Defining Model Transformations for Property Templates

Jochen Wuttke

University of Lugano
Faculty of Informatics
Technical Report 2009/05
August 2009

Abstract

We defined the notion of property templates after we observed that many system failures can be clustered according to their symptoms, and the design constraints these failures violate [PW09]. In this technical report we document the semantics of property templates, and how we implemented these semantics in AspectJ code templates that support assertion generation for Java programs.

1 Introduction

In previous work we introduced the idea of property templates and outlined a methodology that uses property templates to facilitate automatic translation of design-level properties into code-level assertions [PW09]. Property templates encode best practice assertions similar to design patterns, which encode best practices for structural design problems. Defining correct and complete assertions is not trivial. Defining assertions for design-level properties is particularly difficult, because such properties often affect large parts of the code, and hence require a holistic view of the system to be able to decide which assertions are needed, and where they are needed. Providing developers with property templates solves this problem, and automatic code generation and update also make the maintenance of assertions easier when the system evolves.

The foundation of the automated translation are formal semantics for the properties captured in property templates, and code templates that encapsulate the necessary assertions and the rules for where to place assertions in the code. In this technical report, we document the semantics of a selected set of property templates, outline the research process we followed to define the code templates, and document the code templates implemented in LuMiNous. The development of both, the formal semantics and the concrete implementation of assertions, has been guided by several considerations and trade-offs.

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.

The development of property templates and LuMiNous is situated in the scope of a larger project. Our group aims at developing techniques that enable self-healing systems to recover from software failures. *Self*-healing 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 participation. Additionally, 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 previously argued that a consistent set of well tailored assertions can meet the requirements above, and yet be precise and effective enough to avoid negative impacts on the running software, and that current practice can only give insufficient assurance of the quality of assertions [PW09]. This is because 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. Our property template-based approach tackles this problem by encoding best practices developed by expert programmers into templates for assertions. As such, property templates are an abstract problem solving concept similar to design patterns [GHJV94].

In this technical report we briefly discuss related work in Section 2 and then outline the technique and prototype implementation in Section 3. We document the formal semantics of property templates in Section 4, and the code templates developed for them in Section 5. Section 6 summarizes and overviews ongoing and future work.

2 Related Work

Runtime monitoring covers a wide range of applications, for example monitoring service-level agreements or structural and architectural constraints of systems. Using models and constraint patterns to derive oracles or validate systems occurs in

different contexts. Froihofer et al. provide an overview and a taxonomy of constraint evaluation approaches, focused on Java [FGOG07]. Wang and Shen describe an approach to detect violations of constraints intrinsic in UML class diagrams [WS07]. 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. translate explicitly given OCL constraints into aspects that encode assertions matching the OCL constraints [DBL05]. Stirewalt and Rugaber present an approach to *enforce* OCL invariants at runtime [SR05]. They use specified invariants to generate wrappers around the classes that are referred to in the invariants. These wrappers notify all classes involved in the constraints about changes in values, and suitably update the related objects. Raimondi et al. 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 [RSE08]. They translate typical patterns into timed automata and execute the automata together with the services to determine if events violate the SLA. Clark proposes a diagrammatic approach to express heap invariants, and suggests that diagrams can serve as the input for validation and verification [Cla09]. Clark's work indicates interesting research directions, but is in a preliminary stage, and is not yet supported by evidence.

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 used during development. The approach by Raimondi et al. shows how extensions to standard modeling languages can be used for application or task specific specification and monitoring. The work makes strong assumptions about the checked properties, which effectively limits its applicability to timing constraints. The monitoring approaches closest to ours 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 our approach 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 we focus on design constraints that are not expressible in OCL.

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 template-based 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 The LuMiNous Prototype

To assess how well property templates address the challenges outlined in the introduction, we developed LuMiNous, a prototype tool that uses property templates based on the formal semantics defined in the previous section to generate runtime checkers for Java programs. LuMiNous facilitates the *specification* of the design-level properties that have to be monitored at run-time and *automatically generates* the necessary code-level assertions.

