# Property Templates and Assertions Supporting Runtime Failure Detection

Jochen Wuttke

University of Lugano
Faculty of Informatics
Technical Report 2008/04
Month Year

**Abstract**

In the context of our research program we addressed the question whether or not requirements documents contain information about system level properties that can be exploited to automatically create assertions for run-time checks of these properties. In this technical report we define the concept of *property template* and report details on the studies we carried out to address this question. The results are presented in the form of a catalog of property templates, and details from the individual studies showing which properties occur in which projects.

## 1 Introduction

Good software engineering practice and thorough verification and validation improve software quality, but do not eliminate all faults. Thus, software systems fail also after deployment. Precise reports of failures in deployed systems have many uses. They may help users to find ad-hoc workarounds, aid developers in locating and fixing the faults causing the failure, and they can drive self-adaptation mechanisms aiming at preventing similar failures in the future.

Our work focuses on self-healing mechanisms for functional software problems. Building on key ideas expressed by Kephart and Chess, most self-healing systems rely on a variant of the "autonomic cycle" [KC03]. In their model, an autonomic element, that is a component or a whole system, is under the control of an autonomic manager, which *monitors* and *analyzes* the execution of the element, and in the case of problems *plans* and *executes* changes to the systems configuration.

In self-healing systems monitoring and analysis aim to identify failures and locate faults. Some self-healing systems rely on execution-independent heuristics instead of explicit failure detection. Examples are micro-reboot techniques and approaches that extend classic garbage collection to deal with memory leaks [CKF+04, GSW07]. These techniques execute repair actions periodically and independently from the kinds of failures that may have occurred, whether they are necessary or not. Thus, they trade precision and computation time

for simplicity, and apply only to some classes of failures. Although execution-independent heuristics can be useful in some contexts, explicit failure detection techniques are an important component of many self-healing systems.

Most research on self-healing systems has addressed issues directly relating to adaptability, and has assumed suitable failure detection mechanisms exist. Here we explicitly address the problem of precise failure detection for self-healing systems. Even though detection of functional failures has been explored extensively in the literature on software validation and verification, in the context of self-healing systems, we must face new challenges.

To be acceptable as a monitoring technique deployed in production systems, automatic failure detectors (1) cannot rely on human operators to arbitrate the validity of detected problems, (2) must have only limited performance overhead, (3) must detect failures precisely and produces only few, if any, false alarms, and (4) must detect failures as early as possible, to enable efficient automatic fault localization.

*Self*-adaptive mechanisms are built to operate transparently for the users, who should not be involved in the diagnosing and fixing process. Thus, failure detection mechanisms cannot rely on partial oracles or on repeated test executions that require human operations. *Self*-adaptive mechanisms work in the field where major performance degrading cannot be tolerated. Thus, the runtime overhead that derives from on-line monitoring must not affect the performance of the systems significantly. Executing repair actions is often computationally expensive. False alarms that trigger adaptation mechanisms may badly affect the performance of the system, and thus should be avoided. Fault damage often tends to amplify and worsen if the software continues executing under the effect of a fault, because the initial impact of faults may propagate and corrupt the system state permanently and extensively. Thus, detecting failures as early as possible enables diagnosing and fixing faults before major damage.

We argue that a complete and consistent set of well tailored assertions can meet the requirements above. Encoding thoroughly analyzed system invariants into assertions produces automated oracles, and removes human operators from the loop. Careful choice of logic constructs in assertions can assure low overhead. Assertions suitably placed in critical locations can detect failures precisely and early enough to support efficient fixing.

In current practice, assertions are either added directly to the code by programmers [Ros95, Das06], or are generated from formal specifications that describe invariants of data-structures and algorithms [Mac00, RAO92]. In both cases getting the specification right is a non-trivial issue, and thus highly error prone [CMOP08, VM94]. Additionally when writing specifications, the developers focus on the implementation details, hence they might miss constraints stemming from the larger context in which the code will be used. Concentrating on code-level specifications also makes it difficult to express constraints that are not directly related to how the system is implemented, but are imposed by domain specific limitations the system has to adhere to.

We address the problem of producing well tailored assertions by defining a technique to map end-user requirements onto code assertions. We provide developers with a catalog of *property templates* that help developers identify constraints that are implicit in the requirement specifications, and create annotated models that can be automatically transformed into code level assertions. We do not require complete formal specifications, usually hard to write and

maintain, but we rely on simple model annotations used to generates proper code assertions. It is a common notion that adaptation in autonomic systems should be driven by the *goals* systems are designed to achieve, and we focus on requirements specifications that reflect those goals [KM07].

