in how many cases identified patterns or constraints directly relate to clusters of problems identified during the fault analysis in the second step.

It turns out that complete requirements specifications are hard to come by for open-source projects. However, in all cases API specifications and some end-user documentation are available. I analyzed these specifications, which reflect black-box requirements and high-level design of the system.

Spotting patterns (not *design patterns*, but patterns defining less well-defined relationships between parts of the system) and recurring constraints in API specifications and end-user documentation is difficult and relies on the judgment of the person doing the analysis. For example, the Cocoon API specification[2] contains the following phrases spread across several classes and interfaces: "A simple immutable and serializable implementation of Location", "Lock this component info object at the end of processor building to prevent any further changes", "It is important that the cache key has the following attributes: [...] It must be Immutable", "Close this instance, or in other words declare that no other sources will be added ...". Intuitively, these phrases have in common that they specify that objects must not change their state. However, there are also differences. Some of the specifications above literally state that objects must be immutable, while others provide methods that allow users of an interface to declare that they no longer intend to change the state of the object. It is also not precisely defined what *immutable* means, so implementors are left to decide that on their own. Whether or not these specifications constitute the definition of the same property, that is that object state must not change, requires judgment involving the decision whether the commonalities overrule the differences and whether declarations of the interface provider and interface users can be treated as equivalent. This judgment can and should take a deeper analysis and understanding of the context in which these specifications occur into account. In an extreme case this may even lead to the decision that syntactically equivalent specifications might be treated as not being semantically equivalent.

The examples above led to the definition of the `immutable` property, and their differences led to the flexibility introduced in the semantics shown in Table 4.5 later in this chapter. This example illustrates why it is difficult to describe the process and the decisions that led to the identification of each property discussed later in a way

[2] `http://cocoon.apache.org/2.2/apidocs/index.html`

that makes it easily reproducible. Appendix B describes the study setup in more detail. For each study, the data there also lists which specifications (of which classes and interfaces) contributed to the definition of the properties listed in Table 4.1.

Despite the difficulty to describe the decision process in more detail, there are some general conclusions that can be drawn from the studies. With respect to patterns in specifications, well designed and documented API specifications yield results more easily, and as a consequence, it may be easier to reproduce the results of such studies. For example, the Java *Servlet* and *Java Server Pages* API documentation directly specify many instances of the `initialized` and `unique` properties. On the other hand, the API documentation for Lucene is very sparse and gives barely any information about constraints on the use of the library. In most cases end-user documentation was less yielding than API specifications and required more in-depth study to obtain useful results.

4.1.2 Failure Analysis

For all applications in the studies, issue repositories collect reports, colloquially called *bug reports*, of system failures and other problems. Those are the source of the failures that enter the classification reported here.

Bug reports for projects hosted by Apache are maintained either using Bugzilla or JIRA. In both cases an issue report refers to a specific release version of the software where the issue was first noticed and each issue has a status. Because all the software systems I analyzed are still under development, the issue databases are constantly changing. To have a stable set of issues to address throughout the time of the study I chose to study issues reported for older versions of each software. This not only yields a more stable set of reports to analyze, but also means that most of the issues have been resolved by the developers, easing the task of determining bug cause and fixes. I used the release version to select only small subsets of issues clearly associated with a particular version of the code. The issue status or resolution indicates how the developers consider the problem. In all cases I discarded issues that were marked as *invalid*, *non reproducible*, or as issues that *won't be fixed*. The rationale for discarding these issues is that if the developers do not consider something a bug, then neither should I.

After filtering those non-issues out of the result sets returned by queries to the issue databases, I studied every remaining issue in detail. Questions to be answered about each issue before they could be assigned to a cluster, create a new cluster, or be discarded were:

- What are the failure symptoms? For example, does the failure crash the system, return an illegal value, or fails to return an expected result?

- What kind of fault causes the failures? For example, is it a simple null pointer

Property	**Cocoon**		**Lucene**		**Tomcat**	
Total instances	151	85	–	63	14	109
`caching`	–	2	–	1	–	–
`explicit<I>`	19	–	–	3	1	–
`concurrency`	47	5	–	2	1	4
`immutable`	22	–	–	–	3	2
`initialized`	32	3	–	1	4	6
`language<L>`	1	5	–	1	2	3
`resource mgmt`	8	3	–	–	–	–
`unique`	22	–	–	–	3	–

Table 4.1. Properties identified during requirements analysis. For each application, the first column represents the number of properties found during requirements analysis, the second column represents the number of reported failures.

exception, a concurrency problem due to incorrect locking, or an incorrect algorithm to compute a result?

- Is there a clear statement in the requirements that is being violated by this failure? For example, an API call returns `null` even though the specification claims that a method never returns `null`.

The first two questions define the dimensions along which issues are clustered. The third question connects the failure clusters to the results from the requirements analysis: an issue for which I cannot identify a clear requirement that is violated will not help in defining property templates. The numbers of relevant issues reported in Table 4.1 reflect this last filtering step. The difficulties of analyzing bug reports are similar to those encountered during requirements analysis, but in many cases were more complex. Consider for example bug reports 42361 and 42409 in the Tomcat bug database. Their commonality is that in both cases incorrect data is returned under certain conditions, and that in both cases this is caused by internally changing data values that should remain unchanged during a single request. However, none of this is immediately apparent from just reading the initial bug reports. Finding these commonalities in both cases requires reading code patches and understanding the impact of the changes contained in these patches. Thus, analyzing bug reports to identify patterns requires even more developer involvement, understanding and judgment to determine commonalities among reports, which based on simple text analysis would have nothing in common at all. The data in Appendix B describes the failure report study in more detail, providing the necessary queries to obtain the dataset used, and lists the mapping between failure reports and identified clusters.

Table 4.1 lists only those properties that have been judged to have sufficiently well

defined semantics and occur frequently. A more detailed analysis of the classes listed in Table 4.1 revealed two important facts:

- Not all classes are homogeneous enough to be amenable to formalization and the definition of a matching property template. For example, resource management problems typically refer to low-level locking and allocation problems that are case specific and not easily generalized.
- Some classes, even though conceptually different, can be captured by others. For example, most caching problems I identified could be detected by a check that the involved classes explicitly implement necessary interfaces instead of just inheriting the required methods.

Consequently, the catalog of property templates in Section 4.2 lists semantics and templates only for those properties that are general and not subsumed by others.

4.1.3 Threats to Validity

The results summarized in Table 4.1 and the details elaborated in the following sections are evidence that some kinds of runtime failures can be clustered as hypothesized. How far these results can be generalized is impacted by three factors: (1) the studies are limited to Java programs, (2) the studies analyze only few applications, and (3) the assessment whether bug reports and specifications are similar is based on human judgment.

Restricting the studies to a single programming language poses a threat to generalizability, and thus overall validity of the results only if the obtained clusters and the subsequent formal semantics contain language specific concepts. Such language specific concepts might be that the classes crucially rely on programming constructs that are only available in some languages and are hard to emulate in others, or rely on other language specific capabilities, like preprocessor macros in C and C++. As the definition of formal semantics in Section 4.2 shows, the semantics does not contain concepts that are inherently language dependent. It is, however, based on object-oriented concepts, and therefore it is not clear how the results would generalize to systems written in languages that do not natively support object orientation.

Analyzing only few applications makes it difficult to draw conclusions that apply to all programs. The data in Table 4.1 shows significant differences in the frequency with which patterns occur in the applications under study. These three applications are of clearly distinct types, Cocoon is a component integration framework, Lucene a search library, and Tomcat a J2EE application server. Hence, the differences in the results suggest that the application type influences which kinds of properties occur more frequently. Unfortunately this also suggests that the results may not generalize to applications of other types, since their behavior might be different from those already studied. The second factor in the study design that influences and potentially

biases the results is that in every study only the specifications for a sub-component have been studied, and for that sub-component only the bug reports for a single release version have been analyzed. This reduces the absolute number of properties and failure clusters reported in Table 4.1, and since the studied sub-components are even more specialized than the overall application, this tight focus of the study is likely to exacerbate the bias introduced by studying few applications of different types.

Founding the analysis on human judgment poses two problems: the first is reproducibility, the second is consistency. In the previous two subsections I outlined the process I followed to define the clusters in Table 4.1. However, no amount of detailed description can guarantee that other researchers will follow the exact same reasoning in all cases, and hence the results may be different. Further, the studies have been conducted over a period of several weeks. It is prudent to assume that growing experience and other factors impacted on each individual judgment that has been made at different times. Hence, the studies, even though they have been conducted by a single researcher, may not be fully internally consistent due to drift introduced by such factors.

4.2 Semantics Specification

The property template catalog in this section lists all available property templates, describes their semantics, and defines the assertions used to detect violations of the properties. It also discusses the stereotypes used to implement the annotations to UML models. In many cases annotations require parameters to fine-tune their behavior. These parameters are provided as attributes to the stereotypes.

During the discussion several conventions simplify the notation:

(1) I use UML terminology when discussing model elements and stereotypes. *Classifier* refers to classes and interfaces. *Attributes* and *operations* map to *fields* and *methods* in Java.

(2) I use JavaBeans terminology when talking about the interfaces exposed by classifiers. Hence, a *property* of a classifier is a data value exposed through getter and/or setter methods.

(3) Since the LuMiNous prototype is based on Java, I use names and class hierarchies from the Java standard library to naturally classify commonalities among groups of classes. Further, the assertion templates use standard idioms and patterns to identify relevant code locations to place assertions.

Unfortunately, this leads to some overloading of terms. A *property* of a Java class is not the same as a property in the sense of the system-level properties that are the subject of this catalog. However, the usage should be clear from the context.

Each catalog entry consists of a summary table and a more detailed textual description of the template. The summary table contains two sections: the top section contains information relevant to developers using the template during requirements

Property	Description
`explicit<I>`	The constrained class must implement a comparison operation matching interface `I`.
`immutable`	The constrained entity may not change its visible state once it is created.
`initialized`	The constrained entity must complete all custom initialization before becoming accessible to clients.
`language <L>`	The constrained entity must be a string and must match a regular expression defining the language `L`.
`notnull`	The annotated parameter or return value must never be `null`.
`singleton`	The annotated class must maintain the semantics of the *Singleton* design pattern [Gamma et al., 1994].
`unique`	The constrained entity must be unique within its context. If the constrained entity is a relation, tuples in the relation must be unique.

Table 4.2. Classes of constraints for property templates.

analysis. It contains the name of the property, which equals the name of the stereotype, a short description of the purpose of the template, a list of model elements the annotation may be applied to, and if applicable a list of parameters. The bottom part of the table contains information relevant to the model transformation. It lists the context elements relevant to a given annotation, the target location of the generated assertions, and a list of formal assertion templates.

Developers are not required to understand the contents of the transformation section to be able to annotate system models. However, the assertion templates represent the most formal definition of a properties semantics, and detailed understanding of the translation process is required to customize the transformations and to provide application specific assertion templates.

Precisely defining the semantics of the properties in the catalog, and the additional constraints implied (for example observability constraints), requires some definitions and conventions. This section defines precise notation to augment the natural language definitions of property semantics.

Definition 4.1 (Classifier)**.** A *classifier C* is a type that consists of a set of methods, denoted *C.methods*, and a set of attributes, denoted *C.attributes*. Attributes may be read-only, in which case the expression *attribute.isReadOnly* has the boolean value *true*.

Definition 4.2 (Subtype)**.** For classifiers C and D, $C <: D$ denotes that C is a subtype of D.

Definition 4.3 (Weak Implementation)**.** A classifier C is said to *weakly implement* a method m, in symbols $C|m$, if there is a classifier D such that $C <: D$ and the concrete implementation of D provides a concrete implementation of m.

Definition 4.4 (Strong Implementation)**.** A classifier C is said to *strongly implement* a method m, in symbols $C\|m$, if and only if the concrete implementation of C provides a concrete implementation of m. Obviously, $C\|m \rightarrow C|m$.

The first four definitions are defining notation for common concepts found in object-oriented languages. The concepts of subtypes, interfaces, and strong and weak implementation are necessary to reason about the semantics *and* the applicability of properties.

Definition 4.5 (Pre- and Post-State)**.** For a given attribute $a \in C.attributes$ and a method $m \in C.methods$ of a classifier C, $a@pre_m$ denotes the value of the attribute a directly before the execution of m, and $a@post_m$ denotes the value of a directly after the execution of m.

Access to system states before and after methods are executed is a prerequisite for assertion languages to be able to assert over state changes.

Definition 4.6 (Assertion)**.** An assertion is a first-order logic formula. Expressions comprising the formula may be subtype, implementation, interface, and state expressions, as well as common mathematical expressions.

To reason about where assertions should be inserted into systems, and under which conditions they must be evaluated, we need notation to describe program locations and execution conditions. The definitions and terminology for program locations are a generalized form of the terminology common in aspect-oriented languages, such as AspectJ[3] [Laddad, 2009].

Definition 4.7 (Pointcut)**.** A pointcut describes either a method call site or a attribute access site in the code.

The syntax for method call sites is:

$$T.m(p_1, \ldots, p_n)$$

where T is a type, m a method name, and p_i are parameter types. T and m may contain '*' as a wildcard matching arbitrary strings. Any p_i, may be replaced by the wildcard '..' matching an arbitrary number of parameters of arbitrary type. The special method name `new` signifies constructors.

[3] `http://www.eclipse.org/aspectj/`

Property:	`comparable`
Description:	Annotated classifiers must directly implement `equals` and `hashCode`.
Elements:	Classifier
Generalization:	`explicit<I>`
Context:	annotated entity
Location:	annotated entity
Assertions:	C.`new`(..)$\{\forall m \in M : C \| m\}$

Table 4.3. Catalog entry for `comparable`.

The syntax for attribute access sites is :

$$set|get(T.f)$$

where *set* and *get* refer to write and read access respectively. T is a typename, f is an attribute name, and both T and f may contain the '*' wildcard to match arbitrary strings.

To define complete assertion templates, assertions have to be associated with pointcuts describing which part of the code they belong to.

Definition 4.8 (Assertion language)**.** Assertion templates are defined as

$$(pointcut\ \{assertion+\})+$$

Following conventions for regular expressions, '+' signifies that each specified pointcut must be associated with at least one assertion.

`comparable`

The `comparable` property is a special case of `explicit<I>`. It addresses the case where language frameworks and libraries provide consistent sets of container classes following a common super interface, or as is more often the case, assume that classes using the framework implement a set of well-defined comparison methods, denoted M.

For example, in the Java Collection Framework M would contain `equals()` and `hashCode()`. The precise contents of M are language and framework dependent, hence they have to be specified by the transformation (see Chapter 6), and not as part of the high-level semantics.

`explicit <I>`

The `explicit` property declares that annotated classifiers must directly implement the interface `I`. By *directly* we mean that the classifier must provide its own implementation

Property:	`explicit <I>`
Description:	Annotated classifiers must directly implement interface `I`.
Elements:	Classifier
Refinement:	`comparable`
Context:	annotated entity
Location:	annotated entity
Assertions:	C.`new`(..)$\{\forall m \in I.methods : C \| m\}$

Table 4.4. Catalog entry for `explicit`.

of the interfaces methods. This property is violated if the classifier only inherits the interface's methods from a superclass.

A good example for when this property is useful is in implementations of the *AbstractFactory* or *FactoryMethod* design patterns [Gamma et al., 1994]. The patterns are centered around the concept of concrete classes explicitly implementing an abstract method or interface that decouple concrete implementations from client code.

This property is particular in the sense that at a first glance it appears that checks for implicit implementation would best be done statically by a compiler. However, even though technically this is possible, to my knowledge no programming language provides declarative means to specify this property. Furthermore, it would be even harder to statically enforce this property across library and component boundaries. The compiled binaries would have to contain enough information to enable static checking of clients, which requires additional infrastructure to be used by the developers of the components and the final system, while runtime checks of this property only require additional infrastructure deployed together with the final system.

`immutable`

`immutable` declares that instances of annotated classifiers must not change their visible state after creation or the invocation of an explicit locking operation. If no locking operation is specified, the object must not change its visible state after instance construction completes. The latter is inspired by Unkel and Lam [2008], who's research indicates that immutable behavior after explicit locking or from a certain location in the code on is typical for many kinds of programs. The current implementation considers the constructor to be the locking operation.

`initialized`

The `initialized` property implies that classifier specific initialization has completed before any references to instances of the classifier are used. The basic semantics for

Property:	`immutable`
Description:	Annotated classifiers must not change their visible state.
Elements:	Classifier
Parameters:	`lock` (optional, default: constructor)
Context:	annotated entity
Location:	annotated entity
Assertions:	C.`lock()`$\{locked \leftarrow true\}$ C.`m(..)`$\{m \in C.methods \setminus \{$`lock(),new(..)`$\} : locked = true \rightarrow \forall a \in C.attributes : a@pre_m = a@post_m\}$

Table 4.5. Catalog entry for `immutable`.

Property:	`initialized`
Description:	Annotated entities must not be `null`, and classifier specific initialization must have occurred when entities are accessed.
Elements:	Classifier
Parameter	`initializer` (optional, default: constructor)
Context:	annotated entity
Location:	clients of annotated entity
Assertions:	C.`initializer(..)`$\{init@post \leftarrow true\}$ C.`m(..)`$\{$`m` $\in C.methods \setminus \{$`new(..),initializer(..)`$\} : init@pre = true\}$

Table 4.6. Catalog entry for `initialized`.

a classifier C annotated with `initialized` state that whenever a method that is not a constructor or the specified initializer is executed, then initialization must have occurred. Note that this refers only to method executions, not to field accesses. The rationale for this is that typical idioms in object-oriented languages suggest that fields be only accessible through dedicated accessor methods. Many languages, for example Objective C [Apple Inc., 2008] and Groovy [König, 2007], even generate accessor methods automatically if developers do not specify them explicitly, hence making direct field access unnecessary.

To allow for various design patterns that require initialization of objects outside the object constructor, the `initialized` property takes an optional parameter that specifies a dedicated initialization operation. `initializer` methods are exempt from the rule that instances of the classifier have to be initialized before operations can be used. If there is no explicit `initializer` defined on an `initialized` classifier, the constructor is considered the initializer. In this case, `initialized` is merely a tag with

Property:	`language <L>`
Description:	Annotated strings must conform to regular language `L`.
Elements:	String values
Parameters:	regexp (required)
Context:	annotated entity
Location:	annotated entity
Assertions:	$C.m(..,S,..)\{S \sim L\}$
	$C.set(S)\{S \sim L\}$

Table 4.7. Catalog entry for `language <L>`.

no effect.