For the LuMiNous prototype we made some design decisions about third-party tools and target platforms. As the modeling and transformation framework we chose techniques associated with the Model-Driven Architecture (MDA) [MDA03]. Models are written in UML, augmented with a profile for property annotations, and the assertion generation is implemented as a JET[1] model transformation. The LuMiNous model transformation generates AspectJ code that combines both assertions and placement rules targeting the Java platform.

We chose to use profiles instead of the Object Constraint Language (OCL) to express the property constraints, because our properties are very hard or even impossible to express in OCL. 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 means it cannot iterate over all attributes, we have to formulate this post-condition explicitly listing 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, with our property

[1] `http://www.eclipse.org/modeling/m2t/?project=jet`

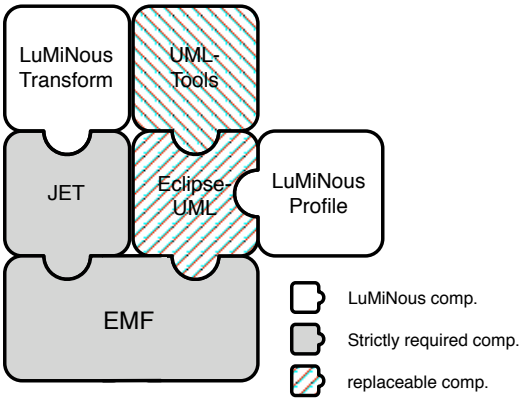

Figure 1: The LuMiNous architecture.

templates, either the transformation or the generated assertion can iterate over all attributes implicitly and are thus robust against model evolution.

The prototype is implemented as two Eclipse[2] plugins, one containing the UML profile representing the annotation language and one for the model transformation. Figure 1 shows the architecture of the LuMiNous prototype, including the prerequisite components for it to work. 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[3] to create and annotate system models. We implemented the model transformation using Java Emitter Templates (JET), and we compile and deploy assertions with the AspectJ compiler and weaver.

Developers use the LuMiNous prototype following three basic steps: (1) annotating a design model using the UML profile provided by LuMiNous, (2) running the model transformation to generate assertion code, and (3) compiling the assertions into the application before running the application.

Step 1: Developers annotate design models with their favorite UML design tool that supports profiles. The LuMiNous profile defines stereotypes that can annotate classifiers, methods, parameters, and attributes of classes. Each property in Table 2 corresponds to a stereotype in the profile. Some of the stereotypes accept additional parameters to specify the annotation semantics more precisely. Notice that with LuMiNous we did not intend to develop a new assertion language. Rather we provide an easy way to place pre-defined constraints in models. Hence the language we provide offers only simple declarative means to express these constraints. To further reduce the requirements for using the approach, LuMiNous works with partial models of the system and does not require complete or formal specifications.

Step 2: The second step is fully automated by the LuMiNous prototype. LuMiNous parses the annotated model and instantiates a set of concrete assertions for every

[2] `http://www.eclipse.org`

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

property annotation by using the assertion templates and deployment rules implemented in the transformation. Assertions and their respective deployment locations are encoded in aspects.

Step 3: The third step is semi-automatic and depends on deployment decisions developers have to make. By using AspectJ to compile and deploy our assertions, we can exploit the powerful static analysis of AspectJ to weave the aspects into all relevant locations in the target application. Using AspectJ has two further technical advantages. First, AspectJ's ability to weave aspects into binary code allows us to instrument not only applications for which we have source code, but also purely binary distributions. Second, generated assertions can be woven into the application using either of the weaving techniques provided by AspectJ. Static weaving optimizes the startup time of the instrumented system, while load-time weaving does not change the target application and weaves assertions on demand.

4 Classes of Runtime Failures

In our research to develop reliable automatic failure detectors for self-healing software systems, we have studied runtime failures as opposed to failures observed during unit and system testing. Doing this, we have observed that many runtime failures can be clustered into relatively few groups related to violated design-level properties. This observation led us to hypothesize that the commonalities between failures within a cluster can be exploited to automatically generate detectors for the associated failures.