For some relevant classes of constraints, developers are relieved from the burden of deriving, tuning, and inserting matching checks for these constraints. As such, the technique improves developer productivity and software quality. Since assertion generation and insertion are fully automated, this process also allows efficient maintenance of constraints even when the systems structure changes, because matching assertions can simply be re-generated and will be inserted, even if new relevant locations have been created by the change.

In this field, fine grained failure detection is often indispensable for precise, effective, and efficient reactions to runtime problems. To enable automatic detection and healing of functional problems, grouping faults and failures occurring in software into clearly separable classes with distinct properties that help identifying them at runtime can be useful.

Assuming the existence of such classes, we define a methodology that targets failures, which commonly manifest at the boundaries between the components that form the system. It employs a model-based specification language that developers use to capture system-level properties extracted from requirements specifications. These properties, reflecting failure classes derived from analysis of real systems, are automatically translated into assertion-like checks and inserted in all relevant locations of the systems code. The custom generated code not only detects failures, but also collects data relevant to support automatic healing actions based on the detected failures.

In this technical report we briefly discuss related work (Sec. 2) and then outline the technique (Sec. 3). We summarize a set of studies we have conducted to obtain clusters of properties from real software projects, and report the results we have obtained (Secs. 4–5). We argue that our results constitute clear evidence that useful property templates exist. The collected information is organized on a per-project basis, and the derived properties are listed in a catalog of requirements-level property descriptions.

## 2 Related Work

The idea to use specifications to derive recurring patterns also appears in the work by Dwyer et al. [DAC99]. They study a large set of finite state specifications from the research literature and academic course-work and collect typical properties appearing in these specifications. Cobleigh et al. build on this work and provide tools to support developers in selecting and applying the derived patterns in their own verification tasks [CAC06]. Our work on identification of properties that can lead to the generation of property templates is similar, in that it also studies software artifacts to derive patterns. However, our study uses two complementary approaches to obtain patterns. Studying not only what the software does wrong, but also relating it to the actual high level specification is an additional internal validation step that is missing from previous work.

There is substantial work at the boundaries between software engineering and programming languages, which addresses problems related to some of the properties we have listed in our catalog. For example, Zibin et al. and Unkel

and Lam address questions regarding immutable objects and variables [ZPA+07, UL08]. Zibin et al. discuss a language extension to Java that allows explicit specification of immutability properties of objects and references, which go far beyond the possibilities of Java's `final` qualifier. Unkel and Lam introduce the notion of *stationary fields*, that is fields where all write operations to the field happen before any read operation. Thus, at read time these variable can be treated similar to `final` fields. The work by Unkel and Lam introduces a static analysis algorithm to identify such fields, but no method to specify them beforehand, which means there is no way of dynamically checking for violations of such a property.

Boyland and Retert address the notion of object *uniqueness* and how languages can enforce the semantics of unique objects [BR05]. The similarity between these approaches and our methodology is that they address similar properties, partially by explicitly specifying them to enable static or dynamic checks, and partially by analyzing real system to determine which properties occur frequently. However, in all cases their analysis and specification happens at the code level, and is thus far from the semantics of the end-user specifications, which we use to identify the properties to monitor.

Chan et al. and Beckman et al. work on static analysis methods dealing with concurrency problems [BBA08, CBS98]. In both cases the basic technique uses annotations in the code to specify additional constraints on classes and methods. Chan et al. provide annotations to explain the behavior of objects or methods, for example *reading* or *writing* certain variables. Beckman et al. specify constraints that declare the allowed accesses to objects, and use the typestate system to declare usage protocols of methods. In both cases, a static analysis method uses these annotations to detect potential or real race conditions and deadlocks.