If we are dealing with static singletons like in the Tomcat case study, conceptually the semantics of `initialized` remain the same, but since static methods cannot refer to instance objects the tracking is implemented slightly differently.

`language <L>`

The `language` property declares that a visible string value must match a regular language `L`. Formally, the annotation `language<L>` on a method parameter or class attribute a denotes that $a \in \mathtt{L}$. Since we require that `L` be a regular language, we use common regular expressions to specify `L`. The assertions shown in Table 4.7 should be considered an exclusive choice. If and only if the annotation is placed on a parameter, then the assertions with the method-call pointcut is used, and vice versa.

This property effectively defines method pre-conditions or class invariants. As such it is at the same abstraction level as the implementation, if the language constraint is language specific. However, there are cases, for example the case study in Section 7.3.3, where the language constraint stems from the system requirements rather than implementation decisions.

`notnull`

The `notnull` property is not a direct result of our failure and specification study, but emerged from the definition of more abstract properties such as `initialized` (a reference can only point to an initialized object if the reference is not `null`). Even though the concept of `null` values exists only at the implementation level, this property has been shown to be useful for debugging and quality assessment in component based software (see the `nanoxml` case study in Section 7.2).

Property:	`notnull`
Description:	Annotated parameters or return values must not be `null`.
Elements:	Method parameters, method return values
Context:	annotated entity
Location:	annotated entity
Assertions:	For method parameters p: $C.m(..,p,..)\{p \neq \texttt{null}\}$ For method return values: $C.m(..)\{m(..) \neq \texttt{null}\}$

Table 4.8. Catalog entry for `notnull`.

Property:	`singleton`
Description:	The annotated class must maintain the semantics of the *Singleton* design pattern.
Elements:	Class
Context:	annotated entity
Location:	annotated entity
Assertions:	C`.new(..)`$\{instance@pre = false \wedge instance@post \leftarrow true\}$

Table 4.9. Catalog entry for `singleton`.

`singleton`

The `singleton` property checks that a class maintains the semantics of the *Singleton* design pattern, that is, there is only one instance of the class in the entire system.

Even though the pattern is structurally simple, concurrent programs and language specific issues can introduce corner cases that are easily missed by implementers. For example, the naive implementation in Listing 4.1 is not thread safe. Given two threads $T1$ and $T2$, the sequence $T1.8, T2.8, T1.9-11, T2.9-11$ leads to two instances of the same class being created and returned to the respective callers.

Other, language specific factors, for example the time and order of static variable creation and class loading, can also introduce subtle problems with straightforward implementations of this pattern.

`unique`

The `unique` property declares that instances of a given type must be unique within some context. The context is determined by the annotated model element. An annotation on a classifier declares that instances of this type must be globally unique. An annotation on an association declares that instances must be unique within the context of the annotated association relation. Uniqueness of an instance is determined by an

```
1  public class Singleton {
2    private static Singleton instance = null;
3
4    private Singleton() {
5    }
6
7    public static Singleton getInstance() {
8      if ( instance == null } {
9        instance = new Singleton();
10     }
11     return instance;
12   }
13 }
```

Listing 4.1. Naive Singleton implementation

Property:	`unique`
Description:	Annotated classes must be globally unique. Tuples in annotated relations must be unique within that relation.
Elements:	Association, Classifier
Parameters:	uniquenessProperty (optional, default: `hashCode`)
Context:	annotated entity, association ends, container
Location:	annotated entity or container
Assertions:	for globally unique objects C.`new(..)`$\{o \leftarrow C$.`new(..)`$\wedge \left\vert\{x \in heap@post_{\mathsf{new}} \vert x.\texttt{uniquenessProperty} = o.\texttt{uniquenessProperty}\}\right\vert \leq 1\}$
Assertions:	per-relation unique objects A.`add(..,o,..)`$\{\left\vert\{x \in A@post_{\mathsf{add}} \vert x.\texttt{uniquenessProperty} = o.\texttt{uniquenessProperty} \wedge x\}\right\vert \leq 1\}$
Assertions:	attribute change $C.*(..)\{$`uniquenessProperty`$@pre \neq$ `uniquenessProperty`$@post \rightarrow \vert\{x \in A \vert x.\texttt{uniquenessProperty} = o.\texttt{uniquenessProperty}\}\vert \leq 1\}$

Table 4.10. Catalog entry for `unique`.

attribute of the involved classifiers. By default the `#hashCode` of instances is used. If this is not appropriate, the stereotype defines an additional attribute, declaring which property of the classifier determines uniqueness. Intuitively, monitoring tracks potential changes to the uniqueness attribute and evaluates the assertions every time a change occurred. For `unique` annotations placed on classifiers, the context, and thus the container is the entire heap rather than a specific association[4].

Treatment of annotated classifiers is straightforward. Instance creation and destruction are monitored, and every new instance is compared against all currently existing instances.

Handling of annotated associations is more involved, because the context of the association must be included. First, associations may only be annotated if they represent a 1-to-*n* or a *n*-to-*m* relationship. Instances participating in 1-to-1 relationships are obviously unique and no monitoring is necessary. Additionally, the container must explicitly expose operations to query its contents, or to add elements. Defaults can be inferred from the association's name. Alternatively, the developer could specify the relevant operations through parameters to the annotation (this is not expressed in the above semantics and not yet implemented). For the semantics above, `null` values are not considered objects and they are skipped in the checks for uniqueness.

4.3 Discussion

The results in this chapter show that some software failures can be clustered into classes that correspond to violations of general properties of software systems. It is encouraging that a modest number of studies on medium and large software systems already yield enough data to corroborate this hypothesis. However, the studies producing these results were set up with the goal to determine existence of such classes, not to provide an exhaustive list of them. This explains, at least partially, the relatively small number of classes. That most of the classes focus more on structural than on behavioral features of software systems can be explained by the construction of the studies. Analyzing API specifications yields structural features more easily than behavioral constraints. However, it is to be expected that more extensive studies, or experience from development will yield an increasing amount of properties that relate to behavioral and non-functional properties as well as more structural properties of the type shown in this chapter.

The biggest threat to the validity of the classes produced here is the fact that they are not based on a strictly formal definition, and have been derived using human judgment rather than an algorithm. However, the discussion of the methodology, the data presented and the catalog of property templates provide ample data and justification explaining when human judgment yielded a new property template, and when clusters

[4]The `unique` property on associations is so far the only property that can be directly specified in OCL using the implicit semantics of OCLs `Set` collection.

were dismissed. The `singleton` property is one of those that are on the boundary to being a useful property for runtime detection. Implementing the `Singleton` design pattern correctly is trivial in non-concurrent programs, but significantly more complicated when the pattern must be thread-safe. Nonetheless, in both cases it can be done automatically when the systems code is completely generated from models. However, as part of the Tomcat case study in Section 7.3 shows, non-trivial programs often do not use standard implementations of design patterns, but rather tune them to their needs. In such a setting automatic generation of the pattern itself is no longer possible and faults may be introduced. Then a runtime monitor capable of detecting violations can be helpful in finding very subtle errors due to race conditions or similar when accessing the singleton. But in summary, whether or not the `singleton` property template is useful in a project depends on many other design decisions, not only on the property template itself.

The examples for property templates in the previous section and the case studies using the property templates to detect runtime errors are evidence supporting the claim that meaningful classes of failure exist and that they can be exploited to detect violations of system properties.

Chapter 5

Effective and Efficient Failure Detectors

To automate the generation of runtime checks for property templates, the code structure of these checks must be either pre-defined or be derivable algorithmically by the generator. This chapter defines code templates for the property templates selected in Chapter 4. These templates represent the platform specific part of the approach outlined in Chapter 3. They are implemented in AspectJ, and hence can be applied to applications implemented in languages that translate to Java Bytecode.

The second contribution of this thesis is the definition of effective and efficient runtime monitors and oracles that can detect violations of the property templates defined in the previous chapter. From a high-level point of view, the formal assertions defined in the property template catalog provide the basis for these oracles. However, in practice many factors, for example the deployment platform and language, and the idioms used to translate specifications to code, impact on how the assertions have to be realized, and how and where they have to be placed in the system code.

This chapter discusses the challenges of defining efficient and effective oracles for property templates. The effectiveness of oracles is impacted by the precision of placement and the detail and amount of monitoring data available. The challenge here is to find the best trade-off between the runtime overhead of monitoring and checking, and the precision and recall of the generated checks. The efficiency of the oracles in terms of runtime overhead is a cross-cutting concern. The incurred overhead is partially due to the inherent complexity of the assertions, and partially may be fine-tuned and improved by exploiting idioms and making assumptions about the system structure. All these considerations must be viewed in the light that the goal is automatic generation of oracles from annotated system models. Because these models are on a higher level of abstraction than the system implementation, they contain less detail and information

about implementation specifics. This means that transformations mapping high-level entities into the implementation must make assumptions about these details, for example using idioms and design patterns for translating models to code. The templates presented in this chapter already contain several such assumptions, which are documented for each template. If these assumptions do not hold in the implementation of the annotated system, the generated code has to be modified manually to match the actual implementation.

The property templates defined in the previous chapter define constraints on systems. The formal semantics describes assertions encoding state invariants for the entire system. Consequently, negating those conditions provides test criteria to determine if failures occurred, that is if property templates have been violated. Having such a formal criterion determining system failures is the basis for building oracles, that is software that can determine based on the state or output of a program whether or not the failure criterion has been met, and hence a software failure has occurred. In practice however, even a formal and well-defined failure criterion does not mean that a complete oracle can be defined. The common problems of precision and recall, that is the rate of false positives and false negatives, depends on how difficult it is to define an algorithm that is much simpler than the actual program, but that is yet able to detect deviations from expected behavior [Pezzè and Young, 2007].

Most of the literature considers oracles in the context of software testing. Pezzè and Young [2007] state that oracles, in particular comparison based oracles, are part of testing frameworks that provide the necessary scaffolding and control to put the program under test into a predefined state before executing test cases, and then to extract the necessary information for the oracle to determine the outcome of the test case. When considering oracles for runtime checking, one cannot assume the existence of a framework that controls the execution of the program and allows the extraction of necessary data. Rather, oracles for runtime checking must work while the program under *observation* is executing without the possibility to influence the program, and must be able to decide if a failure occurred based on the data available at runtime. Further, in software testing, several partial oracles can be combined in different test cases to ensure that all critical paths of the software are covered by meaningful test cases. But since runtime overhead is an important factor in a production environment, oracles for runtime checking should be few and very efficient. Hence, simple, yet complete oracles would be ideal, while in practice oracles are partial, and there is a trade-off between precision and performance.

These requirements on the oracles go beyond the common need to be able to detect property violations. They must be taken into account when defining templates for automatic generation of runtime monitors from property template specifications, and make the development of templates more complex than just defining assertions. The monitoring approaches discussed in Chapter 2 usually consider mostly what data is required to monitor the properties interesting to a particular approach. On one ex-

treme end are approaches that use assertions in the code as runtime self-checks. The advantage there is that fine-grained program state is available when executing assertions. However, the visibility of program state for most assertion approaches is limited by the program scope within which an assertion is placed. That means that without additional monitoring code, assertions as defined by `assert()` statements or similar in many programming languages, can not easily reason about global properties, because they have access only to the local state visible within their current scope. On another extreme end, monitoring approaches only utilize data that is available at system interfaces. Such approaches have exactly the opposite problem as assertions and cannot reason about properties that involve fine-grained program state.

The types of checks one wants to perform strongly influence which end of this scale an approach will tend towards. For the code templates discussed later in this chapter, a lot of detailed state information is necessary, hence an assertion based approach appears appropriate. However, many of the properties use local state information in many places to reason over global properties, so facilities to access aggregate information about the system are also necessary. In principle, all this can be implemented using standard assertions defined with the `assert` statement in Java, and extra code that enables access to the aggregate information required. However, the problem with using only this technology is that extra code must be inserted in many places in the system for a single property, and these placements must be updated and changed when the system evolves. Table 7.9 gives an overview of how many different locations in the code are affected by various of the property templates. The numbers there, 50 and more code locations for a single property template, make clear that the simple, language based approach is not feasible unless it is supported by a tool. However, inserting assertion statements in the system code requires the tool to modify the system code, or requires an unnecessarily complicated code structure. Aspect-oriented techniques that weave code fragments non-intrusively, that is without changing the original code, can alleviate the problem, and at the same time provide the necessary tool support. Further, the point-cut concept of aspect-oriented techniques makes sure that all locations described by point-cuts receive the necessary assertion code, no manual intervention is necessary when the system evolves.

5.1 Methodology

To obtain an implementation of property templates that not only matches the formal definitions in Chapter 4, but also meets the non-functional requirements of performance and maintainability discussed above, I defined the code templates in five steps:

1. For every property I selected an application with a known fault that leads to violations of that property.

2. I manually implemented assertions that successfully detect the violation.

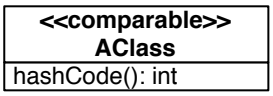


Figure 5.1. Simple application of `comparable`

3. I abstracted the concrete implementation into a template that can be instantiated automatically from model annotations.

4. I applied the tool and template to other applications with known faults of the same type to check if they were successfully detected.

5. Where necessary, I fine-tuned the oracles and make trade-offs between precision and performance.

The decision to build oracles for property templates starting from known faults, rather than trying to directly implement the semantics as defined in Chapter 4 is based on the fact that the code templates for the assertions and the point-cuts describing where in the system the assertions must be checked are implemented on a different level of abstraction from the properties. Thus, the assertion templates, and in particular the point-cuts, encode the mapping from a higher level of abstraction down to the programming language level. This is not trivial and cannot be solved in a general fashion. There are always trade-offs between the generality of the assertions and their precision in detecting violations. The choice to base the assertions on known faults and to validate their effectiveness on an additional test set of known faults assures that the trade-offs are known and allows the fine tuning to implementation details specific to the programming language at hand.

5.2 Templates

The implementation discussed here refers to the transformation from models to AspectJ code. Hence, the templates contain a lot of platform specific information. The code templates contain placeholders of the form `<Placeholder>`. The transformation replaces these with the concrete model elements relevant for each annotation in the model. Table 5.1 lists possible placeholders and what they expand to in the transformation. Since the transformation always views these placeholders within the scope of a single annotation, they unambiguously resolve to model elements that are relevant to this annotation.

`comparable`

The code template for `comparable` is very straightforward. The point-cut attaches the assertion after the static initialization of loaded classes finished. Static initialization

Placeholder	Expansion
`Association`	The name of an annotated association.
`Attribute`	The name of an annotated class attribute.
`Class`	The name of the annotated class.
`Classifier`	The name of the annotated classifier.
`ContainedClass`	The name of the classifier to which an annotated association *points*.
`ContainerClass`	The name of the classifier from which an annotated association *originates*.
`Initializer`	The `initializer` parameter to the `initialized` property.
`Interface`	The name of the interface required in `explicit` implementations.
`L`	The language parameter to the `language<L>` property.
`Method`	The name of a method relevant to an annotation.
`MethodParameters`	The list of all method parameters of an annotated method with their types.
`Parameter`	The name of an annotated method parameter.
`UniqueId`	The value of the `uniquenessProperty` parameter to the `unique` property.

Table 5.1. Template placeholders and their expansions

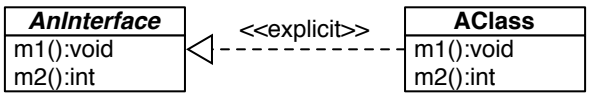


Figure 5.2. Simple application of `explicit`

always finishes before any instances of the class are created, and hence is the right place to check the assertion that the two methods `equals` and `hashCode` are directly implemented.

`explicit<I>`

The close relation between `comparable` and `explicit<I>` also extends to the implementation in AspectJ. Essentially the template is identical, except that it iterates over all methods declared by the parameter interface, instead of the fixed set hard-coded in the `comparable` template.

```
public aspect <Class>_Comparable{
  after(): !cflow(adviceexecution()) && staticinitialization(<Class>) {
    try{
      <Class>.class.getDeclaredMethod("hashCode",new Class[] {});
    } catch (NoSuchMethodException e) {
      logPropertyViolation( "comparable", null,
        "<Class>.hashCode", false );
    }
    try{
      <Class>.class.getDeclaredMethod("equals",
        new Class[] {Object.class});
    } catch (NoSuchMethodException e) {
      logPropertyViolation( "comparable", null,
        "<Class>.equals", false );
    }
  }
}
```

Listing 5.1. Code template for `comparable`

```
public aspect <Class>_implements_<Interface>_explicitly {
  after(): !cflow(adviceexecution()) &&
  staticinitialization(<Class>) {
    Class<?> aClass=<Class>.class;
    Class<?> anInterface=<Interface>.class;

    for (Method m: anInterface.getDeclaredMethods()){
      try{
        aClass.getDeclaredMethod(m.getName(), m.getParameterTypes());
      } catch (NoSuchMethodException e) {
        logPropertyViolation("explicit", null,
          "<Class>." + m.getName(), false );
      }
    }
  }
}
```

Listing 5.2. Code template for `explicit<I>`

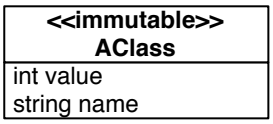


Figure 5.3. Simple application of `immutable`

```
public aspect <Class>_Immutable{
  before(Object _this ): !cflow(adviceexecution()) && set(* <Class>.*)
      && !cflow(call(<Class>.new(..))) && this(_this){
    logPropertyViolation( "immutable", _this, null, false);
  }
}
```

Listing 5.3. Code template for `immutable`

`immutable`

The code template for `immutable` attaches the advice to every attempt to set a value to a class field. Notice that at the moment the possibility for an explicit locking method is not implemented.

`initialized`

The implementation of `initialized` is a straightforward realization of the formal semantics. However, there are some differences between the implementations for static and non-static initializer methods. Listings 5.4 and 5.5 show the templates for non-static and static initializers, respectively. The main difference is that when a non-static initializer is defined, it implies that an instance of the annotated classifier must exist and the initialization must be tracked per instance. With static initializers, this implication does not hold, and initialization is tracked per classifier.

