

An Approach to Detecting Failures Automatically

Jochen Wuttke
University of Lugano
Via G. Buffi 13
6904 Lugano, Switzerland
wuttkej@lu.unisi.ch

ABSTRACT

Failure detection is a difficult and often expensive task. The principle of self-healing addresses this cost issue, but poses new research questions. This work focuses on detecting non-trivial failures that are hard to specify and difficult to detect within the context of self-healing software. Typical failures in this class would be problems arising at the component integration level. In this paper we discuss general requirements for failure detection in self-healing software, propose an approach to automatically map system level specifications to run-time checkable code-level assertions, and illustrate our technique through an example.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications; D.2.4 [Software Engineering]: Software/Program Verification—*Programming by contract*

General Terms

design, reliability

Keywords

failure detection, self-healing, model driven engineering

1. INTRODUCTION

Detecting failures is difficult and usually labor intensive. Especially when the failures are not catastrophic, like system crashes, but more subtle, like unexpected return values, or states inconsistent with the executed actions. Designing detection mechanisms for such failures is complex and requires considerable engineering effort. Many failure detection mechanisms are not designed to work fully automated, but when they raise warnings, system operators have to decide whether these warnings signal real problems, or whether they are spurious.

In this paper we discuss an approach to the design of fully automated failure detection mechanisms. The overarching

scheme of our discussion is the concept of *self-healing* software, that is software systems that can detect and fix problems automatically. Developing principles and techniques that enable the creation of fully automatic failure detectors is an important step towards the realization of such systems. Even though all the mechanisms discussed here can in principle be applied to any failure detection scenario, the goal of fully automatic detection and fixing of failures poses stricter requirements on the employed mechanisms.

In this paper, we focus on failures that arise at the component integration level. These failures are usually hard to detect, because they do not cause catastrophic events such as system crashes, but lead to invalid states.

To facilitate the development of efficient and effective failure detectors for self-healing systems, we propose a partially automated approach to derive these mechanisms from high-level system specifications. The goal of this approach is to simplify the creation of detectors for well-defined classes of failures, based on a model driven methodology.

The main contributions of this paper are:

- An approach to automatically map system level specifications to run-time checkable, code-level assertions.
- A research agenda aiming to improve failure detection in the context of self-healing software.
- An example to illustrate how the approach can be applied and to highlight open research questions.

The rest of this paper is structured as follows: Section 2 discusses some recent applications of simple failure detection mechanisms, shows the advantages of more detailed detection mechanisms as the ones proposed in this paper, and outlines the additional requirements imposed by self-healing. Section 3 describes our approach in detail. Section 4 gives a complete example and shows how to apply the technique to a simplified software system. Section 5 outlines a research agenda that addresses the open issues in failure detection not targeted by this paper. Section 6 discusses and compares our approach to related work. Section 7 concludes with an outlook on concrete plans for future research.

2. SELF-HEALING SOFTWARE

Self-healing systems rely on four main phases in order to react to adverse conditions in their runtime environment: failure detection, fault diagnosis, fault healing, and verification of healing actions. *Failure detection* mechanisms reveal conditions that violate correctness assumptions about

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOQUA '07, September 3-4, 2007, Dubrovnik, Croatia
Copyright 2007 ACM ISBN 978-1-59593-724-7/07/09 ...\$5.00.

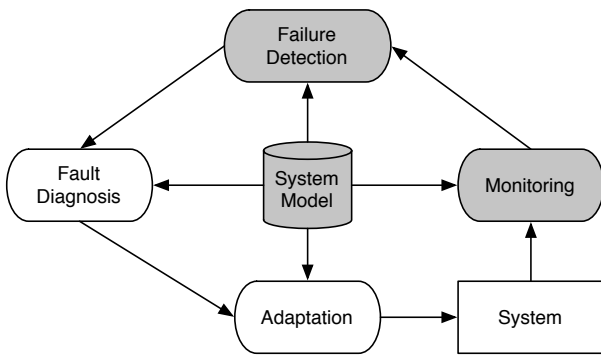


Figure 1: Feedback and information flow in self-managed systems. Adapted from [13].

the execution of the program, usually expressed with constraints. *Fault diagnosis* analyzes detected failures to find the parts of the system that are responsible for those failures. In the *adaptation* phase, the system decides which changes should be applied to the system to fix the detected problem. *Verification* of healing actions makes sure that the measures adopted to overcome the problems do not cause other problems. The verification phase is often implicit: *monitoring* the execution of the system often substitutes for explicit checks in validating the changes. Figure 1 shows how the different phases relate to each other. The *system model* contains information useful in the different phases. This information may be static in the sense that the structural patterns and other constraints on the system do not change at runtime, or it may contain dynamic elements that can change, for example due to learned optimizations.