Work addressing self-healing, that is autonomic repairing of functional problems, usually addresses either transient failures like race conditions through concurrent execution, or resource management problems like memory exhaustion. For example, RX by Qin et al. propose an approach based on checkpoints and dynamic changes of the *environment* of the system, to fix failures due to memory leaks and buffer overflows [QTSZ05]. Goldstein et al. try to delay system crashes due to memory exhaustion by monitoring object activity on the heap and transparently removing unused objects from main memory [GSW07]. This effectively delays and reduces the effects of memory leaks languages with managed memory. From a direction that appears to be more inspired by traditional fault tolerant systems, Candea et al. take the idea of software rejuvenation and introduce *micro-reboot*, a technique that regularly restarts individual components of a system, in the hope that this rejuvenation occurs before faults like memory leaks lead to irreparable damage or system crashes [CKF+04]. Interestingly, none of these techniques actually *heals* the problem in the sense that the fault still exists and failures will eventually occur. This is why these techniques can be reasonably successful without precise failure detection.

Using specifications to automatically derive oracles is one of the objectives of specification- or model-based testing. However, the approaches in that field that attempt to derive oracles usually require a *complete* and *formal* specification [AH00]. Furthermore, most practical approaches require the developer to specify constraints on the level of individual methods or classes, which makes it hard to keep the bigger picture of system level requirements

in mind [BLS05, CL04]. Our property templatebased methodology explicitly addresses this issue by providing a link from the end user requirements all the way down to assertions in the code. The approach only requires a partial, annotated structural model of the parts of the system that should be augmented with failure detectors.

There are several fully or partially automated failure detection approaches that do not require formal specifications, but using capture and replay techniques or dynamic invariant inference to build models of the system. Hangal and Lam use dynamic invariant inference to build a model of system executions [HL02]. Since their goal is complete automation of the model building and monitoring process, their system only issues warnings to be analyzed off line by developers. Approaches the explicitly try to solve the oracle problem usually rely on a separate training phase to learn the model [BGH06, LMP08]. After the learning phase this model remains fixed and serves as the oracle to distinguish between valid and invalid executions. The approaches by Baah et al. and Lorenzoli et al. combine Trace information with static invariants to improve the quality of the models prediction. However, since dynamic behavior inference relies only on the implementation of a system, it is not able to incorporate notions expressed in end-user requirements.

# 3 Mapping Requirements to Assertions

To support self-healing systems, failure detection techniques must have a clear notion of what constitutes a failure, must provide means to detect failures at runtime, and must provide enough information about the failures to allow subsequent analysis to determine the cause of the failures. Even though the first two items seem similar, they have to be treated separately: The first requires an explicit specification of what the system should do. The second requires techniques to monitor the system execution, and means to decide when an execution violates the system specifications.

Our goal is to create high-quality runtime failure detectors for system-level requirements. The purpose of the methodology we developed is to automate the creation of such detectors as much as possible. To do so we have to address two orthogonal concerns: First, we have to address the efficiency and quality concerns associated with runtime monitoring techniques, and second, we have to bridge the semantic gap between system goals, which are the source of the constraints we monitor, and the implementation details of the system. Our technique focuses on two aspects: (1) How to derive and specify system-level constraints, and (2) how to automatically translate constraints into assertions that meet the desired performance and precision requirements.

The central concept in our approach is the notion of property template. A property template is a triple $\langle C, T, R\rangle$ consisting of a constraint $C$, used to annotate, and thus constrain, the behavior of model constructs, a set of assertion templates $T$ that encode how to detect violations of the constraint $C$, and a set of rules $R$, which describe where and how annotations may be placed in the model, and where assertions need to be placed in the code.

Figure 1 shows how our methodology addresses the two key aspects in a process centered around a catalog of property templates. In the first step, developers derive properties from requirements specifications, and annotate the

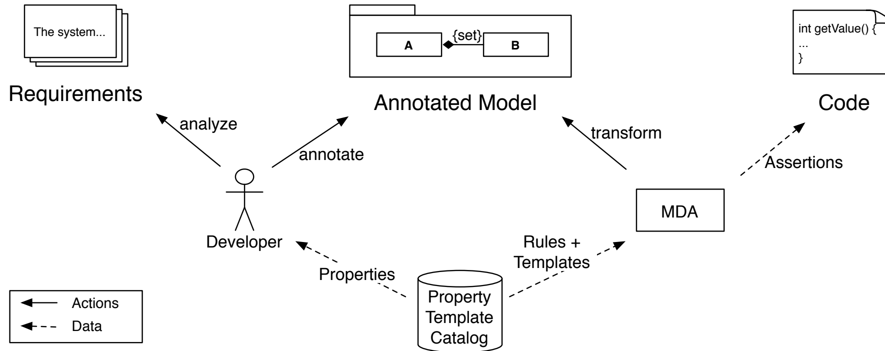


Figure 1: Methodology activities

system design model with constraints that reflect the properties identified at the requirements specification level. This step is difficult to automate, because it requires understanding and analyzing the semantics of requirements specifications, which are usually provided in natural language or informal notations that do not lend themselves well to automatic analysis.