`language<L>`

The specification of the `language` property states that it can be applied to either method parameters or class attributes. These different types of specification can be combined in any way, as the example in Figure 5.5 shows. Transforming this model

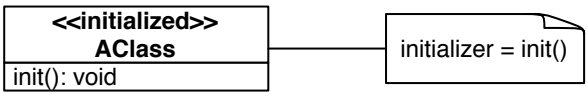


Figure 5.4. Simple application of `initialized`

```
public aspect <Classifier>_Initialized pertarget(target(<Classifier>)) {
  private boolean initialized = false;

  pointcut initMethod(): call(* <Classifier>.<Initializer>(..));
  after(Object target) returning: initMethod() && this(target) {
    initialized=true;
  }

  pointcut checkedMethods():
    call(* <Classifier>.*(..))
    && !call( * java.lang.Object.clone())
    && !call(* java.lang.Object.getClass(..))
    && !cflow(initMethod()) ;

  before(<Classifier> target, Object _this):
    checkedMethods() && target(target) && this(_this)
    && if(_this!=target) {
    if (!initialized) {
      logPropertyViolation( "initialized", target, null, _this, false );
    }
  }
}
```

Listing 5.4. Code template for `initialized`

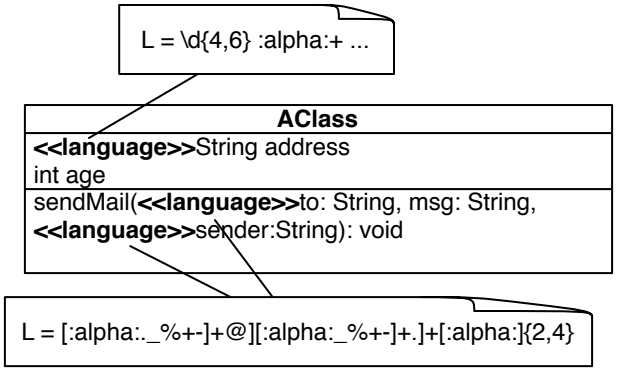


Figure 5.5. Simple application of `language<L>`. `:alpha:` is a mnemonic for `[a-zA-Z0-9]`.

```
public aspect <Classifier>_Initialized {
  private boolean initialized = false;

  pointcut initMethod(): call(static * <Classifier>.<Initializer>(..));
  after() returning: initMethod() {
    initialized=true;
  }

  pointcut checkedMethods(): call(* <Classifier>.*(..))
    && !call( * java.lang.Object.clone())
    && !call(* java.lang.Object.getClass(..))
    && !cflow(initMethod()) ;

  before(): checkedMethods() {
    if (!initialized){
      logPropertyViolation( "initialized", null, "<Classifier>",
        null, false );
    }
  }
}
```

Listing 5.5. Code template for static `initialized` with static initializer methods

```
public aspect <Classifier>_<Method>_<Parameter>_Language{
  before(<Classifier> _target, <MethodParameters>):
  call(* <Classifier>.<Method>(..)) && args(<MethodParameters>)
    && target(_target) {
    if (<Parameter>==null || !<Parameter>.toString().matches(<L>)){
      logPropertyViolation( "language", _target, <Parameter> ,false );
    }
  }
}
```

Listing 5.6. Code template for `language<L>` on method parameters

```
public aspect <Class>_<Attribute>_Language{
  before(<Class> _target, Object arg): !cflow(adviceexecution())
    && set(* <Class>.<Attribute>) && args(arg) && target(_target) {
    if (arg==null || !arg.toString().matches(<L>)){
      logPropertyViolation( "language", _target, "<Attribute>", false );
    }
  }
}
```

Listing 5.7. Code template for `language<L>` on class attributes

```
public aspect <Classifier>_<Method>_<Parameter>_NotNullParameter{
  before(<Classifier> _target, <MethodParameters>): target(_target)
    && call(* <Method>(..)) && args(<MethodParameters>){
    if (<Parameter>==null) {
      logPropertyViolation( "notnull", _target,
      "parameter <Parameter>", null, false);
    }
  }
}
```

Listing 5.8. Code template for `notnull` on method parameters

generates two instances of the aspect template shown in Listing 5.6, one for each annotated parameter (`to` and `sender`), and on instance of the aspect template shown in Listing 5.7 for the `address` field.

`notnull`

The templates for `notnull` checks are straightforward. Listings 5.8 and 5.9 show the simple checks for parameters or return values not to be `null`. The only notable difference between the two templates are the different point-cuts.

`singleton`

The singleton property implies a simple runtime check determining whether or not an instance of the annotated class already exists. Exploiting the fact that AspectJ by default creates only one instance of an aspect and reuses it for all advice executions, we can simply use an aspect-private variable flag signifying whether or not the aspect has detected the creation of an instance or not. If client code attempts to execute the constructor and the flag is already set, this constitutes a violation of the property. The

```
public privileged aspect <Classifier>_<Method>_NotNullReturn{
  after(<Classifier> _target) returning(Object retVal):
    !cflow(adviceexecution()) && call(* <Method>(..))
      && target( _target ) {
    if(retVal==null) {
      logPropertyViolation("not null", _target,
        "return <Method>", null, false);
    }
  }
}
```

Listing 5.9. Code template for `notnull` on method return values

```
public aspect <Class>_singleton {
    private boolean instance_exists = false;

  pointcut constructor(): execution( <Class>.new(..));
  before(<Class> _this): constructor() && this(_this) {
    synchronized(this) {
      if ( instance_exists ) {
        logPropertyViolation( "singleton", _this,
          "before-" + Thread.currentThread().getId(), null, false );
      }
    }
  }

  after(<Class> _this) returning: constructor() && this(_this){
    synchronized(this) {
      if ( instance_exists ) {
        logPropertyViolation( "singleton", _this,
          "after-" + Thread.currentThread().getId(), null, false );
      }
      instance_exists = true;
    }
  }

}
```

Listing 5.10. Code template for a class annotated with `singleton`

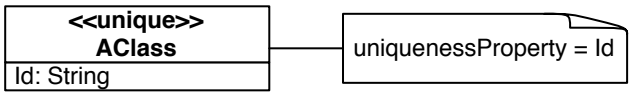


Figure 5.6. Simple application of `unique` to a classifier

synchronization in the template is necessary, otherwise the checking code would be vulnerable to the same data race as the incorrect concurrent implementation of the singleton pattern.

Depending on the action a system manager wants to implement in reaction to such a warning, it is often more helpful to be able to react before a new instance is created. Therefore, the aspect implementation raises the warning *before* the constructor is executed. However, the flag can only be set *after* the constructor successfully returned. Hence, there is a time-lapse between calling the constructor and successfully returning, that can create a race condition in multi-threaded programs.

Note also that this implementation for Java at the moment *cannot* detect multiple instance of the singleton when these are created by different class loaders. This is in line with the typical semantics of `static` variables in Java, but may be counter-intuitive if one is not aware of the special treatment introduced by multiple classloaders within the same application.

`unique`

The templates for `unique` are fully implemented as documented in Section 4.2.

The code in Listing 5.11 shows the relevant parts of the `unique` template. The first `before` advice enables tracking of collection objects that represent instances of annotated associations. This template assumes that 1-to-*n* associations are implemented using appropriate implementations of `java.util.Collection`, and are referenced by a local variable in the class representing the "1" end of the association. The second `before` advice intercepts all calls to `add()` methods, and if the add method is called on a monitored collection, checks if the assertion holds. The assertion checking the elements in the association is completely encoded in the method `assertUnique()`.

When the `unique` annotation is placed on a classifier, like in Figure 5.6, we cannot assume that there is an explicit collection object that will contain all instances of the annotated classifier. Hence, monitoring this constraint requires tracking instances across the entire program heap. The code in listing 5.12 shows the differences to the previous template. Instead of tracking collection objects and calls to their `add()` methods, here we monitor the creation and modification of instances of the annotated classifier. The instances are kept as weak references in a global collection that performs the assertion check when the `addElement()` method is called. The other difference is that when tracking global instances, changes to the attribute values of these instances are monitored. This is necessary, because attribute changes can make two previously

```
public privileged aspect <ContainerClass>_Unique {
  before (Collection collection):
    set(* <ContainerClass>.<Association>) && args(collection) {
    if (collection != null) {
      UniqueCollectionTracker.addCollection( collection );
    }
  }

  pointcut addElement(<ContainedClass> param): target(Collection+)
    && call( * add(..) ) && args( param );

  before(Collection collection, <ContainedClass> param, Object _this):
    addElement(param) && target(collection)
    && !this(UniqueCollectionTracker+) && this(_this) {
    if ( param == null ) return;
    if ( !assertUnique( collection, param ) ) {
      logPropertyViolation( "unique", collection,
        "<ContainerClass>.<AssociationName>", _this, true );
    }
  }

  private boolean assertUnique( Collection collection,
    <ContainedClass> param ) {
    if ( UniqueCollectionTracker.contains( collection ) ) {
      if ( contains( collection, param ) ) {
        return false;
      }
    }
    return true;
  }

  private boolean contains( Collection c, <ContainedClass> object) {
    for( Object comp : c ) {
      if (comp == null ) break;
      <ContainedClass> component = (<ContainedClass>)comp;
      if ( component != object
      && component.get<UniqueId>.equals( object.get<UniqueId>() ) )
        return true;
    }
    return false;
  }
}
```

Listing 5.11. Code template for an association annotated with `unique`

unequal object equal. The template for per-association uniqueness omits this check based on the assumption that updating objects within a collection is discouraged in Java. There are idioms to update objects within collections that essentially remove the object, update and then add it again. With these idioms, the checks based on the `add()` methods will catch invalid updates.

Note that for a specified `uniquenessProperty` it is required that the annotated class, or the class at the "*n*" end of the association, exhibits a pair of setter and getter methods named `set<UniquenessProperty>` and `get<UniquenessProperty>`, respectively.

In practice, the global uniqueness variant where the `unique` annotation is placed on a classifier is less useful, because it tends to generate false positives due the large grain of monitoring. The uniqueness within associations is more practical, but requires that implementors use common idioms and naming conventions for the generated code to match. Otherwise developers will have to adjust names in the generated aspects (see MyFaces example and DaTeC study).

5.3 Discussion

The five steps used to define the assertion templates should be understood as an iterative process. The discovery of a new instance of a property template may create the need to adapt the assertion code to also address that case without compromising the ability to handle previous cases. On the one hand side, the choice to base the assertions on known faults guarantees effective detection of failures due to similar faults, but may also compromise the ability to detect violations of the same property that are due to different kinds of faults.

The assertions presented in this chapter are designed for maximum precision, sometimes at the expense of performance. A part of the performance cost is due to the use of aspect-oriented techniques. Depending on various factors, for example how many dynamic checks are necessary to determine if a join-point matches a point-cut, the overhead introduced by AspectJ can be high, independent of the cost of the constraint checks being introduced as advice. One way to alleviate this problem is to increase the amount of static join-point matching, and to increase the precision of the point-cuts, such that dynamic checks are introduced only when truly required. The case studies in Chapter 7 provide some evidence that the weaving of assertions into the system is very precise in the sense that assertions are only placed where they are needed, and not in spurious locations. However, the assessment if the weaving missed locations that should have received assertions is based on studying the code and deciding whether or not assertions where necessary. This is a tedious, time consuming task for developers and hence error prone. However, the faults used to define the assertions occur in applications whose code is well structured, so confidence that no relevant locations have been missed is high.

```
public aspect <Classifier>_Unique pertarget( instanceCreation(<Classifier>)
  || setUniqueProperty(<Classifier>) ) {
  private boolean lumi_uniquePropertySet = false;
  private Object uniqueProperty = null;

  pointcut instanceCreation(<Classifier> _consumer): this(_consumer)
    && execution( <Classifier>+.new(..) );
  after(<Classifier> _consumer) : instanceCreation( _consumer ) {
    safeAddObject( _consumer );
    lumi_uniquePropertySet = true;
  }

  pointcut setUniqueProperty(<Classifier> _consumer): target(_consumer)
    && call( * <Classifier>+.set*(..) );
  before( <Classifier> _consumer ): setUniqueProperty( _consumer ) {
    if ( lumi_uniquePropertySet ) {
      uniqueProperty = _consumer.<UniqueId>();
    }
  }
  after( <Classifier> _consumer ): setUniqueProperty( _consumer ) {
    if (propertyHasChanged(uniqueProperty, _consumer.<UniqueId>())) {
      lumi_uniquePropertySet = true;
      updateContainer( uniqueProperty, _consumer );
    }
  }

  // return true if uniqueProperty and newPropertyValue differ
  private boolean propertyHasChanged( Object uniqueProperty,
    Object newPropertyValue ) {...}

  //update container contents to match the new key value
  private void updateContainer( Object oldKey,
    <Classifier> updatedObject ) {...}

  private void safeAddObject( <Classifier> newValue ) {
    try {
      <Classifier>_UniqueContainer.<<Classifier>>addElement(
        newValue.<UniqueId>(), newValue );
    } catch (AssertionViolatedException exception ) {
      <Classifier>_UniqueContainer.<<Classifier>>forcedAdd(
        newValue.<UniqueId>(), newValue );
        logPropertyViolation( "unique", newValue, null , null, false );
    }
  }
}
```

Listing 5.12. Code template for `unique` classifier

Chapter 6

Tool Chain and Prototype

Automation is an integral part of the approach introduced in Chapter 3. This chapter introduces the prototype tool developed to validate the approach. It must be noted at this point that even though the approach is independent of implementation platforms and programming languages, tools generating concrete implementations of runtime monitors are naturally tied to the target platform and language. This chapter first discusses general considerations regarding the choice of existing tools and languages. Then it discusses the concrete choices made for the prototype within this framework, and at the end introduces the concrete prototype implementation of the tool chain for Java.

Tools translating design constraints to code-level assertions have to deal with several issues. Baresi and Young [2004] concisely summarize these as (1) language mapping, (2) name mapping, and (3) quantifiers. Design constraints have to be mapped to a form that is executable together with the target system. Baresi and Young [2004] consider only systems that translate such constraints into constructs of the programming language used to implement the target system. However, translations that map to other languages that bind comfortably to the target system are also feasible. Design models are often more abstract and less detailed than their implementations. One of the consequences of this is that names in design models may not precisely map to names in the implementation, and constraint translation tools must have means to establish a connection between names in design models and implementations. Many interesting constraints, be they design-level constraints or class invariants, involve universal or existential quantification over collections of objects. Executing quantified checks can be computationally expensive, and translation tools should offer means for developers to control the runtime cost of the generated assertions.

This chapter describes the prototype tool implemented to experiment with and validate the design-level constraints defined in Chapter 4. The tool addresses the issues

outlined above and addresses other practical issues, such as ease of use and exploitation of existing technologies. The following subsections first discuss the motivation for choosing the Model Driven Architecture (MDA) as the overall framework, the briefly discuss how specifications of property templates are written, and then discusses the implementation of the LuMiNous prototype within this framework.

6.1 The MDA tool chain

Specification languages and tools utilizing specifications written in them exist for all levels of abstraction occurring in software engineering, from requirements engineering down to detailed method pre- and post-conditions, as well as with different degrees of formality, from very informal, like user stories to completely formalized, like algebraic specifications or state machines. However, the higher the level of abstraction of the specification language, the harder it is to transform the specification into code.

The Object Management Group (OMG) defined the MDA to provide a comprehensive modeling and specification framework for all levels of abstraction in system development [OMG, 2003]. Several models capture different aspects of the same system: a *computation independent model* (CIM) captures the requirements and domain concepts of the system, a *platform independent model* (PIM) describes the system structure and functionality at a level of detail that the system's use of the underlying platform is not visible, and *platform specific models* (PSM) are refinements of the PIM for individual platforms that indicate how the PIM is to be realized on a given platform. These models are semantically linked, and a model transformation framework provides for at least partially automated transition from the platform independent to the platform specific models.

Even though the MDA is in principle independent from tools and languages realizing the process, the MDA as defined by the OMG is strongly centered around UML and other technologies, which are also standards defined by the OMG. The following discussion follows this choice and discusses the concrete technologies required to implement an MDA process with the concrete example of UML-centered technologies. The tool chain associated with the MDA consists at least of a model editing tool to create models, and ideally of a model transformation tool that can translate these models between the different levels of abstraction. Since the translation from more abstract platform independent models to more concrete platform specific models may require domain specific semantic information that cannot be expressed directly in UML, several supporting technologies have been developed. The Object Constraint Language (OCL) can express invariants over models, and Profiles are custom extensions to standard UML that can add almost arbitrary types of information [OMG, 2007]. These standards, UML models augmented with OCL and profiles, together with editing tools and a generic transformation framework constitute the normal MDA tool chain.

Since property templates are a form of specification that is language and platform

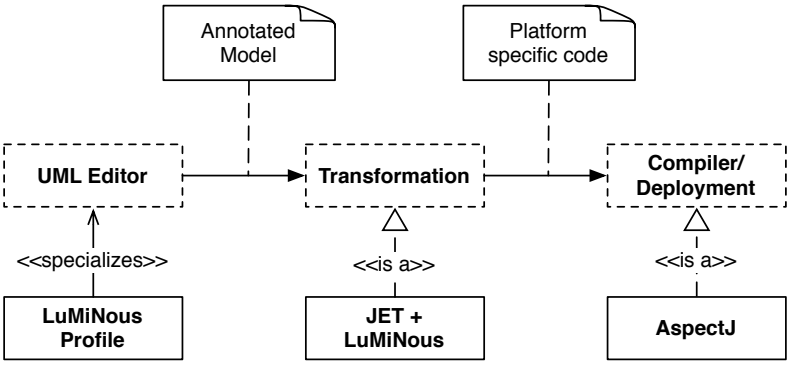


Figure 6.1. Simplified MDA tool chain and LuMiNous extensions

independent, it is natural to take inspiration from an approach like MDA when deciding how to implement a tool for the approach defined in this thesis. Having the annotations for property templates be added at a level of abstraction that corresponds to the platform independent model separates the specification from the concrete implementation of the runtime monitors checking for violations of these properties. In the MDA framework, this then requires a transformation that can translate the platform independent specification into platform specific code. This separation of abstraction levels enables the use of the same annotated PIM to create runtime monitors for different system implementations, running on different deployment platforms.

Figure 6.1 shows the simplified chain of tools and activities that are used for the LuMiNous approach. The boxes with dashed outlines are the standard components of the MDA tool chain, UML editing tools, model transformations, and ultimately compilers or interpreters that compile and run the generated code. The boxes represent abstract concepts, describing what corresponding tools do without prescribing the precise internal workings of them. For the approach here, it does not matter which UML editor is used, but the LuMiNous profile must be used to add annotations to the model. The transformation implemented for the prototype is a concrete instance of the abstract concept of model transformation. The output is AspectJ and Java code that is compiled and deployed by the AspectJ compiler. This tool chain is a simplification, because in a full-size MDA process there would be several transformations, at least from PIM to PSM and then from PSM to code. In the approach here, the distinction between PIM and PSM has no strong impact on the properties, and hence is abstracted away.