To reach a better understanding of this hypothesis' impact, we carried out several explorative studies on medium and large software systems (Section 4.1). The results of these studies confirm our hypothesis about the existence of clusters of failures associated with design-level properties. Towards the goal of automating the creation of runtime checks for such design-level properties we have to formally define what constitutes a design-level property, what assertions can detect violations, and where such assertions have to be evaluated.

In this chapter we first discuss the research methodology we applied to identify candidate properties, and then present a definition of the formal semantics of these properties.

4.1 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[4] 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 is no mining tool that is 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.

4.1.1 Requirements Analysis

The goal of the requirements analysis during our study was twofold: (1) determine recurring patterns that imply constraints on possible implementations in the specifications, and (2) determine in how many cases an identified pattern or constraint directly relates to a cluster 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. 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

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

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.

4.1.2 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 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 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`.

Table 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.

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	–

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 we cannot identify a clear requirement that is violated will not help us in defining property templates. The numbers of relevant issues reported in Table 1 reflect this last filtering step.

The more detailed analysis of the classes listed in Table 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 4.3 lists semantics and templates only for those properties that are general and not subsumed by others.

4.2 Definitions

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. We also discuss 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.

Throughout the discussion we make use of several conventions to simplify the notation:

(1) We 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) We 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, we 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 we are discussing in the context of this catalog. However, usually 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 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 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.

To precisely define the semantics of the properties in the catalog, and the additional constraints implied (for example observability constraints), we require 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.

Property	Description
`explicit<I>`	A constrained class must implement a comparison operation matching interface `I`.
`immutable`	A constrained entity may not change its visible state once it is created.
`initialized`	A 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`.
`unique`	A constrained entity must be unique within its context. If the constrained entity is a relation, tuples in the relation must be unique.

Table 2: Classes of constraints for property templates.

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, classifiers, 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, classifier, 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. Our definitions and terminology for program locations is a generalized form of the terminology common in aspect-oriented languages, such as AspectJ [AJ09, Lad09].

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

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 3: Catalog entry for `comparable`.

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.

The syntax for field access sites is :

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

where *set* and *get* refer to write and read access respectively. T is a type name, f is a field 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.

Definition 4.9 (Execution Event)**.** When stuff is executed.

4.3 Property Catalog

`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

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

Table 4: Catalog entry for `explicit`.

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 **??**), 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 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 [GHJV94]. 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`

The `immutable` property declares that instances of annotated classifiers must not change their visible state after their creation.

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.`m(..)`$\{m \in C.methods \setminus \{$`new(..)`$\} : \forall a \in C.attributes : a@pre_m = a@post_m\}$

Table 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 6: Catalog entry for `initialized`.

`initialized`

An `initialized` annotation on a classifier implies that classifier specific initialization has completed before any references to instances of the classifier are used.

The semantics for 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 [App08] and Groovy [KÖ07], even generate accessor methods automatically if developers do not specify them explicitly, hence making direct field access unnecessary.

To explicitly declare that calling a method results in classifier specific initialization the method can be specified with the `initializer` parameter to the annotation. If we are dealing with static singletons the semantics of `initialized` remain the same conceptually, but since static methods cannot refer to instance objects the tracking is implemented slightly differently.

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 7: Catalog entry for `language <L>`.

`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 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.

`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 a navigable association declares that instances must be unique within the context of the annotated association relation. Uniqueness of an instance is determined by an 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[5].

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 the represent a 1-to-n or a n-to-m relationship. Instances participating in 1-to-1 relationships

[5]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.

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(..)`$\wedge$ $\lvert\{x \in heap@post_{\texttt{new}} \lvert x$.`uniquenessProperty` $=$ o.`uniquenessProperty`$\}\rvert \leq 1\}$
Assertions:	per-relation unique objects A.`add(..,o,..)`$\{\lvert\{x \in A@post_{\texttt{add}} \lvert x$.`uniquenessProperty` $=$ o.`uniquenessProperty` $\wedge\, x\}\rvert \leq 1\}$
Assertions:	attribute change $C.*(..)\{$`uniquenessProperty`$@pre \neq$ `uniquenessProperty`$@post \rightarrow$ $\lvert\{x \in A \lvert x$.`uniquenessProperty` $= o$.`uniquenessProperty`$\}\rvert \leq 1\}$

Table 8: Catalog entry for `unique`.