In the second step, a model driven assertion generator transforms the annotated model into assertions at the code level using the rules of the property templates. The assertion generator uses the catalog to generate assertions and the annotated model to identify all relevant locations of assertions in the code.

Both, the concrete implementation of assertions, as well as the rules to determine where the assertions have to be placed, are domain- or even application-dependent. Therefore, the rules provided by the property catalog are designed to be specialized or replaced on demand.

# 4 Research Methodology

To answer the question if there are high-level properties in system requirements that can be traced to typical classes of failures and faults in systems we have to identify candidate properties in requirements on the one hand. On the other hand we have to cluster failures and faults that occur in actual systems into groups exhibiting similar behavior. A thorough analysis can then clarify if one or more of these failure clusters can be mapped to one or more of the properties identified in the requirements. A further step required to facilitate automatic failure detection is to define failure detector templates for each class of failures, so that tools can generate the detectors based on knowledge of where a property associated with the failure class should hold.

We have designed our 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 we studied in the first step. Whenever

clusters can be mapped to one or more property, we have established an important link between the high-level requirements and the code implementing those requirements. A further question that we will be able to answer with the collected data is if and how the type of application influences the failure classes occurring. It seems likely that a highly multi-treaded server application exhibits failure patterns different from those in a computation-oriented library.

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

To our knowledge there does not exist any mining tool able to process bug databases and extract sets of reports based on semantic criteria of the text content. After a brief study of how bugs are reported, and how symptoms and fixes are discussed, we 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. Unfortunately, the large amount of time required for manual analysis limits our ability to do larger studies.

## 4.1 Requirements Analysis

The goal of the requirements analysis during our study was twofold: (1) determine recurring patterns that imply constraints on possible implementation in the specifications, and (2) determine in how many cases an identified pattern or constraint directly relates to a cluster of problem 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. We 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-formed 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. Therefore, it is difficult to describe this process in a way to make it easily reproducible. However, there are some general conclusions that we believe can be drawn from the studies we did.

With respect to pattern 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 specifies 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.

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

### 4.2 Failure Analysis

Because all the software systems we analyzed are still under development the issue databases are constantly changing. To have a stable set of issues to address throughout the time of our study we chose to study issues reported for older versions of each software. This not only gives us a stable set of reports to analyze, but also means that most of the issues have been resolved by the developers, easing our task of determining root cause and fixes.

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. We used the 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 we discarded issues that were marked as *invalid*, *non reproducible*, or as issues that *won't be fixed*. Our rationale for discarding these issues from our statistics is that if the developers do not consider something a bug, then neither should we.

After filtering those non-issues out of the result sets returned by queries to the issue databases, we studied every remaining issue in detail. Questions we had to answer about each issue before we could assign it to a cluster, create a new cluster, or discard the issue are:

- 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 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 issue are clustered. The third question connects the failure clusters to the results from the requirements analysis: an issue for which we cannot identify a clear requirement that is violated will not help us in defining property templates. The numbers of relevant issues reported in section 5 reflect this last filtering step.

## 5 Studies and Results

In this section 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 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 Section 6.

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

Table 1: Applications studied.

The clustering details, like which issues fall into which category, can be found in the appendix.

## 5.1 Cocoon

The requirements analysis for the Cocoon core component focused on the version 2.2 of the APIs of the core packages[2]. 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-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[3], 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 2 shows the accumulated occurrences of properties across all sub-components of Cocoon Core. 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.

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

Table 2: Properties extracted from the Cocoon API.

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`.

[2] `http://cocoon.apache.org/2.2/core-modules/`
[3] `http://excalibur.apache.org/`

The settings for our query are[4]:

| 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 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.

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

Table 3: Cocoon Issue Clusters

## 5.2 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 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 |

[4] “Released Versions” include version 2.2, which we used for the requirements analysis.

As the data in the summary table (Tab. 4) 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.