6.2 The Model-Based Specification Language

For property templates to be usable with a specification language, the language must be able to express the concepts that the property templates constrain. In the case of the property templates discussed in the previous two chapters, these concepts are

Property Template	Stereotype	applicable to
`comparable`	`comparable`	classes
`explicit<I>`	`explicit`	interface realizations
`immutable`	`immutable`	classifiers
`initialized`	`initialized`	classifiers
`language<L>`	`language`	method parameters, class attributes
`notnull`	`notnullparam`	method parameters
	`notnullreturn`	method return values
`singleton`	`singleton`	classifiers
`unique`	`unique`	classifiers, associations

Table 6.1. Stereotypes defined in the LuMiNous profile

components or classes, their relationships and their means to exchange information, that is component connectors or class methods. Several modeling and specification languages fulfill these requirements, and choices can be influenced by different factors such as tool availability, preference for industry standards, particular specializations of specification languages and others.

The specification language for property templates employed in the prototype is a UML profile. The profile contains the stereotypes listed in Table 6.1. With few exceptions, caused by the typing restrictions of UML, every property template has an associated stereotype that can be used to tag corresponding entities in the model. The semantics defined in Chapter 4 clearly define the scope within which each property template may be applied, and the profile reflects these restrictions.

The distinction between classifiers, classes and interfaces introduced by UML (see Figure 6.2) is only marginally relevant for property templates, but since UML is the underlying modeling language it has to be considered in the design. In UML, interfaces roughly represent what is typically considered to be an interface in software engineering, that is a set of methods, constraints, protocols, etc. In contrast, a class is represents a set of objects that have the same attributes and methods. Both of these concepts are classifiers, which in programming languages are usually types. This distinction is only relevant for `comparable`, which is a property that is only relevant for instantiated objects, and for `explicit`, which directly annotates interface realizations, that is associations that indicate that a given classifier must also implement the associated interface. In all other cases, distinguishing between interfaces and classes is not necessary.

Since several of the techniques for monitoring structural constraints discussed in Chapter 2 use OCL to express the constraints, it is worthwhile to discuss here why OCL is not a good choice to express property templates. In short, using OCL to express

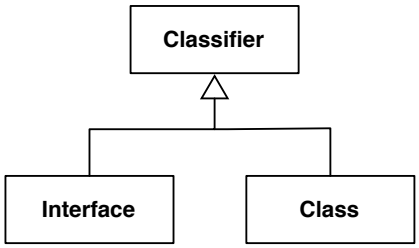


Figure 6.2. UML definition of Interfaces and Classes as Classifiers

the property constraints is very hard, and in some cases even impossible. Consider for example the `immutable` property, which requires that the state of the annotated object does not change. Even assuming that class attributes are only accessed by well-defined accessor methods, expressing this property requires that *every* method of the class be annotated with a post-condition requiring that the state has not changed. Since OCL cannot predicate over meta-model elements, that is it has no expressions to iterate over all attributes, this post-condition must be formulated explicitly mentioning every attribute of the class in the form `attribute@post = attribute@pre`. Adding or removing attributes to or from such an annotated class requires changing all post-conditions. Instead, using property template annotations, either the transformation or the generated assertion can perform the iteration over all attributes implicitly and are thus robust under evolution.

6.3 The LuMiNous prototype

The decision to use UML and profiles as the specification language for the prototype implementation of LuMiNous was influenced by the availability of a wide range of tools that could be utilized. There are a number of open-source UML editors available, and since the introduction of XMI as the interchange format for UML models, tools in general have become interoperable [OMG, 2002]. Mature model transformation tools working on XMI as input format are also available and allow flexible customization of transformations where needed.

Building a tool on these already existing technologies had some advantages. First, having most parts of the MDA tool chain already in place, the prototype implementation can focus on the relevant details of the transformation. Second, using stereotypes to annotate models is a very simple addition to existing modeling processes and is supported by all major UML editing tools, hence there are no practical barriers to using the LuMiNous approach in modeling. Building model transformations that generate runtime monitors from the annotations proved to be more challenging, but again, using mature frameworks, such as the Eclipse Modeling Framework (EMF)[1] kept the

[1] `http://www.eclipse.org/modeling/emf/`

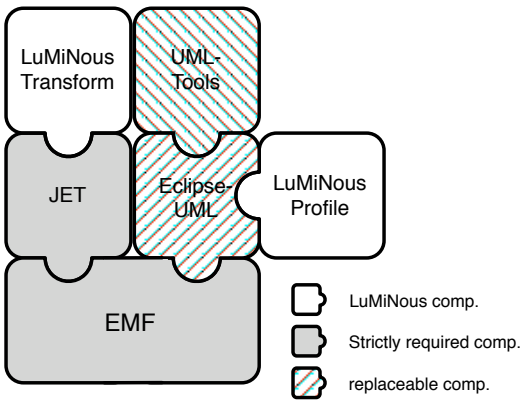


Figure 6.3. The LuMiNous conceptual structure.

necessary effort within reasonable bounds. Overall, from a practical perspective, using MDA oriented, UML based tools and techniques facilitated the implementation of the prototype tool, and also means that there are no or only low barriers for the approach to be used in practice.

For the technical solution, several additional, more detailed design decisions were necessary. The transformation as the platform and language dependent part only generates AspectJ[2] code that can instrument Java applications. As discussed above, the modeling language is UML, augmented with profiles. The model transformation to generate assertions is a JET[3] model transformation.

The prototype is implemented as two Eclipse plugins, one containing the UML profile representing the property template annotations, and one for the model transformation. Figure 6.3 shows the overall structure of the LuMiNous prototype, including the prerequisite components for it to work. The gray components are required libraries and frameworks used by the LuMiNous prototype. The hatched components are replaceable by others. They are a UML editor (UML tools) and the framework connecting UML to the EMF engine below. Since the profile is encapsulated within an Eclipse plugin, it is easy to use LuMiNous within Eclipse, for example by using the Eclipse UML tools or Papyrus[4] to create and annotate system models.

This architecture and the flexibility and extensibility of the used components makes the LuMiNous prototype a good starting point for experimentation with existing and potential new property templates. The case studies in Chapter 7 show how the choices made for the tool, in particular using profiles and supporting partial models, facilitates the creation of models using property templates.

[2] `http://www.eclipse.org/aspectj/`
[3] `http://www.eclipse.org/modeling/m2t/?project=jet#jet`
[4] `http://www.papyrusuml.org/`

Chapter 7

Case Studies

The case studies in this chapter show how the approach discussed in Chapter 3 is applied in practice. Each case study focuses on different aspects relevant to the approach, for example efficiency, usability and the quality of feedback provided. Together, the case studies are evidence supporting the definition of property templates in Chapters 4 and 5, and validate the approach as a whole.

The case studies in this chapter are the second part of the validation of the claims I make with my research hypothesis. Chapter 4 already discusses the claim that failure classes exist, and provides evidence that led to the definition of those classes addressed in Chapters 4 and 5. The cases studies here serve to validate the decisions discussed in those chapters, and support the claim that property templates are useful. For this it remains to define what *useful* means in this context. In summary, the generated runtime monitors and assertions are useful if they

- are effective, that is they can detect violations of the constraints associated with failure classes,
- are efficient, that is they have reasonable runtime overhead,
- can deal with cases not accessible to standard testing,
- provide helpful information to facilitate debugging,
- apply to real cases,
- are usable, that is they are so easy to use in a real-world software development that developers are willing to use them.

Effectiveness, efficiency and scalability are obvious requirements for any automated technique, because if a technique cannot achieve its goals at a reasonable cost,

Study	Criterion
DaTeC	Efficiency, Additional info, real case
`nanoxml`	additional info, real case
Tomcat	inaccessible cases, usability
all	Effectiveness

Table 7.1. Mapping between case studies and evaluation criteria

it is not going to be used. To be a contribution to the state of the art, the technique described in the previous chapters must add something new and useful, or it must improve significantly on already existing techniques. The goal of this thesis is to provide a technique that is capable of detecting new classes of failures at runtime, which by their definition may not be apparent in testing environments. However, the generated runtime checks can also be used during traditional testing. In this case they represent a new contribution if they provide information that makes it easier for developers to debug the system and to locate the faults responsible for the detected failure. Finally, a technique should apply to real cases, rather than only toy examples. The case studies in this chapter are all drawn from active open-source development projects to show that applying the technique is straightforward and requires relatively little effort.

The case studies in the following sections focus on different aspects of the evaluation (see Table 7.1). In the DaTeC study in Section 7.1 I applied the technique to a system with no known faults, and using properties provided by the developers the generated runtime checks identified a subtle corner case that violated the properties. `nanoxml` in Section 7.2 is a hybrid study in which I applied properties to the system and measured how many of the known faults I could find, and if I could identify additional problems. In contrast, the Tomcat studies in Section 7.3 all use known faults, either ones that occurred during development or seeded faults, to assess effectiveness, efficiency and flexibility of the generated runtime checks in a known, controlled environment. Since the focus of the studies is different in each case, the data I report differs in places, because I report and analyze only the data that most pertains to the particular focus of each case study.

7.1 DaTeC

In this case study I applied the LuMiNous approach to DaTeC. When I started the study DaTeC had no known faults, and the goal of the study was to assess how easily runtime checks can be generated and inserted into an application where the source code is unknown. DaTeC is a relatively small but complex static analysis tool developed by Alessandra Gorla and others in the Software Testing and Analysis Lab (LTA) at the University of Milan Bicocca [Denaro et al., 2009]. There were no known faults in

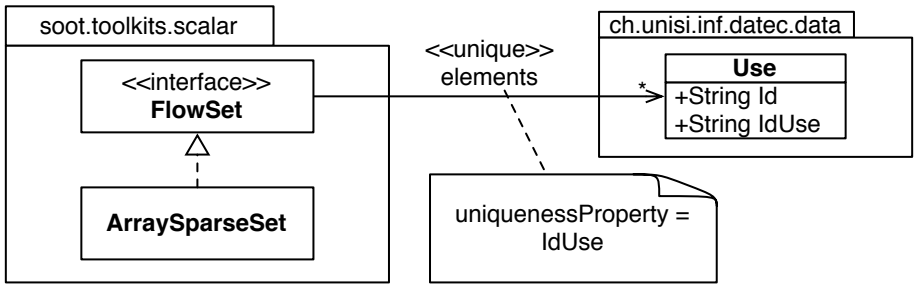


Figure 7.1. Annotated elements of DaTeC and Soot

DaTeC when I started the study, and the developers provided an instance of the `unique` property, applying to `Uses`, a key data structure for the contextual def-use computation performed by DaTeC.

DaTeC is a static analysis tool and builds on the Soot framework[1]. Soot supports the implementation of static analyses by providing a framework that implements the *Strategy* design pattern, and allows users of the framework to supply their own strategies. DaTeC implements such a strategy for the collection of inter-procedural def-use pairs in the class `InterProceduralReachableUsesAnalysis`. Every strategy that collects data in the different steps of an analysis has to store this data in a class implementing the interface `FlowSet`. Soot provides several default implementations of this interface, and DaTeC uses `ArraySparseSet`, which as the name suggests efficiently deals with sparse sets. The instances of these sets collect definitions and uses found by the analysis.

The model in Figure 7.1 shows the `unique` constraint identified by the developers. The code template for `unique` assumes that the aggregating end of the association is a class that contains a reference to a `java.util.Collection`, but because `FlowSet` is an interface, it cannot have instance variables. As a workaround, I annotated `ArraySparseSet`, because that is the concrete implementation of `FlowSet` used by DaTeC. To add more complication, `ArraySparseSet` does not follow the convention to implement the association `elements` with a `Collection` object. Instead, it uses an `Object` array, which necessitates some adjustments in the *typing* of the aspects and monitoring methods. I also had to disable detailed logging, because `ArraySparseSet` has a broken `toString()` method that causes a `NullPointerException` when the `elements` array contains `null` values, which is always the case when the array is first initialized.

To exercise DaTeC in a realistic setting, I used CUP[2], a small parser generator for Java, and two small sample programs provided by the DaTeC developers, *CoffeeMaker*[3] and *Storage* as inputs. While JavaCUP has been developed independently and consists

[1] `http://www.sable.mcgill.ca/soot/`

[2] `http://www.cs.princeton.edu/~appel/modern/java/CUP/`

[3] Originally from `http://agile.csc.ncsu.edu/SEMaterials/tutorials/coffee_maker/`

Property	Caller	JavaCUP	CoffeeMaker	Storage
`unique`	c.u.i.d.a.IPRA	13,427	9	9
	c.u.i.d.a.ReachableUses	203,169	150	150
	s.t.s.ArraySparseSet	110	4,356	4,356
Total		216,706	4,515	4,515

Table 7.2. Number of property violations, by property and caller. The location of the violations is always s.t.s.ArraySparseSet.elements.
Abbreviations for packages and class names:
c.u.i.d.a: ch.unisi.inf.datec.analyses
s.t.s: soot.toolkits.scalar
IPRA: InterProceduralReachableUsesAnalysis

Joinpoint	Count
call(soot.toolkits.scalar.ArraySparseSet.add(java.lang.Object))	4
set(soot.toolkits.scalar.ArraySparseSet.elements)	3
call(soot.toolkits.scalar.FlowSet.add(java.lang.Object))	17
Total	24

Table 7.3. Number of joinpoints woven in DaTeC.

of 38 classes, CoffeeMaker and Storage are toy examples, and consist of only three and two classes, respectively.

Even though there were no known faults at the time when I started the study, there were violations of the `unique` property with every input, as shown in Table 7.2. Detailed analysis of the stack traces and discussions with the lead developer of DaTeC made it clear that these are indeed violations, and not false positives. The large number of violations is due to the fact that these violations remain in the system, and get reported every time a set union, difference, addition or array enlargement is executed[4]. As we will see later in this section, the large number of violations does not prevent developers from quickly locating the responsible fault.

Tables 7.3 and 7.4 show additional data about the experiment runs. There are only a few joinpoints woven, and the matching of woven joinpoints to locations where assertions are truly required is very precise. The three matches on the `set` pointcut, which track object arrays that represent instances of the annotated association, are the only code locations that are relevant for this case study. Also the matches for the two `call` pointcuts occur only in classes of DaTeC and internally to Soot. Even though

[4]The numbers of violations reported for CoffeeMaker and Storage are really the same. This is not a copy/paste error.

Joinpoint	Caller	JavaCUP	CoffeeMaker	Storage
call(FS.add())	c.u.i.d.a.IPRA	2,589	68	72
	c.u.i.d.a.ReachableUses	44,027	649	660
call(AS.add())	s.t.s.AS	3,134,632	10,150	10,176
set(AS.elements)	s.t.s.AS	1,291,717	16,607	18,073
Total		4,472,965	27,474	28,981

Table 7.4. Number of advice executions, by Joinpoint and Caller.
Abbreviations for packages and class-names:
c.u.i.d.a: ch.unisi.inf.datec.analyses
s.t.s: soot.toolkits.scalar
FS: FlowSet
AS: ArraySparseSet
IPRA: InterProceduralReachableUsesAnalysis

AspectJ inserts type checks to ensure at runtime that pointcuts are only executed when all parameter and target types match, a precise static matching reduces the number of required dynamic checks, and hence minimizes the runtime overhead introduced. It is expected that execution and violation counts vary with the input, but since all the *joinpoints* lie within DaTeC itself, the data shown in Table 7.3 does not change when we change the input to DaTeC.

Considering the data in Table 7.2, especially the *JavaCUP* example begs the question how an analysis reporting more than 200,000 violations can help developers to locate the faults causing these violations. The answer is straightforward: The absolute number of reported violations is *not* indicative for the number of faults responsible for these violations. The information logged by the runtime checks allows us to cluster the violations according to 1) the violated property, 2) the annotated model elements, 3) the advice being executed, 4) the context objects, that is the caller and parameters of a particular violations. This clustering reveals that in the DaTeC example only the three code locations within DaTeC shown in Table 7.2 are involved in the violations, and the most frequent one accounts for more than 203 000 of the reported violations. With this information, the DaTeC developers were able to identify the corner case responsible for the violations within a few minutes.

One result of the DaTeC case study that sticks out is the performance impact of the `unique` property. A normal run of DaTeC on the JAR file of JavaCUP takes about 30 seconds. A run with the LuMiNous instrumentation took more then 14 hours, which corresponds to a slowdown factor of about 1700. A more detailed analysis of the log data summarized in Table 7.4 revealed that `ArraySparseSet` is the core data structure used by Soot for its data-flow analysis. Most of the data stored during the data-flow analysis is stored in instances of `ArraySparseSet`, and many expensive operations

like set union, set difference, and the resizing of the internal array representation are executed very often. Since during these operations set elements have to be added to new sets, the assertions are evaluated every time. With an average time of 12ms for each assertion check, this explains the sudden extreme change in execution time. The execution times for the smaller samples were impacted less significantly due to their smaller size, and hence the smaller size of the `ArraySparseSets` at runtime.

In summary, this case study shows that the code LuMiNous generates is effective in detecting new, previously unknown faults, and the extensive logging provided by the checking code allows developers to pinpoint problems quickly. Even though the implementations of Soot and DaTeC do not match all the assumptions made by the LuMiNous transformations, building a usable annotated model proved to be straightforward, despite the complexity of the underlying application. The heavy performance impact of the generated assertions in this case study can be explained by two factors: the relatively large size of the annotated data structure, and the high frequency with which checked operations are called within DaTeC. This suggests that the type of application and the context annotations are placed in can have a strong impact on overall performance.

7.2 nanoxml

In this case study I applied the LuMiNous approach to a small sized application without knowledge about existing bugs to see how effectively we can detect possible problems. We studied the five versions of `nanoxml` hosted by the Software-artifact Infrastructure Repository[5], but due to some of the results discussed in this section, a large part of the studies focus on a more detailed analysis of version v1.

`nanoxml` is a small, non-validating XML parser written in Java. The developer of `nanoxml` states as his main goal to provide a parser that is capable of parsing small snippets of simple XML very fast[6]. Hence, the decision not to validate the XML and not to support advanced linking features that are needed only for complex XML documents.

For this study, I derived a set of possible annotations by studying the API documentation available with `nanoxml` and built a model of the relevant aspects of `nanoxml`. I then executed the test cases provided with the version in the Software-artifact Infrastructure Repository to determine

- how many of the failures highlighted by the test cases we could capture,
- if we could detect additional failures not caught by the test cases from SIR.