So far most work on self-managing systems uses only very simple failure models, for example, exceeding thresholds for the response time. Such models are sufficient to monitor high-level, non-functional properties such as performance requirements, structural integrity, and resource availability constraints. But they do not address subtle errors that are hard to specify and detect, such as typical integration problems between components or services.

Current and past research in self-managed systems has focused on a combination of fault diagnosis and the development of adaptation strategies. For example, Gao and Kermani [7] describe an approach to diagnose faults in transaction based systems and provide an in depth analysis of the computational complexity involved in optimally locating faults. Even though they use the term *failure detection* for their mechanism, its purpose is to determine the root causes of failed transactions, and as such better matches our use of the term *fault diagnosis*. Littman et al. [16] show how reinforcement learning can be used to dynamically optimize healing strategies. These optimizations work under the assumption that failure conditions and faults are easy to determine, and different methods to fix a problem have different costs associated with them. Breitgand et al. [2] describe a system architecture for adaptive performance management. These examples detect failures in the form of failed transactions, interrupted network connections, and violated performance constraints. All those types of failures are simple to detect and have root causes outside the system.

Kramer and Magee [15], and previously Georgiadis et al.

[10] discuss the main research challenges in the area of self-managed systems as problems on the architectural level. Previous work by Garlan and Schmerl [9] addresses self-management problems on this level. They use architectural styles to describe constraints within which the architecture of a dynamically changing system must be maintained. The architectural style describes the valid system compositions, and a runtime monitor checks if the style is violated. This research is closer to our approach than other work discussed above, because its failure model looks for failures stemming from internal causes, but the failure types are straightforward to specify and detect. Though not a complete survey on the research in self-managing systems, the above examples are representative samples of the problems that the community has targeted so far.

Revealing failures caused by integration faults is traditionally the domain of software testing. The testing literature provides a wide range of techniques to test software, including techniques for testing for integration faults. However, traditional testing techniques are only of limited use for self-managing systems. First and foremost, during software testing a dedicated test driver sets up and executes the system with the explicit goal to exercise a specific functionality or to reach a specific state. The more specific such test cases become, the easier it is to decide whether or not the tests failed when executed. Furthermore, the runtime overhead incurred by the test environment and the decision procedures is of small concern. For a self-healing system, where these checks have to be performed at runtime, we cannot control the execution environment through scaffolding and test drivers, and we must meet strong constraints on the runtime overhead incurred by the checks added to the system.

In software testing, the decision whether or not a test was successful, that is the actual *failure detection*, is made by an oracle. Test oracles are designed according to three major approaches: comparison based, partial, or program self-checks [21]. Comparison based oracles require the specification of pairs of program and expected outputs. Partial oracles provide decision procedures that do not check for the exact expected output, but only if an output falls into a specific class of values; . they trade precision for generality, and possibly also performance. Self-checks in programs are assertions that check an invariant over a part of the program state whenever they are executed. Due to the usually infinite state space, we cannot implement comparison based oracles for monitoring the execution of production systems. Partial oracles and self-checks on the other hand have a finite representation and can be applied to all system executions.

It is a challenge for the implementers to know what to test for, or to decide which invariants need to be checked on the code-level. Revealing the connection between requirements, formulated in the idiom of system level entities, and their violations, usually detected as violations of assertions is a non-trivial task. In particular when the implementation of an entity from the conceptual model is distributed among several components, finding all code locations where an assertion must be checked is tedious and error prone. We propose an automatic process that supports the mapping between requirements and code-level assertions.

3. MAPPING REQUIREMENTS TO ASSERTIONS

Before describing our approach, we discuss the underlying assumptions and prerequisites. On the highest abstraction level, we assume to have a specification expressed in an informal requirements document, written in natural language, use cases or similar. Further, we assume a semi-formal, and most likely incomplete, model of the logical and concrete entities in the system, which provides the software engineer with hints on how the requirements and the actual implementation are linked. The model is usually incomplete and contains ambiguities that can cause subtle errors in the implementation. In the examples presented in this paper, we assume that this model is expressed in UML.

These artifacts are semantically related to each other. Software development processes describe this relationship as a sequence of refinements, that is, the system architecture and design refine the requirements, the implementation is based on the design, etc. Despite progress in the Model Driven Architecture (MDA) [18] and similar approaches, most of these refinement steps are only partially automated. This lack of automation is due to the fact that the early artifacts, especially the requirements specification and the high-level system design, are informal or at best semi-formal descriptions, which are incomplete and ambiguous.

With our approach, we propose an automatic technique that addresses the problems of incompleteness and ambiguity by mapping high-level requirements to executable assertions. Our technique follows four basic steps: first we extract useful information from the requirements, second we add this information as annotations to the conceptual model, third we map the constraints implied by the annotations to related code-level entities, and fourth we generate the code that checks for invariant violations whenever a constrained method is called. Here we focus on the first three steps, which represent the main research challenges.