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. Furthermore, in collections that allow `null` values, we ignore them in the comparison. This is necessary to avoid unnecessary exceptions for example in arrays that may be initialized with `null` values.

5 Template Implementation

The second contribution we discuss in this report is the definition of effective and efficient oracles that can detect violations of the properties defining the failure classes of the previous chapter. From a high-level point of view, the 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 UML specifications to code, impact on how the high-level assertion have to be realized, and how and where they have to be placed in the system code.

In this section we discuss the challenges of defining efficient and effective oracles for property templates. The effectiveness of oracles is impacted strongly 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 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 assertion, 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. Most importantly, this means that we must make assumptions about idioms and design patterns used for translating UML models to code. The simplest such assumption is that names of classes and associations in models match the names of classes and collection variables in the implementation. If these assumptions do not hold, a fully automatic mapping from model annotations to assertions is impossible.

To obtain an implementation of property templates that not only matches the formal definitions in Section 4, but also meets the non-functional requirements of performance and maintainability discussed above we followed these steps:

1. For every property, select an application with a known fault that leads to violations of that property.
2. Manually implement assertions that successfully detect the violation.
3. If necessary, decide on trade-offs between performance and precision.
4. Abstract the concrete implementation into a template that can be instantiated automatically from model annotations.

Early results in our ongoing work clearly indicate that the aspects developed this way work well also with other applications and can reveal failures due to previously unknown faults.

5.1 Templates

The implementation we discuss here refers to the transformation from models to AspectJ code. Hence, the templates contain a lot of platform specific information. The templates shown contain placeholders of the form `<Placeholder>`. The transformation replaces these with the concrete model elements relevant for each annotation in the model. Table 9 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 joinpoint attaches the assertion check after static initialization of loaded classes completed. This is typically the case before any instance 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.

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 9: Template placesholders and their expansions

```
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 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 2: Code template for `explicit<I>`

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

Listing 3: Code template for `immutable`

`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.

`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.

```
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 4: Code template for `initialized`

`initialized`

The code for `initialized` (Listing 4) is a straightforward realization of the formal definition. The `initMethod` pointcut captures all calls to the initializer method, which is either explicitly specified with the template parameter `initializer`, or is any constructor of the annotated classifier. A call to the `initializer` sets the initialization flag to *true*, so that consecutive calls to other methods, captured by the `checkedMethods` pointcut, can proceed.

`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 2 shows. Transforming this model generates two instances of the aspect template shown in Listing 5, one for each