However, the test cases provided with `nanoxml` are not unit tests that indicate a clear *pass* or *fail* result at the end. Rather, each test case parses an input file and logs

[5] `http://sir.unl.edu/portal/index.html`

[6] `http://nanoxml.sourceforge.net/orig/NanoXML-Java/introduction.html#nanoxml`

Study	nanoxml
# of classes	16(10)
# of annotations	14
Type of test cases	tsl
# of test cases	214
# of failing test cases	NA
# of detected problems	11

Table 7.5. Summary of the nanoxml study. The number of classes in brackets is the number of classes that contain implementation code, i.e. no interfaces and exceptions. The type of test cases is based on the SIR classification.

Annotation	Entity	Spec'd	Checked
NotNullParam	IXMLBuilder#addAttribute(**key**, value)	x	
NotNullParam	IXMLBuilder#addAttribute(key, **value**)	x	
NotNullParam	IXMLBuilder#startElement(name)	x	
NotNullParam	IXMLBuilder#endElement(name)	x	
NotNullParam	IXMLParser#setBuilder(builder)	x	x
NotNullParam	IXMLParser#setReader(reader)	x	x
NotNullParam	IXMLParser#setValidator(validator)	x	x
NotNullParam	IXMLReader#openStream(systemID)	x	x
NotNullParam	IXMLReader#startNewStream(reader)	x	
NotNullParam	XMLElement#addChild(child)	x	x
NotNullParam	XMLElement#getAttribute(name)	x	
NotNullParam	XMLElement#removeChild(child)	x	x
NotNullParam	XMLElement#setName(name)	x	x
NotNullParam	XMLWriter#write(xml)	x	

Table 7.6. Annotations placed in nanoxml.

its output into a file. Whether or not the absence or presence of exceptions in these logs represent failures or are expected behavior is unclear. Hence, the number of failing test cases is not reported in Table 7.5, and as discussed later, I had to define an alternative criterion to determine the failure of the test cases.

Table 7.5 summarizes the study and its results in a few key numbers. nanoxml consists of only ten classes containing actual implementation code, six more are interfaces and exceptions defined by nanoxml. The versions of nanoxml hosted by SIR come with a set of 214 test cases that test various aspects of the interface exposed by nanoxml and the internal workings of the parser.

Table 7.6 lists in detail which annotations I placed in the model of nanoxml. Addi-

Exception	#
java.lang.ArrayIndexOutOfBoundsException	6
java.lang.IllegalArgumentException	3
java.lang.NullPointerException	15
java.io.IOException	3
net.n3.nanoxml.XMLParseException	18

Table 7.7. Recorded exceptions in nanoxml test cases

tionally, I used an annotated model of the standard JDK API to check for misuses of this API. With the aspects generated from these annotations I was able to capture eleven of the failures occurring during a normal run of the test cases provided with nanoxml.

Test runs with these annotations show that the implementation of nanoxml is inconsistent in the way it implements its own specification. The API documentation often specifies that a particular method parameter must not be null, and for our experiments we only annotated parameters that had such an explicit specification. However, the implementation sometimes checks if a passed parameter is null, and sometimes it does not. The *Checked* column in Table 7.6 lists when I annotated a parameter that was also protected by an explicit check *inside* the annotated method. Note that when the specification originated from an interface, I checked the standard implementation of that interface provided by nanoxml. Further, I also consider a parameter *checked* if the implementation of the method delegates to a JDK API method that specifies that parameters must not be null.

Each test case in the test suite produces one output file, recording program output, messages and exceptions. For the analysis of the results, I consider only those test cases that report exceptions, since they indicate either an API misuse or a fault in nanoxml. The test cases report exactly one of the five possible exception types listed in Table 7.7.

A detailed study of the code revealed that IOExceptions and XMLParseExceptions are both caused by XML input files that the parser considers flawed. Since these exceptions directly relate to the data *content* of the XML files, none of the annotations can capture potential constraints on them. The ArrayIndexOutOfBoundsExceptions occur, because parts of nanoxml do not hide implementation details, such as the use of vectors to store certain elements from the client API. Hence, these exceptions propagate.

The exceptions most interesting in the context of this study are the IllegalArgumentExceptions and the NullPointerExceptions. Since the annotations I placed in the model directly relate to null pointers, these are the exceptions I wish to capture. The IllegalArgumentExceptions occur when test cases exercise methods that do not permit null arguments with null arguments. The annotations capture all these violations as well, but do not provide much additional information. Of the

NullPointerExceptions, about half are due to misuses of annotated methods, and hence the generated runtime monitors capture these misuses, typically long before the exceptions occur, and hence provide information more suited to debugging the true cause of the problem. The other half of the exceptions are cause by unchecked accesses to an internal data structure. XMLElement objects may return null when getName() is called. However, some nanoxml code accesses this method without a following check for null and hence fails. Since the null return is legal behavior in these cases, property templates have no means of checking for this problem.

In summary, all failures relating to annotated methods that are revealed by the test suite are also revealed by the generated runtime monitors. In many cases the monitors detect violations before the actual failure manifests, and thus provide additional information for developers who have to debug the problem.

Since some of the errors occur inside the JDK and then propagate upwards to nanoxml, this study also shows how easily inter-component problems can be detected. I just needed access to an annotated *model* of the JDK, but had no need for the code to be able to detect these violations. On the other hand, the example of the missing internal null check shows that I cannot detect problems when the system design does not separate concerns properly. Had the design included different representations for named and unnamed nodes, this might have been traceable.

In conclusion we can say that even low level annotations like NotNull can help identify the cause of problems more clearly if they are placed with consideration. For example, some of the JDK classes do not specify that parameter values may not be null, even though they use data structures that do not accept such values. Consequently, the data structure throws a NullPointerException, and developers have to search for the cause. In cases where null values have no meaning in the context of the application, such as signaling the lack of data values or similar, such problems could be caught at a more sensible level with appropriate annotations that capture the kind of problem signaled by a null value.

There is one interesting application of the FactoryMethod pattern that is used to create instances of the XML parser. This pattern uses reflection to create an object and then "manually" initializes the instance. This is a variant of the initialized property. However, there is no explicit initializer *inside* the class that has to be initialized, and hence this case is currently beyond the capabilities of LuMiNous.

This case study shows some possible limitations of the LuMiNous approach. There are two important things to notice and discuss:

1. The only properties we identified were several instances of NotNullParameter and a single instance of NotNullReturn.

2. Many of the failures revealed by the test suite have not been captured by our runtime monitors.

Both observations can be explained by the size, structure, and purpose of the stud-

ied application. `nanoxml` is a simple XML parser. That means it is essentially the implementation of a complex *algorithm*. Property templates on the other hand constrain either complex heap data structures or the object-oriented implementation of certain classes. Hence, there is little overlap between the domain of `nanoxml` and property templates. At the same time, property templates regard the *structure* of complex data structures, not their content, while in an XML parser the data *content* is deciding whether or not a particular state is erroneous. The test results of the case study confirm that errors that propagate across class or component boundaries can be captured in most cases, while incorrect parsing or generation of XML files cannot be captured.

Furthermore, even this small study shows that property template annotations can help identify problems that arise from the misuse of libraries, or reveal problems due to inaccurate or missing documentation. One of the failures revealed regards the way `nanoxml` makes use of parts of the JDK API. It uses `java.util.Properties` to store some internal information. However, some test cases executing this feature fail with a `NullPointerException`. It turns out that the class `Properties` extends the default JDK `Hashtable` and overrides several methods. The critical method is `put(key, value)` for adding elements to the property set. While it is documented for `Hashtable` that the key must not be `null`, the overridden method in `Properties` lacks this specification. Further, neither the implementation of `Hashtable`, nor of `Properties` checks for valid parameters, leading to the generic `NullPointerException`. A prudent placement (and consequent checking) of a `NotNullParam` annotation in the JDK can reveal at least the reason for the exception. However, a better solution from the client point of view is deciding where in the client program the responsibility for assuring valid parameters lies, and place the annotation accordingly. With annotations placed carefully like that, possible failures become traceable much more easily, because the final exception is accompanied by a trace of violated system properties telling the developer where things went wrong.

7.3 Tomcat

This section presents the results of four case studies based on the Tomcat application server[7]. The first case study is an application of the `unique` property to a part of the Java Server Pages specification [JSR245], which is implemented by Tomcat. The second case study extends the first, by using different implementations of JSP and JSF to assess if and how much the generated runtime checks have to be modified when one of these components in the deployed system changes. The third case study is an application of the `language<L>` property to a simple security requirement, which corresponds to a cross-site scripting vulnerability that persisted in Tomcat for more than

[7] `http://tomcat.apache.org`

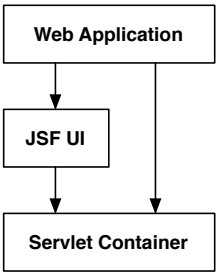


Figure 7.2. Component architecture of the case study

one year before being fixed. The fourth case study is an application of the `initialized` property to the `JspFactory`, a core class of the Jasper JSP compiler.

The assertions generated from the `unique` property effectively detected a subtle seeded fault, and the same property provided the basis for the component-substitution experiment. The assertions generated from the `language` constraint detected a well known vulnerability that allowed cross-site scripting attacks on applications running on top of Tomcat, and the `initialized` constraint was applied to a known fault that has been fixed in later versions, and could clearly identify the precise location of the fault. In all cases, the automatically generated assertions reliably detected property violations before they had serious effects on the running applications, and produced information useful to locate the corresponding faults.

7.3.1 Uniqueness Properties in JSP

In this case study I apply the `unique` property to a part of the JSP specification, which is implemented by the Tomcat application server. The goal of this study is to assess in a controlled environment with a known fault how well the generated assertions for a complex property like `unique` fit into an even more complex implementation of a large application like Tomcat.

This case study illustrates the `unique` property in the context of the interaction of the *JavaServer Faces* (JSF) user-interface (UI) framework [JSR127] with the Tomcat implementation of the Java Server Pages (JSP) specification [JSR245]. Figure 7.2 shows the components deployed in this scenario: A simple web application using the JavaServer Faces UI framework (JSF) [JSR127] deployed on a servlet container.

JavaServer Faces (JSF) is a UI framework for web applications that separates the representation of the UI from the application logic. Developers can define custom tags that can be included into Java Server Pages. These custom tags are a convenient way to define shortcuts for common user interface elements like forms or links, and to connect them to application code that provides their content. The glue code between the tags in JSP pages and the application logic is called *tag handler*. The `tagext` subsection of the JSP specification specifies the requirements for implementing tag handlers.

In this case study, we consider the interface `JspIdConsumer`, a part of `tagext` that contributes to the specification of tag handlers, and thus is relevant for the interaction between Tomcat and JSF. The `JspIdConsumer` specification states that *each tag [implementing this interface] in a JSP page has a unique ID*. This unique ID must be created by the JSP compiler, in our case Tomcat, but the interface is implemented by web applications or frameworks building on the JSP standard, in our case JSF. Additionally, only web applications or frameworks rely on the uniqueness of the generated IDs, while the JSP container is oblivious to them. Hence this specification affects several components that are typically developed by separate organizations. The `unique` property template states that instances of annotated classes must be unique, that is have distinct state, within a given context. The context can be either the entire program or a single association, and is identified by annotating either a class or an association. The context in this case study are individual JSP pages. However, since the containment of tags in a page is not implemented in an association or other container, the instances of `JspIdConsumer` must be globally unique.

To check if the generated code-level assertions can effectively detect and diagnose failures due to duplicate IDs, we used a runtime environment consisting of Tomcat 6.0.9, the MyFaces[8] implementation of the JSF standard, and the JSF Car Demo application, a simple application that is distributed with the Sun JSF implementation to demonstrate the capabilities of JSF. We seeded a fault in the Tomcat's JSP compiler that is responsible for generating unique IDs. Listing 7.1 shows the original code. The commented line in the listing is the fault that we introduced for this case study. The comments in the Mojarra implementation of JSF indicate that a similar fault did exist in earlier versions of Tomcat, and hence this case is not completely unrealistic. We then tested three configurations:

- The correct application *without* the generated failure detectors.
- The faulty application *without* the generated failure detectors.
- The faulty application *with* the generated failure detectors.

We tested the first configuration primarily to establish a baseline against which we could compare the results of the following test cases. As expected Tomcat and the demo application run without problems in this setting.

The test of the second configuration, which corresponds to the Tomcat car demo with the seeded fault but without our failure detectors, leads to a sequence of exceptions when rendering the page. Tomcat returns an error page stating that an internal exception occurred. The exception's stack trace in the log file gives little help as to what caused the failure, since there is no reference to any check for ID uniqueness or similar.

[8] `http://myfaces.apache.org/`

```
private String createJspId() {

  if (this.jspIdPrefix == null) {
    StringBuffer sb = new StringBuffer(32);
    String name = ctxt.getServletJavaFileName();
    sb.append("jsp_").append(Math.abs(
    name.hashCode())).append('_');
    this.jspIdPrefix = sb.toString();
  }
  return this.jspIdPrefix + (this.jspId++);
  //return this.jspIdPrefix + (this.jspId);
}
```

Listing 7.1. Original code and as a comment the seeded fault

The test of the third configuration, which corresponds to the JSF car demo with the seeded fault and our failure detectors, detects the deviation from the specification as soon as an instance with an already existing ID is created, and logs a warning message. The execution stack trace logged by the failure detector points directly to the location where the error occurs in the code, that is the location where the illegal value of `JspId` is assigned. This error message is logged much earlier in the execution than the actual failure while rendering the page, and provides enough information to easily locate the fault.

As a side-effect, our monitoring revealed inefficient memory-usage related to the uniqueness requirement in Tomcat. The Tomcat developers have confirmed this fault[9] and proposed a patch included in release 6.0.19 of Tomcat.

7.3.2 Robustness against Changes

To assess the effects of system changes on the assertions generated by LuMiNous I selected the same system architecture and property as in the previous case study. For this study, I added one additional alternative component for the JSP container and the JSF library, to test how such component substitutions affect the generated runtime checks. For our experiments we used the Apache MyFaces and the Sun Mojarra[10] implementations of JSF, the Apache Tomcat and Sun Glassfish[11] implementations of a servlet container, and deployed the same web application on every possible combination of these four components.

[9] `https://issues.apache.org/bugzilla/show_bug.cgi?id=46397`
[10] `https://javaserverfaces.dev.java.net/`
[11] `https://glassfish.dev.java.net/`

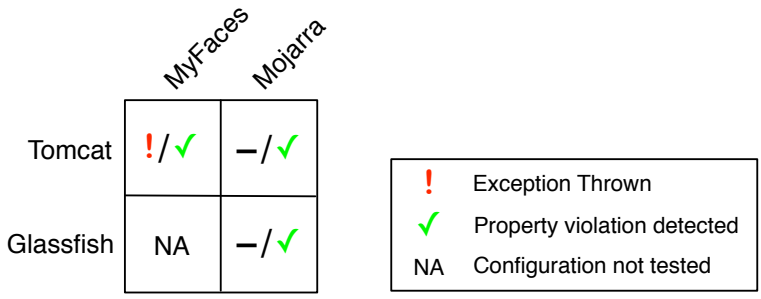


Figure 7.3. Exceptions thrown vs detected property violations in different component configurations.

This case study assesses the robustness of the assertions generated by LuMiNous against changes by answering the following questions:

1. Do we have to change the model to accommodate the different component configurations?
2. How much do we have to change the generated assertions to accommodate the different component configurations?
3. Does the failure detection capability vary with component configurations?
4. How does the failure detection capability compare to exceptions raised without our assertions?

I deployed and ran the `jsf-cardemo` application with the Mojarra and MyFaces libraries in the combinations shown in Figure 7.3. I did not run Glassfish with MyFaces, because Glassfish is strongly tied to Mojarra, and replacing it requires substantial changes to the system. Of the possible component combinations, all ran with assertions generated from the same model. The MyFaces implementation does not follow standard implementation idioms (names for association variables differed from the names of the association in MyFaces), hence the generated code had to be adjusted to capture the correct data. However, this involved changing only a single line of code from

```
... set(* UIComponentBase.children )...
```

to

```
... set(* UIComponentBase._childrenList )...
```

These results indicate that models and the generated code are largely robust against system evolution by component replacement. It confirms our hypothesis that the same

models can be used for different implementations of the same specifications (question 1), and changes to the generated assertions are limited to code peculiarities (question 2).

Figure 7.3 shows the results of the experiments. Each cell indicates whether the seeded fault raised an exception, and whether the assertions automatically generated by LuMiNous revealed and characterized the failure. The data show that the generated assertions reveal and document the failure in all executed combinations (question 3), while exceptions are raised only when running Tomcat with MyFaces. Exceptions are not raised with Mojarra due to a workaround implemented in Mojarra that allows applications to work even with older, broken implementations of Jasper. However, this workaround effectively violates the JSP specification, and masks a fault that ought to be corrected. The optional stack traces logged together with property violations indicate precisely *where* in the code the violation is introduced, while the exception raised while running Tomcat with MyFaces is a `NullPointerException` raised while executing a segment of the code unrelated to the fault, and does not help developers locate the fault (question 4).

The four indicators of stability assessed in this case study show encouraging results. The model does not need to be changed, since the model mostly reflects structure, and not implementation details. The generated assertions have to reflect some changes in the implementation. In this particular case, MyFaces did not follow the standard idiom for implementing associations, and thus the automatically generated code had to be adapted manually. The failure detection capability remained stable over the variations in system configurations, and the value of the runtime information reported together with violations was consistently better than the plain exception stack traces reported only in some cases. Overall, this indicates that LuMiNous properties and assertion templates capture meaningful system features at the appropriate level of abstraction to allow system configuration changes with no or little changes to the model and generated code.

7.3.3 Preventing Cross-site Scripting with `language<L>`

This case study demonstrates how effectively a high-level specification of a `language` constraint could have prevented a security vulnerability in Tomcat[12]. This vulnerability is due to an omission fault that allowed attackers to inject arbitrary HTML and JavaScript code into the error page returned by Tomcat when clients attempt to access restricted pages without authenticating first. With this arbitrary HTML code an attacker could for example send a fake login page from a valid domain and obtain user-names and passwords for that domain.

As most cross-site scripting vulnerabilities, an exploit required a valid web site to pass unfiltered user input to potential error pages. When making a request that causes

[12] `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1232`

an error response, the user input from the request is passed through to the error page and included in the header information. With a tailored request message, the response could thus contain arbitrary HTML and JavaScript code. For the exploit to work, the injected code must be preceded by some *newline* characters, otherwise, the injected code would appear as part of the header text instead of as HTML and scripts.