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

Table 4: Lucene Issue Clusters

## 5.3 Tomcat

We focused our requirements analysis for Tomcat on the API specifications for Java Servlets and Java Server Pages [JSRa, JSRb]. 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 5 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.

| **Property** | **Number** |
|---|---|
| Total instances | 14 |
| `comparable` | 1 |
| `concurrency` | 1 |
| `immutable` | 3 |
| `initialized` | 4 |
| `language` | 2 |
| `unique` | 3 |

Table 5: Properties extracted from Tomcat requirements

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 components 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.

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

Table 6: Tomat Issue Clusters

### 5.4 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 7 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 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.

## 6 Property Templates

The goal of our research program is to create high-quality runtime failure detectors for system-level requirements. For this purpose we developed a methodology to automate the creation of such detectors as much as possible. To achieve good quality for the detectors and full automation, we have to address two orthogonal concerns: First, we have to bridge the semantic gap between system

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

Table 7: 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.

goals, which are the source of the constraints we want to monitor, and the implementation details of the system. Second, we have to address the efficiency and quality concerns associated with runtime monitoring techniques. This report only discusses results obtained towards the first goal, the second is left for future research.

The central concept in our approach is the notion of property template. A property template is a triple $(C, T, R)$ consisting of a constraint $C$, used to annotate, and thus constrain, the behavior of model constructs, an assertion template $T$ that encodes how to detect violations of the constraint $C$, and a set of rules $R$, which describe where and how annotations may be placed in the model, and where assertions need to be placed in the code.

The following paragraphs list the properties we have identified through our studies. The rules of the property template have to be derived with the goals of efficiency and precision of detection in mind, and thus are left for future research.

Property: `caching`
Description: Cached entities must follow a correct caching protocol for storage, retrieval, and expiration.

Property: `concurrency`
Description Groups concurrency related problems like deadlocks and data corruption through insufficient locking.

Property: `language <L>`
Description: Covers problems with string parsing and formatting according to some grammar or regular expression defining language `L`.

Property: `comparable <C>`
Description: Annotated element must provide a customized comparison operation matching interface `C`.

Property: `immutable`
Description: Instances of annotated classes may not change their visible state after creation or an explicit locking operation.

| | |
|---|---|
| Property: | `initialized` |
| Description: | `initialized` implies that a reference is not `null` and the entity specific initialization has been completed. |

| | |
|---|---|
| Property: | `resource mgmt` |
| Description: | Covers all sorts of resource management and exhaustion problems. |
| Property: | `unique` |
| Description: | A constrained entity must be unique within its context. If the constrained entity is a relation, tuples in the relation must be unique. |

## 7 Summary and Future Work

We introduce and define the notion of *property template*. Property templates are the unifying concept that links system level requirements specifications to constraints on the systems implementation that can be checked automatically at runtime. In this technical report we present the data obtained through our studies and a preliminary analysis, which supports our hypothesis that property templates exist.

The data from the requirements analysis shows that there are a number of easily identifiable pattern relating to constraints on the implementation. The data from the failure analysis shows that a substantial fraction of reported faults falls into clearly identifiable clusters with distinct properties. We report a set of requirements properties that are connected to failure clusters. These *property templates* lend themselves to the derivation of assertions for runtime failure detection, due to their strong connection between requirements and implementation.

There are indications that the type of application has an influence of what classes occur, and how many faults in total fall in these classes. These observations provide a foundation for further explorations into techniques to push forward runtime failure detection.

The results reported here are promising, but are not based on a wide enough range of applications to allow generalization of the conclusions. Furthermore, the diversity of professionalism in documenting requirements and handling failure report in each project makes it difficult to generalize from the data we have. For future work, we will extend our study by analyzing more applications to corroborate the evidence we already have with respect to property templates. Having studied many more applications will also allow us to draw more precise conclusions on the effect of application types on the evident property templates.

We are currently defining an engineering methodology centered around the property templates we identified through our studies. The goal is to provide a complete technique, starting from requirements analysis by developers, through model annotation, to automatic generation of failure detectors. Property templates provide the integrating link between these steps. They tell developers what properties are known to often occur in their applications and how to identify them. The model annotation and code generation frameworks then use the assertion templates from the catalog to generate assertions tailored to the deployment platform of the application. Sufficient flexibility of the methodology

is provided by the standard extension mechanisms built into the Model Driven Architecture.

To evaluate how useful the identified property templates are, we have to experiment with applications and know faults, determine how easily the properties can be identified and specified based on the applications specifications, and then asses how well the generated assertions capture failures at runtime.