1. Extracting Information from Requirements.

Extracting information from informal requirements documents is predominantly a human task that cannot be automated easily. The software engineer has to identify useful information, that is, information that implies constraints on the model. For example, requirements that can be formalized through the use of abstract data-types (ADT) or similar concepts that carry implicit semantic information. Abstract data-types are useful, because whenever an entity behaves like an ADT, we can check whether the entity's operations maintain the ADT's invariants. However, our approach is not limited to ADTs. We rather propose to define a generic, high-level annotation language that allows us to express any useful constraint. ADTs are just one well known example of an easy to understand high-level specification that has well defined semantics, which can be exploited in the later steps of our approach.

2. Annotating the Conceptual Model.

Annotating the system model turns the extracted information from a human readable requirements specification into a format that can be handled automatically. In this step we deal with system level requirements described by use cases or similar artifacts resulting from requirements elicitation. The model annotations that we derive from the

documentation are on the same level of abstraction, and do not directly map to implementation details. Our example in section 4 uses `sets` to add more details to the association specifications in the conceptual model of figure 2. The goal here is not to obtain a complete, detailed system model that specifies everything to the last detail. Instead, we aim to augment an inherently incomplete and imprecise model with information that allows us to generate self-checks for the code that implements the annotated entities.

3. Mapping Constraints to Code-Level Entities.

After having decided, based on the requirements, which abstract constraints are associated with the entities in the conceptual model, the implied invariants are mapped to operations and relationships pertaining to those entities. This second step makes the invariants explicit, and associates them with the correct elements in the model. In the examples presented in this paper, the semantics of these invariants is specified in the Object Constraint Language (OCL) [20]. Listing 1 shows an example of one possible invariant associated with the `add` operation of the abstract data-type `set`.

Whenever constrained entities described in the conceptual model are represented differently in the actual implementation, the implied constraints must be mapped to the corresponding code-level entities. The problem to be solved in this step is to find all code-locations that correspond to an operation of an entity in the conceptual model, and to transform the invariant to be checked into a form that is appropriate for this particular location. This step is trivial if there is a 1-to-1 mapping between the conceptual entities and the implementation.

4. Generating Code.

Once we have determined which invariants have to be checked, and in which code locations the checks need to be added, we generate the additional code to check the invariants and insert it into the program. Several concerns complicate this last step. Once the constraints are mapped to the right code-level entities, the code that needs to be generated is strongly influenced by how another component's services are requested. For example, for systems that are implemented in single locations, where service invocations are method calls on objects, we need to generate different code than for distributed systems, where components are invoked as web-services. This information might be available as part of the system model, but this step may likely require intervention by software engineers who need to provide such information manually.

Even though it is not a problem of the *process* of mapping the requirements to assertions, we must take the runtime overhead of the added checks into account when generating code. In a testing environment, where the runtime of checks is not a big issue, we can easily test for the adherence to the invariants in detail. In a production environment, where these checks are executed for every request to the system, performance becomes a major concern. In many cases, checking for the full invariant is too expensive and we can only check a weaker invariant. For example, to fully check the correctness of adding an element to a set we need to check that:

- if the element already exists in the set, the add oper-

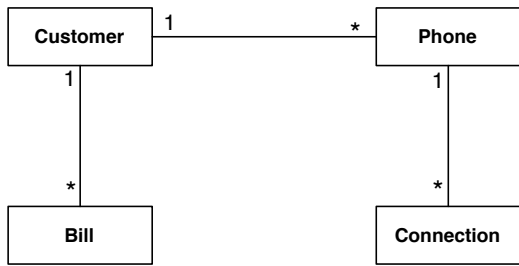


Figure 2: Conceptual model of the relations between data elements in the system.

ation does not alter the set.

- if the element does not already exist in the set, the element is added to the set, and all other elements remain unchanged.

Checking whether or not the set has been altered by adding an element can be done by storing the complete state of the set before and after the operation, but this may be too expensive in terms of memory usage and computation time. We can reduce memory consumption and processing time by checking the size of the set before and after the operation. Although checking for the invariance of the size of the set does not guarantee that the set is not changed, we may be satisfied with this partial check. Similarly, we can weaken the check for the actual addition of an element to a size check. Listing 1 shows the example of this weaker invariant. In general there are many possible ways to weaken an invariant. Which of those provide the best tradeoff between runtime overhead and precision is a question that depends on many factors in the implementation of a software system, and requires the expertise of the system designer (and possibly some experiments) to decide.