The fault that enables this attack violates a simple specification that constrains user input to contain only valid printable ASCII characters and not to begin with *newline* characters. Developers can express the constraint by annotating the `#sendError` method of `HttpServletResponse` with the `language <L>` property and require the parameter to satisfy the regular expression `[\p{Print}&amp;&amp;\p{ASCII}]*`.

Even though the constraint appears obvious at the design level, this fault remained undetected in several versions of Tomcat for more than one year, despite the many tests and executions. I used LuMiNous to automatically generate assertions for a Tomcat version that contains this fault, and executed a simple test suite. The generated assertions flag the property violation before a response page is sent, and provide a stack trace directing developers to the point where the violation occurred.

This case shows a particular strength of runtime monitoring over static testing. This vulnerability can only be detected statically if a developer thinks of this possibility and creates a matching test case. Using LuMiNous, it can be detected easily, provided that the designer has a constraint like "*only valid HTML is allowed here*" in mind and annotates the method parameter with the constraint over the user input.

7.3.4 Assuring Application Initialization with `initialized`

This case study demonstrates that assertions automatically generated and deployed by the LuMiNous prototype can effectively detect property violations that occur in code that is generated at runtime, and hence is not accessible to traditional testing techniques. Similar to the previous case study, this study uses a known fault that existed in Tomcat releases for several months, from version 6.0.0 to 6.0.10[13].

The fault prevents timely initialization of a *Singleton* instance of the class `JspFactory`. As a result, when a web application calls the factory method `JspFactory.getDefaultFactory()` early during its own initialization, the method returns a `null` value rather than the expected instance. Since the specification of `JspFactory` guarantees that the instance is available when applications start up, applications may not check if the returned value is a valid reference. Missing checks, rendered obsolete by the specification then cause a `NullPointerException`, thus preventing the application from starting.

The class `JspFactory` in Tomcat is a classical example of the *Singleton* design pattern, and there are well established code idioms to implement the pattern [Gamma

[13]Tomcat Bugzilla bug database: ID 40820, `https://issues.apache.org/bugzilla/show_bug.cgi?id=40820`

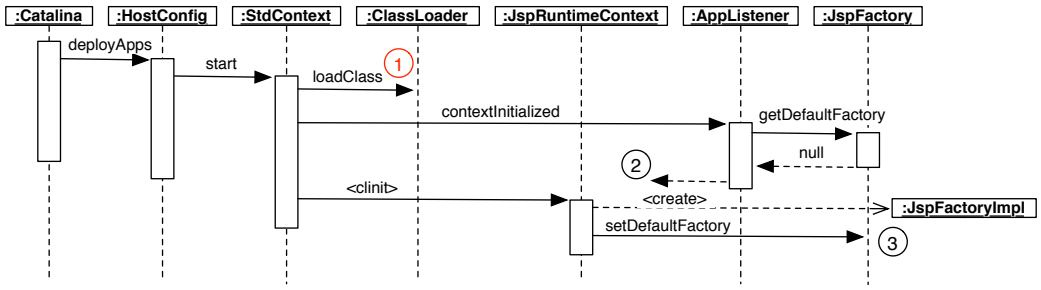


Figure 7.4. The failing Tomcat bootstrap sequence.

et al., 1994]. In the implementation of Tomcat, however, the singleton instance does not follow the standard practice of the design pattern, and this implementation decision led to the fault causing the failures discussed above. This is a typical example of functional problems due to the semantic gap between the design level and the systems implementation.

The actual fault is difficult to locate, because the call sequence involved when initializing the singleton instance is much more complicated than the design pattern would lead one to expect. Figure 7.4 shows the sequence of actions that lead to the failure during the Tomcat bootstrap. The servers main class `Catalina` starts to deploy all installed applications by calling `HostConfig`, which is just in intermediary. Each web-application has a `StandardContext` associated with it. This context controls the state of the web-application relevant to the management of the server. `StandardContext` is also responsible for initializing the singleton `JspFactory`. The `loadClass` call to `ClassLoader` is the faulty attempt at this initialization (marked 1 in the diagram). Then the starting application is notified that its server context has been initialized, which for the application is the signal that `JspFactory` is available. If the application tries to obtain the instance of `JspFactory` in the event handler `contextInitialized`, like it is shown in the figure, a `null` value is returned, leading to the `NullPointerException` marked 2 in the figure. After this potentially failing notification of the web-application, `JstRuntimeContext` is initialized, and only here is the instance of `JspFactory` really created (marked 3 in the diagram). The fault is due to a subtlety of Java class loading. Calls to `loadClass` only load and link the byte-code of the class, but do nothing else. The developers of Tomcat had assumed that when a class is loaded, its static initialization would also be executed. The calls to create the `JspFactory` instance are part of the static initialization of `JspRuntimeContext`, and hence were not executed as expected, but only when the class was being used later. This complicated sequence of initialization might seem like a violation of the clean and tidy design pattern, but complex solutions like this are common in sufficiently complex programs.

From the discussion following the bug report it is clear that for some time the developers had no idea what might cause the reported `NullPointerException` in the web application. The assertions generated from an `initialized` annotation on `JspFactory` report violations and provide semantics for the exception. The fact that the exception was raised, because JspFactory had not been initialized would become immediately clear and direct developers to look at the code that initializes the singleton instance.

This case study shows how the assertions introduced by property templates can help developers locate faults faster, because they provide additional information about failures. It is also a good example of where specifications for one component (here the JSP container) can help debug problems that occur in components utilizing another (here the web application deployed in the container). The code of the web application is mostly generated at runtime by parsing and compiling JSP pages, hence, this code is not available before deploying and actually running the application. This ability to instrument and monitor dynamically loaded or generated code is a particular strength of LuMiNous that distinguishes it from traditional testing techniques, that usually require all source code to be available.

7.4 Summary

The case studies presented in this chapter were designed to provide evidence for the various measures of usefulness defined at the beginning of the chapter. The first measure is *effectiveness*, that is the generated checks should be able to detect failures, and clearly the case studies demonstrate that in all cases failures could be found.

Efficiency was not an issue in most cases. The runtime overhead incurred was negligible for all properties, except `unique`. Proving that a technique is capable of handling problems that are not accessible to standard testing techniques is difficult, because when one starts from a known fault, one can always devise test cases that in principle must reveal the fault under the right conditions. Practically, this often would require enormous effort setting up the testing environment, for example when attempting to build test cases that are guaranteed to find subtle race conditions. In such a case, a technique that monitors the desired property at runtime can work around the need for elaborate testing environments, because the deployment environment will either provide the conditions for the fault to be revealed, and then the monitor will find it, or the deployment environment does not provide the conditions and the fault will never be executed to cause a failure.

There are two facets to scalability in the LuMiNous approach. One regards how defining models and annotations scales with application size, and the other regards how well the number and performance of assertions scale with model and application size. Since partial models are all that is required to generate assertions, the size of an application is not a limiting factor to modeling. Rather, models grow with the number

of annotations. How well performance scales when the number of assertions increases has not yet been assessed and remains an open question.

In all cases the *information provided by the runtime checks* facilitated debugging, not only because often the runtime checks noticed error conditions long before they led to an exception in the system, but also because the fact that each runtime check is associated with a property specified by the developer, means that the warnings carry information about the intention of the developers that is being violated.

Ease of use by developers can be demonstrated in two ways. One way would be to deduce that result from proxy metrics. For example, in all our case studies we could work with very simple partial models of the system that take only a few minutes to create, even if the system contains thousands of classes and associations. Hence, modeling itself is not a big burden and does not require deep understanding of a system before properties can be applied to parts of it. The second way is through a thorough user study in which developers are asked to use the technique and the tool in their daily work, and then later to assess if they were able to do their work faster, better, or if some other relevant metric improved. The first way is straightforward, and if one believes deductive reasoning, then the LuMiNous technique is easily usable. For this thesis I did not conduct a user study, because the technique involves the use of a tool, and the usability of a tool in practice strongly depends on the user interface, stability and reliability of the tool. A research prototype that has been developed as a proof of concept can rarely reach that level of maturity, and hence an evaluation using such a prototype would always be a statement about the prototype tool, rather than the potential usability with a well developed tool. Since developing a tool stable enough for end-user deployment is out of scope for this thesis, a user study seemed to add only little value to the validation and was omitted. Additionally, the component substitution experiment showed that the generated checks are also fairly robust against implementation changes of the system.

The only point where the results fall a bit behind expectations is the efficiency of the runtime checks. In most cases, the runtime checks add little overhead besides the startup cost of weaving the checks into the system. However, in the case of DaTeC, the runtime overhead incurred by the checks caused a execution slowdown of about three orders of magnitude, which even with simple analysis tasks makes the use of these runtime checks infeasible. Analyzing the programs where the `unique` property incurred extreme overhead and comparing them to the programs that were not as badly affected revealed that in most cases, the associations annotated with `unique` are changing only infrequently, while in the case of DaTeC, they changed a lot, because they were at the core of the implemented algorithm. This appears to be a clear case where one might want to trade the precision of detection for more speed. Nonetheless, the other case studies show that with some judgment by the developers, all properties can be applied without incurring heavy performance penalties.

Property	Case Study	Section
`unique`	DaTeC	7.1
	Tomcat	7.3.1, 7.3.2
`initialized`	Tomcat	7.3.4
`language<L>`	Tomcat	7.3.3
`notnull`	`nanoxml`	7.2

Table 7.8. Mapping of which case studies used which properties

Observations

The case studies in this chapter cover a range of different application types, and demonstrate for several properties that the generated runtime checks can indeed detect violations of these properties. Table 7.8 gives a quick reference to which case study utilizes which property. Apart from providing evidence supporting the claims about the utility of the LuMiNous approach, data and experience gathered throughout the experiments led to several observations about the approach and tool.

Context dependent performance. The runtime checks for some properties behave differently depending on the context they are used in. The heavy performance impact of the `unique` property in the DaTeC case study is an example for this. In all the other case studies so far, the impact on performance was small. Execution times for running DaTeC with smaller inputs was dominated by the time required for load-time weaving, the added overhead of the assertions was so small that in most cases it could not be measured at a precision of milliseconds.

It also became clear that some of the properties, namely `explicit` and `notnull`, address developer intent at a lower level of abstraction than the other properties. One consequence of that is that the checks generated from these properties are more likely to reveal problems already during regular unit- and system testing, rather than dealing with difficult and rare corner cases.

Automatic generation and insertion is a *must*. The two main arguments suggesting that manually writing and inserting runtime checks for property templates are that the necessary code can be very complex and that checks have to be inserted in many different, sometimes unforeseeable locations in the code.

The code snippet in Listing 5.12 shows the complexity of some of the property templates. Fortunately, property templates make the necessary checking code boiler plate and enable automatic generation. The data reported in Table 7.9 shows clearly that the number of locations that need to be annotated with code-level assertions can be large, even in small examples. The column *# Advice* shows how many different

Case	Property	# Advice	# Locs.
DaTeC	`unique` (assoc.)	2	24
nanoxml	`notnull` (param.)	14	47
Tomcat*	`unique` (global)	3	56
Tomcat*	`language`	1	30
Tomcat*	`initialized`	2	8

Table 7.9. Number of generated assertions and woven locations.

assertions are created in each case study. All case studies except `nanoxml` use only a single annotation in the model which translates to the shown number of assertions. In the `nanoxml` case there are 14 `notnull` annotations that translate to the given total number of assertions. The *# Locs.* column shows in how many different locations in the code assertions are inserted. As the numbers show, complex properties in medium and large applications require assertions in many places. Furthermore, in the case of Tomcat, the number of locations depends on the applications that are deployed on the server and is likely to increase with the number of applications. Without a means to insert these assertions in all relevant places automatically, developers would have to do this manually, and they would have to check if new assertions are needed every time the code changes. LuMiNous relieves developers from the tedious and error prone task of writing and inserting runtime checks manually.

Dynamic weaving widens the scope of applicability. All case studies involving the Tomcat application server witness to some of the difficulties of directly annotating code with assertions. In these case studies, most woven locations lie within code that is dynamically generated by the JSP compiler. This code is not even available for instrumentation before the web-application is deployed and executed. On-demand deployment through load-time aspect weaving allows us to deal with dynamically generated and loaded code transparently. This also means that the number of woven locations shown in Table 7.9 is not absolute, but depends on the application being executed on top of Tomcat.

Over-approximating weave locations has little impact on performance. Detailed analysis of the weave locations and the executed advice in the DaTeC case study shows that due to a combination of static and dynamic type checks, the weave locations are an over-approximation of the necessary weave locations. That simply means that runtime checking code is inserted in places where it is not needed, but the fact that it is not needed can only be determined at runtime with a dynamic type check. A small experiment aimed at making the approximation more precise statically indicated that reducing the number of weave locations, and hence the number of dynamic type

checks, has only a minimal impact on performance. However, the DaTeC study is an extreme case and it is not clear how these results generalize.

Threats to Validity

The case studies presented in this chapter provide evidence that the technique works. However, there remain some open points and potential threats to validity. Two points that impact on the validity and significance of the results are the number and the type of case studies.

The number of case studies conducted is relatively small and geared towards providing positive evidence for different aspects of usefulness defined at the beginning of this chapter. While an assessment of these factors is of course the goal of all case studies in this chapter, having only few positive cases may be considered a weak argument to support the usefulness of the technique. Many automated failure and fault detection techniques can be applied to standard repositories of applications with a number of know real or seeded faults, and the tools quality can be assessed by how many of those faults they find and how many false reports they generate. These measures of precision and recall would be useful to assess LuMiNous on standard benchmarks, however, none such benchmark exists for the types of properties LuMiNous introduced. And in the case studies that apply LuMiNous to applications with no known faults, it obviously is impossible to derive a measure of recall. On the other hand, precision on almost all case studies is very high, and measurable differences in precision are directly connected to the property checked for, so developers can know what to expect when they apply properties. Another consideration when discussing the number of case studies as threat to validity is the overall goal of the technique. A monitoring technique such as LuMiNous cannot provide guarantees for precision and recall. What it can do, and LuMiNous has been designed with this particular goal and restriction in mind, is provide some detection capability, detecting some classes of failures with high precision, and thus increasing confidence, but not guarantee, that if no such failure is detected, then probably no such failure is present. Hence, the LuMiNous technique must be understood as an augmentation of other existing and still important validation techniques, not as their replacement. And with this goal in mind, providing evidence that LuMiNous can indeed detect some kinds of failure with high precision is good support of the overall claim of the thesis.

The type of case study is limited by two factors. First, the LuMiNous prototype works only with Java, and hence all cases studies are Java programs, and it is unclear if the choice of language has a significant impact on the validity of the results. Second, the case studies draw from server applications, frameworks, and a few algorithms, while some types of applications, like interactive end-user applications, are not covered. The choice of programming language should not pose a strong limitation to the validity of the claims made. As the specification technique uses only common object-oriented concepts, they should be useful for annotating and monitoring pro-

grams written in any object-oriented language or even a mix of languages. However, depending on the programming language, implementing a technical solution for integrating generated checks could be hard. Reflection and dynamic code changes are certainly capabilities of Java and AspectJ that facilitate the implementation of such a tool, while languages that do not directly support these things would probably require a lot more effort to achieve the same results. On the other hand, it would certainly strengthen the results if additional studies show that the properties developed for LuMiNous are applicable in an even wider range of applications than have been tested so far. But being applicable to server applications and frameworks, which represent two very important types of application, is in itself a good result and makes the technique useful.

Besides the number and type of case studies, there is a third factor that might threaten the validity of the results. Some of the case studies presented here use known or manually seeded faults to test the detection capabilities of the generated runtime checks. It is difficult to argue from such experiments that the properties and generated checks are general enough to apply to cases where faults are not known. For the more complex properties, for example `unique` case studies with both known and unknown faults show that at least those properties are general enough to deal with unknown situations. For some other properties, for example `explicit` or `notnull`, one can argue that the necessary checks are straightforward, sometimes akin to what established static analysis techniques can do, but moved to a dynamic environment. For those properties, it could be easier to argue that they obviously generalize well, even without extensive experimental evidence.

Chapter 8

Summary and Conclusion

This thesis developed the idea that specifying constraints at a high level of abstraction and automatically mapping these constraints to runtime checks at appropriate lower levels of abstraction can lead to significant improvements in software quality and can at the same time reduce developer effort. The research presented in this dissertation shows that there is a correlation between functional design-level properties and system failures. Detailed analysis shows that many system failures can be clustered into classes of failures that represent violations of said properties. This correlation between specifications and failures can be exploited to automatically generate runtime monitors and assertions to observe if systems respect the constraints of the specifications. Property templates and the LuMiNous prototype tool are examples of how such failure classes can be used to implement such a mapping from higher-level specifications to lower-level checks.

Chapters 4-6 discuss details of this research and development work. The results clearly support the hypothesis set out in Chapter 1. The hypothesis postulates the existence of failure classes that represent violations of design-level constraints. Chapter 4 discusses examples for such failure classes identified during several software studies conducted during the work on this thesis. However, the results should not be considered absolute. Similar to other kinds of patterns, any catalog can only be a subset of all possible patterns. Therefore, the catalog of failure classes presented in Chapter 4 should be considered as a set of examples. It is likely and desirable that developers identify more failure classes during their work and formalize them in the framework of property templates.

Thinking of property templates as a pattern language similar to design patterns helps to understand the motivation for the rest of the contributions. Like design patters, property templates do not only identify common problems, but also provide a monitoring solution in the form of tried and tested code templates. Chapter 5 discusses everything related to the templates defined for the failure classes identified in Chapter 4. The templates are presented in a concrete programming language rather

than in a more abstract notation. This way they are directly usable at least in the domain of Java applications, and having concrete code for one programming language likely facilitates that code's adaptation for other languages.

A big concern of the approach outlined in Chapter 3 is making the use of the research results practical, rather than showing that it can be done in principle. One of the results addressing this concern is the model transformation tool developed to automatically generate code from annotated models. Many other questions regarding practicality in terms of usability, quality and performance of the approach are addressed by the case studies discussed in Chapter 7.

Considering the hypothesis that *violations of high-level properties lead to failures that are sufficiently similar to each other to cluster them into classes, and to exploit those classes to define general runtime monitors for the failures in each class*, the contributions represented by

- the identification and formal specification of a set of failure classes,
- the definition of templates for runtime monitors for these classes,
- the implementation and evaluation of a prototype tool utilizing the classes and templates for the general runtime monitors.

are evidence that this hypothesis holds. Nonetheless, there are several interesting questions and possible future extensions to the approach that are out of the scope of this thesis.