# References


[AH00] Sergio Antoy and Richard Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, 2000.

[BBA08] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks with typestate. In *Proceedings of OOPSLA 2008*, 2008. To appear.

[BGH06] George K. Baah, Alexander Gray, and Mary Jean Harrold. On-line anomaly detection of deployed software: a statistical machine learning approach. In *Proc. 3rd Int. WS on Software Quality Assurance, SOQUA '06*, pages 70–77. ACM, 2006.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proc. Int. WS Construction and Analysis of Safe, Secure, and Interoperable Systems, CASSIS 2005*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[BR05] John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. *SIGPLAN Not.*, 40(1):283–295, 2005.

[CAC06] Rachel L. Cobleigh, George S. Avrunin, and Lori A. Clarke. User guidance for creating precise and accessible property specifications. In *Proc. 14th Int. Symp. on Foundations of SW Eng., SIGSOFT '06/FSE-14*, pages 208–218, 2006.

[CBS98] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proc. Int. Conf. on SW Eng., ICSE'98*, 1998.

[CKF+04] George Candea, Shinishi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot - a technique for cheap recovery. In *Proc. 6th Symp. on OS Design and Impl.*, 2004.

[CL04] Yoonsik Cheon and Gary T. Leavens. The JML and JUnit way of unit testing and its implementation. Technical Report TR #04-02, Department of Computer Science – Iowa State University, 2004.

[CMOP08] Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, and Alexander Pretschner. Finding faults: Manual testing vs. random testing+ vs. user reports. Technical Report 595, Department of Computer Science, ETH Zurich, Switzerland, 2008.

[DAC99] Matthew B. Dwyer, Geoerge S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. Int. Conf. on SW Eng., ICSE*, pages 411–420, 1999.

[Das06] Manuvir Das. Formal specifications on industrial-strength code – from myth to reality. In *Proc. 18th Int. Conf. Computer Aided Verification, CAV*, 2006. Invited Talk.

[GSW07] Maayan Goldstein, Onn Shehory, and Yaron Weinsberg. Can self-healing software cope with loitering? In *Proc. 4th Int. WS on SW Quality Assurance, SoQUA'07*, pages 1–8, 2007.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. 24th Int. Conf. on SW Eng., ICSE '02*, pages 291–301. ACM, 2002.

[JSRa] Java Servlet 2.5 Specification. `http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html`. JSR 154.

[JSRb] Java Server Pages 2.1 Specification. `http://jcp.org/aboutJava/communityprocess/final/jsr245/index.html`. JSR 245.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Proceedings of Future of Software Engineering, FOSE 2007*, 2007.

[LMP08] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proc. 30th Int. Conf. on SW Eng., ICSE '08*, pages 501–510. ACM, 2008.

[Mac00] Patrícia D. L. Machado. *Testing from Structured Algebraic Specifications: The Oracle Problem.* PhD thesis, University of Edinburgh, 2000.

[QTSZ05] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies — a safe method to survive software failures. In *Proc. 20th Symp. on OS Principles, SOSP'05*, pages 235–248, 2005.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proc. 14th Int. Conf. on SW Eng., ICSE '92*, pages 105–118, 1992.

[Ros95] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[UL08] Christopher Unkel and Monica S. Lam. Automatic inference of stationary fields: a generalization of java's final fields. In *Proc. 35th Symp. on Principles of Prog. Lang., POPL '08*, pages 183–195. ACM, 2008.

[VM94] Jeffrey M. Voas and Keith W. Miller. Putting assertions in their place. In *Proceedings of the 5th International Symposium on Software Reliability Engineering*, pages 152–157, 1994.

[ZPA+07] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and reference immutability using java generics. In *Proc. 6th European SW Eng. Conf. and the Symp. on the Foundations of SW Eng., ESEC-FSE '07*, pages 75–84. ACM, 2007.

# Appendix

## Cocoon Details

| 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 8: Classes or interfaces of each identified property in the Cocoon API. o.a.c refers to the common top-level pacakge org.apache.cocoon. (I) marks interfaces, and (A) marks abstract classes.

## Tomcat Details

| 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 9: Classes and documentation sections of each identified property. JSP refers to the Java Server Pages specification, SRV to the Java Servlet specification.

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

Table 10: Mapping between properties and bug report numbers for Tomcat.