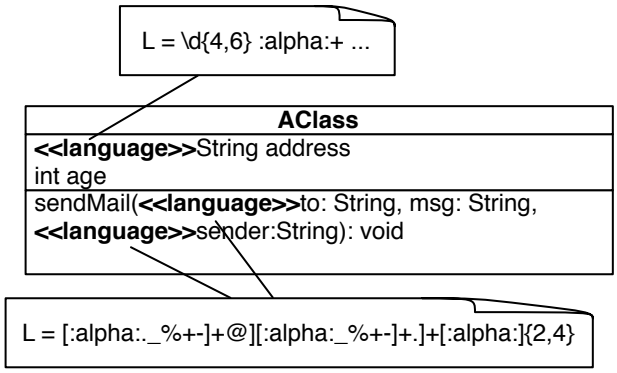

Figure 2: Multiple applications of `language` to the same classifier

```
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: 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 6: Code template for `language<L>` on class attributes

annotated parameter (`to` and `sender`), and on instance of the aspect template shown in Listing 6 for the `address` field.

The aspect for method parameters captures all method call to methods that have an annotated parameter and checks the value before the method is executed. For classifier attributes, only assignments are captured, since initially the value of any String attribute is `null` and hence satisfies the constraint.

`unique`

The templates for `unique` are fully implemented as documented in Section 4.3.

The code in Listing 7 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 **??**, 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 8 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 observant reader will also notice that with global uniqueness, we also track changes to attribute values of relevant objects, while we do not do this with association uniqueness. The reason for this is that developers have the responsibility to maintain and update their collections when relevant attributes of their collected classes change. For example, many collections in that Java SDK use `hashCode` to identify individual objects. If an object is already inside a container and then its attributes change such that its `hashCode` changes, it is the developers responsibility to update the collection. These updates are captured by the aspect as it is written.

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. This is also expressed in the observability requirement in the specification.

In practice, the global uniqueness variant where the `unique` annotation is placed on a classifier is less useful, because it tends to generate many 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

```
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 7: Code template for an association annotated with `unique`

```
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 8: Code template for unique classifier

generated aspects (see MyFaces example and DaTeC study).

- Discuss trade-offs and design decisions.
- Threats to validity.
- Justify choices and discuss tool support

6 Summary and Future Work

We introduced and defined the notion of *property template* in a previous paper [PW09]. In this technical report we document the formal semantics of property templates and their realization as AspectJ code templates in our prototype tool LuMiNous.

To investigate the usefulness of property templates for capturing design-level properties and to generate effective and efficient code-level assertions to monitor them, we used our prototype tool LuMiNous in several case studies. We assessed the generated assertions' ability to detect failures cause by previously unknown faults, and the assertion robustness with respect to changes in the applications. The results we obtained indicate that the semantics and templates we documented here are effective in detecting specific types of runtime problems.

In our ongoing work we are evaluating how well the assertions generated from the templates can capture previously unknown failures, how to extend the set of properties, and how to broaden the scope in which property templates can be applied. Preliminary results indicate that property templates are a powerful abstraction that makes it easy for developers to include runtime checks for design-level properties into their applications.

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.

[AJ09] 2009. `http://www.eclipse.org/aspectj/`.

[App08] Apple Inc. *The Objective-C 2.0 Programming Language*, 2008.

[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.

[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.

[Cla09] Tony Clark. Model based functional testing using pattern directed filmstrips. In *ICSE Workshop on Automation of Software Test, AST '09*, pages 53–61. IEEE, 2009.

[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.

[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.

[DBL05] Wojciech J. Dzidek, Lionel C. Briand, and Yvan Labiche. Lessons learned from developing a dynamic ocl constraint enforcement tool for java. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 10–19. Springer Berlin/Heidelberg, 2005.

[FGOG07] Lorenz Froihofer, Gerhard Glos, Johannes Osrael, and Karl M. Goeschka. Overview and evaluation of constraint validation approaches in java. In *Proceedings of the 29th International Conference on Software Engineering, ICSE'07*, pages 313–322. IEEE, 2007.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[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.

[KÖ7] Dierk König. *Groovy in Action*. Manning Publications, 2007.

[Lad09] Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring*. Manning Publications, 2nd edition, 2009.

[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.

[MDA03] The Object Management Group. *MDA Guide*, version 1.0.1 edition, September 2003. Specification omg/03-06-01, downloaded from `www.omg.org` on 2005-09-29.

[PW09] Mauro Pezzé and Jochen Wuttke. Automatic generation of runtime failure detectors from property templates. In Betty H. C. Cheng, Rogerio de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 229–264. Springer Verlag, 2009.

[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.

[RSE08] Franco Raimondi, James Skene, and Wolfgang Emmerich. Efficient online monitoring of web-service SLAs. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 170–180. ACM, 2008.

[SR05] Kurt Stirewalt and Spencer Rugaber. Automated invariant maintenance via OCL compilation. In *8th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2005*, pages 616–632, 2005.

[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.

[WS07] Kun Wang and Wuwei Shen. Runtime checking of UML association-related constraints. In *Proceedings of the 5th International Workshop on Dynamic Analysis*. IEEE, 2007.