The property templates are the result of detailed studies of reported software failures. The property templates identified so far focus on a set of properties that relate to structural properties of systems. It is in principle possible to extend the property catalog and the tool with properties of different types, for example behavioral or possibly non-functional properties. Another interesting aspect to explore would be *when* runtime monitors signal problems. The current templates signal errors only when they occur. For some types of problems, it might already be possible to detect conditions that are sufficient for a problem to occur later, and hence, the system might be able to react to such a warning and prevent the further development towards an error. This type of consideration falls into the domain of self-healing software, and is one large field of potential applications for the runtime monitoring techniques developed here.

Some of the case studies in Chapter 7 show that in extreme cases the overhead introduced by the runtime checks can be prohibitive. Two potential ways of improving the performance of the runtime checks would be to consider weakening the constraints expressed by property templates in an attempt to reduce the complexity of the checks. Another approach could be to adapt techniques for incremental constraint checking, as for example proposed by Xu et al. [2006]. In their work they show that incremental checking can deliver orders of magnitude in performance improvements.

Conceptually, the LuMiNous approach works on the level of the platform independent model of the MDA. However, due to the restrictions imposed by available implementations of the MDA, the case studies have been applied to the platform specific model, where the technology platform and programming language are already known. To extend and reuse the technique and the developed property template catalog for other platforms, both need to be lifted to the PIM-level and an additional level of transformations to the PSMs has to be introduced.

In summary, this dissertation developed the notion of property templates, provided evidence that meaningful property templates exist, and developed a prototype tool that provides proof that property templates work for automatically generating runtime monitors.

Appendices

Appendix A

Using and Extending LuMiNous

This chapter contains details on the implementation of the two Eclipse plugins that constitute the tool prototype. Both plugins make extensive use of existing frameworks, which effectively means that the parts explicitly implemented for the tool are only small extensions within these frameworks. For this reason, rather than discussing implementation details of the tool separately, all relevant detail in incorporated into the documentation of how to use and extend the tool.

A.1 Installing LuMiNous

The latest version of the LuMiNous plugins is available for end-users via the update site `http://www.inf.usi.ch/phd/wuttke/resources/eclipse`. The transformation requires a complete installation of Eclipse UML and the JET engine to work, even though the plugin installation does not check these requirements.

The sources of LuMiNous are available as well. To extend the templates or the profile, the source code of the plugins is required. The source code is structured in several projects, matching the various plugins and should be added to an Eclipse workspace as that. Using the profile requires the installation of the profile plugin, as profile plugins cannot be loaded from a workspace. Besides this limitation, using the development installation is the same as the end-user installation.

A.2 Using LuMiNous

Using LuMiNous is straightforward and consists of two steps:

1. Defining and annotating models.

2. Generating runtime checks.

Defining and annotating models

In principle, models can be defined in any UML editor. In practice, slight differences in how different editors represent associations, it is recommended to use either the standard editor of the Eclipse UML tools, or Papyrus[1].

Model annotations are applied via UML stereotypes contained in the plugin `ch.unisi.inf.luminous.profile`. The stereotypes are constrained to be applicable only to model elements for which valid checks can be generated, and these constraints match those documented in the semantics in Chapter 4. To annotated a model with property template constraints, simply apply the appropriate stereotype to the entity you want to constrain, and add supply the parameters to the stereotype where necessary.

Generating runtime checks

The generation of runtime checks is also straightforward, provided the annotated model was created within Eclipse with one of the recommended tools. If other tools were used, the model might need adjustments before the code generation plugin can parse it.

Even though we use XMI as the input format for the transformation, there are some minor restrictions to the content of the XMI file. In principle, the `name` attribute of all entities must be set to a non-empty string. Otherwise the transformation will fail to look up related objects occasionally.

To generate the runtime checks, create a new run configuration in Eclipse. The type of the run configuration is "JET Transformation". Select the `ch.unisi.inf.luminous` transformation in the appropriate field and specify the name of the model file you wish to process (the model file must be contained in a project). You can reuse this run configuration to regenerate the runtime checks when you change your model.

Running the run configuration generates a number of AspectJ and Java files within the project that contains the model file. If your project already has the AspectJ builder associated with it, the generated code will be compiled automatically. If not, you can convert your project to an AspectJ project to enable this automatic compilation.

If the project in which you generate the runtime checks also contains the code of the program you wish to check, then the checks will be statically woven into all available classes. If the code is not within the same project, you can either weave the code statically by referring the AspectJ compiler to the project containing your runtime checks, or dynamically by using load time weaving. The code generator already provides you with an `aop.xml` file that defines all the aspects to weave. Since the setup of load time weaving is highly project specific, we cannot give any guidelines here and must refer you to the AspectJ documentation.

[1] `http://www.papyrusuml.org`

Notes

- There is a default aspect `ch.unisi.inf.luminous.Timer` that prints the date and time of entering and exiting the `main()` method to `stdout`. NB: This time measurement does not capture weaving times happening before `main()` starts executing, but will capture anything that happens within the execution context of it. This aspect can be safely removed from weaving if it is not required for debugging purposes.

- The advice execution log messages contain the execution time of advice in seconds, at a precision of milliseconds as provided by `System.currentTimeMillis()`.

- When creating a model, applying the profile, but not using the profile, will lead to failing runs of the transformation. Ergo, if you apply the profile, you must use it at least once, i.e. annotate one element in the model with a stereotype from the profile.

- By default advice gets woven everywhere, including `javax.*` library classes. When using LTW (and maybe even with static weaving), some of the weaving can be avoided by providing explicit `<include within/>` statements declaring within which packages and classes aspects should be woven. This can limit the amount of weaving done, and effectively reduces the amount of dynamic joinpoint match checks and possibly the number of advice executions. Depending on the problem this may help to increase precision and reduce overall processing time.

- By default printing stack traces of violations is turned off. The default can currently only be changed by editing the templates. Turning trace logging on in individual cases requires editing the calls to `Utils.logPropertyViolation`.

- When using dynamic weaving, editing the generated `aop.xml` allows fine-grained control over where advice is inserted and which generated aspects are considered for weaving.

A.3 Extending LuMiNous

This documentation regards extending LuMiNous with more templates for transformations from UML to AspectJ. The definition of new properties requires the definition of an appropriate stereotype in a UML profile and the definition of a JET transformation that generates the code. Extending the tool to support other output formats requires only the specification of JET templates. The templates discussed in this dissertation are contained in the plugin `ch.unisi.inf.luminous.transform.aj`. Extensions may

be defined either in the same plugin or as a separate plugin. If a completely new transformation to a new language is desired, a new plugin should be defined, and not all the instructions below may apply.

Defining new Annotations

New stereotypes may be defined either in the standard profile contained in the `ch.unisi.inf.luminous.profile` plugin, or in a separate profile. To add more stereotypes to the profile, follow these steps:

1. Use the UML editor to create a new stereotype.
2. To make the stereotype applicable to UML elements
   (a) Make sure that the profile imports the UML elements you want to annotate.
   (b) Use the UML Editor to create an extension (`Stereotype --> Create Extension`)
   (c) Delete old profile definitions.
   (d) Define the profile.
   (e) Export the profile plugin and install it in your Eclipse.
   (f) Restart Eclipse.
   (g) Open your model, remove the profile application from the model manually (or do that with the menu as the very first step of this process)
   (h) Apply the profile to your model.
   (i) You should now be able to use the new Stereotype in your models, and all old uses should still work, even though the model editor doesn't show them anymore.

Defining new Templates

At the time of writing, the EMF model parser does not support UML profiles, hence applications of stereotypes to model elements have to be located by searching the XML DOM for these applications. For this reason, defining new templates is a little tricky. It requires two distinct steps: first, adding calls to the template from `main.jet` whenever an XML element matching the new stereotype is encountered, and writing the new template itself.

`main.jet` is the control file of any JET transformation and directs all actions when processing an XML file. The code for the various existing templates can usually be adapted for a new template call. The code below shows an instructive example:

```
<c:iterate select="/XMI/Immutable/@base_Class" var="classId">
    <c:setVariable select="//packagedElement[@id=$classId]"
```

```
    var="class"/>

    <c:setVariable select="'class'" var="classVar"/>
    <c:include template="templates/setClassPackage.jet"/>
    <ws:folder path="{$classFolderName}">
      <ws:file template="templates/Immutable.aj.jet"
             path="{$classFileName}_Immutable.aj"/>
    </ws:folder>
</c:iterate>
```

The `<c:iterate ...>` instruction iteratively selects all instances of the stereotype that should be handled by the template. The two calls to `<c:setVariable>` and the call to the first template `setClassPackageName` are necessary to create the code from the template in the correct Java package. The LuMiNous convention is for an annotated class `x.y.C` to create an aspect in the package `x.y.aspects.C_property`. Not making the three calls described above will enable you to change this default. The `ws:folder` instruction creates the folder with the name in `classFolderName` is needed, and execute all nested instruction, that is store all template generated code, inside that folder. The `ws:file` instruction processes the template and creates the file named in the parameter `path` from the output. More complex templates may need to extract more information from the XMI file before calling the templates, or the templates may extract that necessary information. The `unique` template is a good example for more advanced processing.

Each template must begin with the following code:

```
<%-- --%>
<% //generating the corresponding aspect in aop.xml
String aop=context.getVariable("aopxml").toString();
aop=aop.concat("<aspect
name=\"").concat(context.getVariable("classPackageName").toString());
aop=aop.concat(".aspects.")
  .concat(context.getVariable("classFileName").toString());
context.setVariable("aopxml",aop.concat("_Immutable\"/>\n"));
%>
```

Only the string `_Immutable` should be replaced with something unique to the new template. This code mostly generates appropriate package names for the generated code and unique names for the aspects generated. Parts of this code deal with bugs in older versions of the JET engine, and parts just collect and set important local variables. The rest of the template is standard JET XML code and can contain whatever the template needs.

Appendix B

Data

In this chapter we report detailed results of the studies we carried out. Each subsection presents the data for one application, and the final section summarizes the data to derive conclusions about the number and type of property templates we found. Table B.1 lists all applications that we studied. The type gives a rough idea of what the application is designed for.

The sections below (1) discuss the documents we used for requirements analysis, summarize the results from requirements analysis, (2) describe precisely which queries were run against the issue repositories to retrieve the base set of issues to study, and provide a summary of the clusters during failure analysis. More details on the "types" of the individual clusters are provided in Chapter 4.

Cocoon

The requirements analysis for the Cocoon core component focused on the version 2.2 of the APIs of the core packages[1]. The study proved tedious, because the core component is split into 16 highly interdependent sub-components, with their source, and thus their API documentation, spread across as many different Maven projects.

The implementation of Cocoon makes heavy use of the *Factory* design pattern. It also has a deep inheritance hierarchy and uses many small interfaces to spread cross-

[1] `http://cocoon.apache.org/2.2/core-modules/`

Application	Type
Cocoon	Web-Application Framework
Lucene	Text-search library
Tomcat	Server

Table B.1. Applications studied.

Property	Number
Total instances	151
`comparable`	19
`concurrency`	47
`immutable`	22
`initialized`	32
`language`	1
`resource mgmt`	8
`unique`	22

Table B.2. Properties extracted from the Cocoon API.

cutting behavior throughout the code. Together, these two facts explain the high number of occurrences of the `initialized` property reasonably well. Cocoon inherits the `SingleThreaded` and `ThreadSafe` marker interfaces from the Avalon project[2], which are used to explicitly place concurrency constraints on several core interfaces, thus the property is inherited by a large number of implementation. Other properties occur less frequently.

Table B.2 shows the accumulated occurrences of properties across all sub-components of Cocoon Core and Table B.3 shows for which classes these properties have been identified. In general, the high numbers are due to the fact that often constraints are attached to interfaces instead of individual classes, and we assume that these constraints are "inherited" by implementing classes.

The bugs we analyzed during the study of Cocoon were retrieved from the Cocoon bug database via a custom query. The Apache's JIRA issue tracker is located at `https://issues.apache.org/jira/secure/IssueNavigator.jspa`. The settings for our query are[3]:

Parameter	Value
Project	Cocoon
Issue type	Bug
Components	Cocoon Core
Affects Versions	Released Versions
Resolution	"Unresolved" or "Fixed"
Created before	2008-04-12

This query retrieves 290 reports. We analyzed only the first 100 reports from the retrieved list (sorted by ID), and after removing reports that were clear duplicates, we

[2] `http://excalibur.apache.org/`

[3] "Released Versions" include version 2.2, which we used for the requirements analysis.

Property	Classes, Interfaces
comparable	(I) o.a.c.caching.CacheableProcessingComponent
concurrency	(I) o.a.c.components.modules.output.OutputModule, o.a.c.modules.input.*, o.a.c.components.source.impl.DelayedRefreshSourceWrapper, o.a.c.components.treeprocessor.SimpleSelectorProcessingNode, o.a.c.core.container.spring.PoolableFactoryBean
immutable	(I) o.a.c.caching.CacheableProcessingComponent, o.a.c.util.location.LocationImpl, o.a.c.components.pipeline.PipelineComponentInfo, o.a.c.components.source.impl.MultiSourceValidity
initialized	(I)(trans) o.a.c.components.xslt.XSLTProcessor, o.a.c.components.xslt.XSLTProcessorImpl, o.a.c.components.xslt.TraxProcessor, o.a.c.generation.*, (A) o.a.c.serialization.AbstractTextSerializer, (A) o.a.c.components.pipeline.AbstractProcessingPipeline, o.a.c.thread.impl.DefaultThreadPool
language	o.a.c.components.modules.input.DateMetaInputModule
resource mgmt	(I) o.a.c.components.treeprocessor.TreeBuilder, (A) o.a.c.components.treeprocessor.SimpleSelectorProcessingNode
unique	(I) o.a.c.caching.CacheableProcessingComponent, o.a.c.sitemap.SitemapModelComponent, o.a.c.sitemap.SitemapOutputComponent, o.a.c.sitemap.DefaultContentAggregator

Table B.3. Classes or interfaces of each identified property in the Cocoon API. o.a.c refers to the common top-level package org.apache.cocoon. (I) marks interfaces, and (A) marks abstract classes.

were left with 85 useful reports. The decision to not completely analyze all reports in this study is based on the observation that most reports, even those created several years ago, are not discussed, closed or in any other form commented on. This led us to the conclusion that the issue reporting system is not well utilized by the developers and thus cannot give us a realistic view of the issues that occurred in the system.

Lucene

For the study of Lucene, we picked the *Search* component of the Java implementation. This component is used to search a pre-generated index of files for terms or expressions. Studying the API documentation of this part of library turned out to be fruitless, because there is barely any documentation beyond class- and method signatures.

For the fault analysis the query parameters below retrieve 69 issues, which after

Class	Number
Total bugs	85
`caching`	2
`concurrency`	5
`initialized`	3
`language`	5
`resource mgmt.`	3
other	4

Table B.4. Cocoon Issue Clusters

Class	Number
Total bugs	63
`caching`	1
`comparable`	3
`concurrency`	2
`initialized`	1
`language`	1
other	5

Table B.5. Lucene Issue Clusters

dropping the usual duplicates, left us with 63 issues to study.

Parameter	Value
Project	Lucene-Java
Issue type	Bug
Components	Search
Affects Versions	Released Versions
Resolution	“Unresolved” or “Fixed”
Created before	2008-05-06

As the data in the summary table (Tab. B.5) show the results are less clearcut as in most other cases. A large fraction of the issues that we could track back to some end-user requirement violation do not fall into one of the classes we have identified so far, but occur as singletons, that is, they occur only once in the entire analysis. This might indicate that the classes we identified so far occur less frequently in tightly coupled applications.

Property	Number
Total instances	14
comparable	1
concurrency	1
immutable	3
initialized	4
language	2
unique	3

Table B.6. Properties extracted from Tomcat requirements

Property	Classes, Documentation section
comparable	javax.el.Expression
concurrency	javax.el.ExpressionFactory
immutable	javax.servlet.jsp.PageContext, javax.el.Expression
initialized	javax.servlet.Filter, javax.servlet.Servlet, javax.el.ELResolver, JSP.11.2.1
language	javax.servlet.jsp.PageContext, SRV.2.6.5
unique	javax.servlet.http.Cookie, javax.servlet.jsp.JspApplicationContext, javax.servlet.jsp.tagext.JspIdConsumer

Table B.7. Classes and documentation sections of each identified property. JSP refers to the Java Server Pages specification, SRV to the Java Servlet specification.

Tomcat

We focused our requirements analysis for Tomcat on the API specifications for Java Servlets [SRV] and Java Server Pages [JSR245]. These APIs lie at the heart of Tomcat and define much of what it has to do as a server. In addition we studied the end-user documentation regarding server configuration and other aspects that are not explicitly covered by the APIs. Table B.6 lists the number of occurrences of properties identified in the API and end-user documentation without in-depth study of the software architecture and design. Note that the numbers here are much lower than for example for Cocoon, because we studied only the API definitions, but now how Tomcat implements them. Thus the multiplying effect of inheritance is not reflected in these numbers. Table B.7 shows the classes affected by the identified properties.

The bugs that entered our study for Tomcat were retrieved via a custom query to the Apache Bugzilla issue tracker. We selected only “Tomcat6” as a product, all compo-

Class	Number
Total bugs	109
`concurrency`	4
`immutable`	2
`initialized`	6
`language`	3
other	1

Table B.8. Tomat Issue Clusters

Property	Issues
`concurrency`	42753, 42803, 42840, 43846
`immutable`	42361, 42409
`initialized`	39355, 40012, 40820, 41797, 42077, 42934
`language`	41521, 42683, 43338

Table B.9. Mapping between properties and bug report numbers for Tomcat.

nents except “Documentation” and“Examples”, all status flags except “NEEDINFO” and “VERIFIED”, with resolutions either *none* or “FIXED”, and severities above “enhancement”. Since we conducted the study in two steps, we had to exclude eventual new issues recorded in between by allowing only change dates before 2008-04-23. Running a query set up like this, we retrieved 140 reports, out of which we then dropped 30 as being documentation and example related, and one as clear duplicate that was copied from earlier versions of Tomcat. The table below summarizes the clustering for these remaining issues, and Table B.9 lists which issues have been identified as matching which property.

Summary

The results obtained in our study have several implications: first, they corroborate our hypothesis that a reasonable number of problems occurring in software can be classified according to our scheme. Table B.10 summarizes the results. It lists the total number of code bugs analyzed for each application, and how many have been found to match our criteria of violating end-user requirements. It also gives a brief overview of how many distinct classes the identified failures fall into, and which proportion of the identified failures falls into these classes (coverage column). If the coverage is less than 100% this indicates that there are some failures that we consider relevant to our

Application	Bugs		Classes	Coverage
	total	relevant		
Cocoon	85	22	5	86 %
Lucene	63	13	5	61 %
Tomcat	109	15	4	94 %

Table B.10. Results of the failure analysis for the three selected applications. *Total bugs* reports the number of code bugs, *relevant bugs* are those that have a clear connection to a requirement in the user or API specification, *Classes* reports the number of different classes into which more than one of the relevant bugs fall, and *Coverage* is the percentage of relevant bugs that fall into one of the classes.

study, but that we could not place in one of the classes described at the beginning of this section.

The results for the three different types of applications we analyzed vary enough to hypothesize that the application type has an effect on the number and distribution of property templates that occur. In particular the Lucene study may indicate that tightly couple components, like they are typical within one library or application, may not exhibit as many clear cut cases as other application types. This possibility encourages us to further explore this connection, since it would also corroborate our hypothesis that the types of failures captured by property templates are typical integration failures, which should be comparatively rare within a tightly coupled library. However, we need more data and further studies to be able to draw statistically valid conclusions.

Bibliography

Aldrich, J., Chambers, C. and Notkin, D. [2002]. Archjava: Connecting software architecture to implementation, *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, ACM, pp. 187–197.

Allen, R. J. [1997]. *A Formal Approach to Software Architecture*, PhD thesis, Carnegie Mellon University.

Ammann, P. E. and Knight, J. C. [1988]. Data diversity: An approach to software fault tolerance, *IEEE Transactions on Computers* **37**(4): 418–425.

Apple Inc. [2008]. *The Objective-C 2.0 Programming Language*.

Avizienis, A. [1985]. The n-version approach to fault-tolerant software, *IEEE Transactions on Software Engineering* **11**(12): 1491–1501.

Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr, C. [2004]. Basic concepts and taxonomy of dependable and secure computing, *IEEE Transactions on Dependable and Secure Computing* **1**(1): 11–33.

Baah, G. K., Gray, A. and Harrold, M. J. [2006]. On-line anomaly detection of deployed software: a statistical machine learning approach, *SOQUA '06: Proceedings of the 3rd International Workshop on Software Quality Assurance*, ACM, pp. 70–77.

Bahati, R. M., Bauer, M. A. and Vieira, E. M. [2007]. Policy-driven autonomic management of multi-component systems, *CASCON '07: Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, ACM, pp. 137–151.

Baresi, L. and Young, M. [2004]. Toward translating design constraints to run-time assertions, *TACoS'04: Proceedings of the International Workshop on Test and Analysis of Component Based Systems*, Vol. 116 of *Electronic Notes in Theoretical Computer Science*, Elsevier B.V., pp. 73–84.

Barnett, M., Leino, K. R. M. and Schulte, W. [2004]. The Spec# programming system: An overview, *in* G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet and T. Muntean (eds), *CASSIS 2004: Proceedings of the International Workshop on the Construction*

and Analysis of Safe, Secure, and Interoperable Systems, Vol. 3362 of *Lecture Notes in Computer Science*, Springer, pp. 49–69.

Bhat, V., Oarashar, M., Liu, H., Khandekar, M., Kandasamy, N. and Abdelwahed, S. [2006]. Enabling self-managing applications using model-based online control strategies, *ICAC'06: Proceedings of the 3rd International Conference on Autonomic Computing*, IEEE, pp. 15–24.

Boehm, B. W. [1981]. *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, NJ.

Boyapati, C., Lee, R. and Rinard, M. [2002]. Ownership types for safe programming: Preventing data races and deadlocks, *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, pp. 211–230.

Boyland, J. T. and Retert, W. [2005]. Connecting effects and uniqueness with adoption, *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, pp. 283–295.

Breitgand, D., Henis, E. and Shehory, O. [2005]. Automated and adaptive threshold setting: Enabling technology for autonomy and self-management, *ICAC '05: Proceedings of the 2nd International Conference on Autonomic Computing*, pp. 204–215.

Brilliant, S. S., Knight, J. C. and Leveson, N. G. [1990]. Analysis of faults in an n-version software experiment, *IEEE Transactions on Software Engineering* **16**(2): 238–247.

Brooks, F. P. [1975]. *The Mythical Man-Month*, Addison Wesley Longman.

Bruschi, D., Cavallaro, L. and Lanzi, A. [2007]. Diversified process replicae for defeating memory error exploits, *Proceedings of the 3rd International Workshop on Information Assurance*, IEEE.

Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G. and Fox, A. [2004]. Microreboot - a technique for cheap recovery, *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*.

Cengarle, M. V. and Knapp, A. [2005]. Operational Semantics of UML 2.0 Interactions, *Technical Report TUM-I0505*, Technische Universität München.

Chandra, T. D. and Toueg, S. [1996]. Unreliable failure detectors for reliable distributed systems, *Journal of the ACM* **43**(2): 225–267.

Chen, F. and Roşu, G. [2007]. MOP: An efficient and generic runtime verification framework, *OOPSLA '07: Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ACM, pp. 569–588.

Chen, Y., Iyer, S., Liu, X., Milojicic, D. and Sahai, A. [2007]. SLA decomposition: Translating service level objectives to system level thresholds, *ICAC'07: Proceedings of the 4th International Conference on Autonomic Computing*, IEEE.

Ciupa, I., Meyer, B., Oriol, M. and Pretschner, A. [2008]. Finding faults: Manual testing vs. random testing+ vs. user reports, *Technical Report 595*, Department of Computer Science, ETH Zurich, Switzerland.

Clark, T. [2009]. Model based functional testing using pattern directed filmstrips, *ICSE Workshop on Automation of Software Test, AST '09*, IEEE, pp. 53–61.

Clarke, D. G., Noble, J. and Potter, J. M. [2001]. Simple ownership types for object containment, *ECOOP 2001: Proceedings of the European Conference on Object Oriented Programming*, Springer Verlag, pp. 53–76.

Clarke, D. G., Potter, J. M. and Noble, J. [1998]. Ownership types for flexible alias protection, *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ACM, pp. 48–64.

Clarke, D. and Wrigstad, T. [2002]. External uniqueness is unique enough, *Technical Report 048*, Institute of Information and Computing Sciences, Utrecht University.

Cobleigh, R. L., Avrunin, G. S. and Clarke, L. A. [2006]. User guidance for creating precise and accessible property specifications, *FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 208–218.

deGrandis, P. and Valetto, G. [2009]. Elicitation and utilization of application-level utility functions, *ICAC '09: Proceedings of the 6th International Conference on Autonomic Computing*, ACM, New York, NY, USA, pp. 107–116.

Delgado, N., Gates, A. Q. and Roach, S. [2004]. A taxonomy and catalog of runtime software-fault monitoring tools, *IEEE Transactions of Software Engineering* **30**(12): 859–872.

Denaro, G., Gorla, A. and Pezzè, M. [2009]. Datec: Dataflow testing of java classes, *ICSE'09: 31st International Conference on Software Engineering – Companion Volume*, pp. 421–422. Tool demonstration.

Dietl, W. and Müller, P. [2005]. Universes: Lightweight ownership for JML, *Journal of Object Technology (JOT)* **4**(8): 5–32.

Dwyer, M. B., Avrunin, G. S. and Corbett, J. C. [1999]. Patterns in property specifications for finite-state verification, *ICSE 1999: Proceedings of the 1999 International Conference on Software Engineering*, pp. 411–420.

Dzidek, W. J., Briand, L. C. and Labiche, Y. [2005]. Lessons learned from developing a dynamic OCL constraint enforcement tool for java, *Satellite Events at the MoDELS 2005 Conference*, Vol. 3844 of *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, pp. 10–19.

Fischer, M. J., Lynch, N. A. and Paterson, M. S. [1985]. Impossibility of distributed consensus with one faulty process, *Journal of the ACM* **32**(2): 374–382.

Francis, P., Leon, D., Minch, M. and Podgurski, A. [2004]. Tree-based methods for classifying software failures, *ISSRE'04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pp. 451–462.

Froihofer, L., Glos, G., Osrael, J. and Goeschka, K. M. [2007]. Overview and evaluation of constraint validation approaches in Java, *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, IEEE, pp. 313–322.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. [1994]. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional.

Gan, Y., Chechik, M., Nejati, S., Bennett, J., O'Farrell, B. and Waterhouse, J. [2007]. Runtime monitoring of web service conversations, *CASCON '07: Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, ACM, pp. 42–57.

Garlan, D. and Schmerl, B. [2002]. Model-based adaptation for self-healing systems, *WOSS '02: Proceedings of the 1st Workshop on Self-healing Systems*, pp. 27–32.

Ghezzi, C., Jazayeri, M. and Mandrioli, D. [2002]. *Fundamentals of Software Engineering*, 2nd edn, Prentice Hall.

Giorgini, P., Mylopoulos, J. and Sebastiani, R. [2005]. Goal-oriented requirements analysis and reasoning in the Tropos methodology, *Engineering Applications of Artificial Intelligence* **18**(2): 159–171.

Goldstein, M., Shehory, O. and Weinsberg, Y. [2007]. Can self-healing software cope with loitering?, *SOQUA'07: Proceedings of the 4th International Workshop on Software Quality Assurance*, pp. 1–8.

Gorbovitski, M., Rothamel, T., Liu, Y. A. and Stoller, S. D. [2008]. Efficient runtime invariant checking: A framework and case study, *WODA '08: Proceedings of the 2008 International Workshop on Dynamic Analysis*, ACM, pp. 43–49.

Hein, C., Ritter, T. and Wagner, M. [2007]. System monitoring using constraint checking as part of model based system management, *Proceedings of the Second International Workshop on Models@run.time*.

Heo, J. and Abdelzaher, T. [2009]. AdaptGuard: Guarding adaptive systems from instability, *ICAC'09: Proceedings of the International Conference on Autonomic Computing*, ACM Press, pp. 77–86.

Hoare, C. A. R. [1969]. An axiomatic basis for computer programming, *Communincations of the ACM* **12**(10): 576–583.

Horn, P. [2001]. Autonomic computing: IBM's perspective on the state of information technology, *Technical report*, IBM Research.
URL: *http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf*

IEEE [1990]. *IEEE Standard Glossary of Software Engineering Terminology*.

ITU [2004]. *Message Sequence Chart (MSC)*. ITU-T Recommendation Z.120.

Jackson, D. [2000]. Automating first-order relational logic, *FSE 2000: Proceedings of the International Conference on Foundations of Software Engineering*, pp. 130–139.

Jackson, D. [2003]. Object models as heap invariants, *Programming Methodology*, Springer-Verlag New York, pp. 247–268.

Jiang, M., Munawar, M. A., Reidemeister, T. and Ward, P. A. S. [2009]. System monitoring with metric-correlation models: Problems and solutions, *ICAC'09: Proceedings of the International Conference on Autonomic Computing*, ACM Press, pp. 13–22.

JSR127 JavaServer Faces 1.2 Specification, `http://jcp.org/en/jsr/detail?id=127`. JSR 127.

JSR245 Java Server Pages 2.1 Specification, `http://jcp.org/en/jsr/detail?id=245`. JSR 245.

Kapoor, K. and Bowen, J. P. [2007]. Test conditions for fault classes in boolean specifications, *ACM Transactions on Software Engineering and Methodology* **16**(3).

Kazman, R. and Carrière, S. J. [1999]. Playing detective: Reconstructing software architecture from available evidence, *Automated Software Engineering* **6**(2): 107–138.

Kephart, J. O. and Chess, D. M. [2003]. The vision of autonomic computing, *IEEE Computer* **36**(1): 41–50.

Kiviluoma, K., Koskinen, J. and Mikkonen, T. [2006]. Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects, *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ACM, pp. 181–190.

Knapp, A. and Wuttke, J. [2006]. Model checking of UML 2.0 interactions, *Models in Software Engineering*, Vol. 4364 of *Lecture Notes in Computer Science*, pp. 42–51.

König, D. [2007]. *Groovy in Action*, Manning Publications.

Kramer, J. and Magee, J. [2007]. Self-managed systems: an architectural challenge, *FOSE 2007: Proceedings of Future of Software Engineering*, pp. 259–268.

Kuhn, D. R. [1999]. Fault classes and error detection capability of specification-based testing, *Transactions of Software Engineering and Methodology* **8**(4): 411–424.

Laddad, R. [2009]. *AspectJ in Action: Enterprise AOP with Spring*, 2nd edn, Manning Publications.

Lamsweerde, A. v. [2001]. Building formal requirements models for reliable software, *Ada Europe '01: Proceedings of the 6th Ade-Europe International Conference Leuven on Reliable Software Technologies*, Springer-Verlag, London, UK, pp. 1–20.

Leavens, G. T., Baker, A. L. and Ruby, C. [1999]. JML: A notation for detailed design, *in* H. Kiloc, B. Rumpe and I. Simmonds (eds), *Behavioral Specifications of Businesses and Systems*, Kluwer, chapter 12, pp. 175–188.

Lin, P., MacArthur, A. and Leaney, J. [2005]. Defining autonomic computing: A software engineering perspective., *ASWEC 2005: Proceedings of the 16th Australian Software Engineering Conference*, pp. 88–97.

Lorenzoli, D., Mariani, L. and Pezzè, M. [2008]. Automatic generation of software behavioral models, *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, ACM, pp. 501–510.

Luckham, D. C., Kerry, J. J., Augustin, L. M., Vare, J., Bryan, D. and Mann, W. [1995]. Specification and analysis of system architecture using rapide, *IEEE Transactions of Software Engineering* **21**(4): 336–355.

Mariani, L. and Pezzè, M. [2005]. Behavior capture and test: Automated analysis of component integration., *ICECCS 2005: Proceedings of the 10th International Conference on Engineering of Complex Computer Systems*, pp. 292–301.

Meyer, B. [1988]. *Object-Oriented Software Construction*, Prentice Hall.

Morell, L. J. [1990]. A theory of fault-based testing, *IEEE Transactions on Software Engineering* **16**(8): 844–857.

OCL [2005]. *Object Constraint Language*. Available specification formal/06-05-01, downloaded from `www.omg.org` on 2006-12-11.

OMG [2002]. *OMG XML Metadata Interchange (XMI) Specification*.

OMG [2003]. *MDA Guide*. Version 1.0.1,Specification omg/03-06-01, downloaded from `www.omg.org` on 2005-09-29.

OMG [2007]. *OMG Unified Modeling Language Superstructure*. Adopted specification formal/2007-11-02, downloaded from `www.omg.org` on 2008-01-02.

Oreizy, P., Medvidovic, N. and Taylor, R. N. [1998]. Architecture-based runtime software evolution, *ICSE '98: Proceedings of the 20th International Conference on Software Engineering*, IEEE Computer Society, pp. 177–186.

Parnas, D. L. [1976]. On design and development of program families, *IEEE Transactions on Software Engineering* **SE-2**(1): 1–9.

Parnas, D. L. [1979]. Designing software for ease of extension and contraction, *IEEE Transactions on Software Engineering* **SE-5**(2): 128–138.

Pezzè, M. and Young, M. [2007]. *Software Testing and Analysis: Process, Principles, and Techniques*, John Wiley and Sons.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J. and Wang, B. [2003]. Automated support for classifying software failure reports, *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, IEEE Computer Society, pp. 465–475.

Polikarpova, N., Ciupa, I. and Meyer, B. [2009]. A comparative study of programmer-written and automatically inferred contracts, *ISSTA '09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, ACM, pp. 93–104.

Qin, F., Tucek, J., Sundaresan, J. and Zhou, Y. [2005]. Rx: Treating bugs as allergies — a safe method to survive software failures, *SOSP'05: Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pp. 235–248.

Raimondi, F., Skene, J. and Emmerich, W. [2008]. Efficient online monitoring of web-service SLAs, *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, pp. 170–180.

Richardson, D. J. and Thompson, M. C. [1993]. An analysis of test data selection criteria using the relay model of fault detection, *IEEE Transactions of Software Engineering* **19**(6): 533–553.

Rosenblum, D. S. [1995]. A practical approach to programming with assertions, *IEEE Transactions on Software Engineering* **21**(1): 19–31.

Schuler, D., Dallmeier, V. and Zeller, A. [2009]. Efficient mutation testing by checking invariant violations, *ISSTA '09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, ACM, pp. 69–80.

Sommerville, I. [2006]. *Software Engineering*, 8th edn, Addison Wesley.

SRV Java servlet specification. `http://java.sun.com/products/servlet/2.2/javadoc/index.html`.

Stirewalt, K. and Rugaber, S. [2005]. Automated invariant maintenance via OCL compilation., *MODELS 2005: 8th International Conference on Model Driven Engineering Languages and Systems*, pp. 616–632.

Sykes, D., Heaven, W., Magee, J. and Kramer, J. [2008]. From goals to components: a combined approach to self-management, *SEAMS '08: Proceedings of the 2008 International Workshop on Software Engineering forAadaptive and Self-Managing Systems*, ACM, pp. 1–8.

Unkel, C. and Lam, M. S. [2008]. Automatic inference of stationary fields: a generalization of Java's final fields, *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, pp. 183–195.

Voas, J. M. and Miller, K. W. [1994]. Putting assertions in their place, *Proceedings of the 5th International Symposium on Software Reliability Engineering*, pp. 152–157.

Walsh, W. E., Tesauro, G., Kephart, J. O. and Das, R. [2004]. Utility functions in autonomic systems, *ICAC'04: Proceedings of the First International Conference on Autonomic Computing*, IEEE Computer Society, pp. 70–77.

Wang, K. and Shen, W. [2007]. Runtime checking of UML association-related constraints, *Proceedings of the 5th International Workshop on Dynamic Analysis*, IEEE.

Wiggerts, T. [1997]. Using clustering algorithms in legacy systems remodularization, *WCRE'07: Working Conference on Reverse Engineering*, Vol. 0, IEEE Computer Society, p. 33.

Xu, C., Cheung, S. C. and Chan, W. K. [2006]. Incremental consistency checking for pervasive context, *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, ACM, pp. 292–301.

Zibin, Y., Potanin, A., Ali, M., Artzi, S., Kieżun, A. and Ernst, M. D. [2007]. Object and reference immutability using java generics, *ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM, pp. 75–84.

Zimmermann, T., Nagappan, N. and Zeller, A. [2008]. *Predicting Bugs from History*, Springer, chapter 4, pp. 69–88.