4. PHONE BILLING EXAMPLE

In this section we illustrate our technique with the example of a phone billing system. We refer only to a simplified excerpt of the whole system, which is already complex enough to show how to apply the technique in a realistic setting.

The conceptual model considered in this example, with the data entities and their relationships, is shown in figure 2. Customers can have several phone numbers registered to their name, and many calls can originate from each phone. Since bills are issued monthly, customers have a 1-to-many relationship with the *Bill* class. Note that because the class diagram in figure 2 represents the conceptual model of the data entities in the system, these entities will not be directly implemented as classes. Customers and traffic records reside in the respective databases, while bills are created on the fly from that data.

The relations between the three relevant components that implement the business logic of our system are shown in figure 3. The *Customer Management* component is responsible for maintaining all information pertaining to customers, registered phone numbers, etc. The *Traffic Monitor* registers the calls made from a phone and stores all information about calls that is needed for billing. The *Billing* component

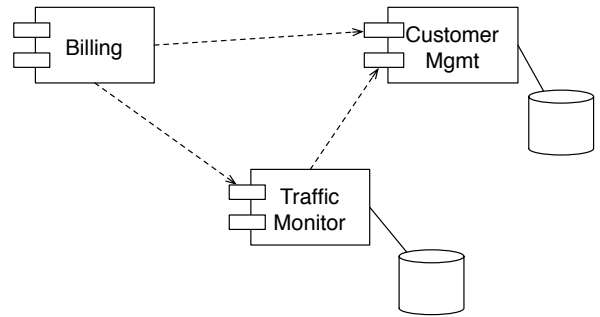


Figure 3: System model of the components relevant to the example.

is the functional unit that uses the information maintained by the other two components to generate the monthly bills for customers.

Furthermore, we assume that the entire application is designed as a 3-tier web-application, separating the business logic, such as compiling invoices, from the data storage of customer and traffic information.

As mentioned in the previous section, UML models are usually imprecise and ambiguous. For example, the associations in figure 2 do not specify whether the elements in a 1-to-many relation must be unique, or whether they need to be sorted. These design decisions must be made at some point and should be documented in the model. If they are appropriately documented we can use automated techniques to check that the constraints are not violated.

1. Extracting Information from Requirements.

The first step of our approach is analyzing the user requirements. For failure detection, we are interested in requirements that either explicitly specify or implicitly describe constraints. For example, one of the requirements for the phone billing system is that every connection should be recorded only for the phone to which it will be charged. Phrased in a more technical manner, this means that every such connection should be recorded only once in order to avoid double charges. This constrains the 1-to-many relation between *Phone* and *Connection* in figure 2, and we can conceptually treat it as a *set* of connections being associated with a phone.

Version 2 of the Unified Modeling Language (UML) allows associations to be annotated with *property strings* to specify more detailed semantics [19]. We can extend this notation and use the property string `{set}` to make clear that elements participating in this association must observe the behavior of sets. Treating an association between elements in our data model as an instance of a well known abstract data-type gives us an abstract interface specification and some constraints to check. In our example, repeatedly adding the same connection record to a phone should not change the number of connections associated with that phone, because the `add` operation of sets is idempotent.

2. Annotating the Conceptual Model.

In the second step of our approach we annotate the conceptual model with information to allow the automatic generation of assertions. To insert the assertions into the code,

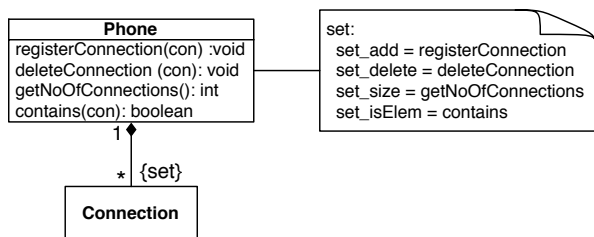


Figure 4: The Phone–Connection association with annotations.

we need to know which operations can have effects on the truth value of the assertions. In the simplest case the association is an aggregate association. The containing class would then need to have explicit methods to maintain the relationship, like `add()` or `remove()`. A typical pattern to achieve this in Java would be to implement the `Collection` interface. Since, in general, we cannot rely on the use of these simple implementation patterns, we need to add information about the required methods to the model. Figure 4 shows the aggregate association with the `{set}` annotation. The additional information about which methods in the `Phone` class correspond to the set interface is contained in a comment associated with the class itself. The `set` keyword in the comment starts a block of declarations that specify the names of the actual methods implementing the operations of the set interface.

3. Mapping Constraints to Code-Level Entities.

The invariants that we can check based on these annotations depend on the abstract data-type used. For example, for the `set` we could check that adding an element that already belongs to the set does not increase the size of the set. Listing 1 shows a template for this invariant written in OCL. The place-holders like `set_size`, `set_isElem`, etc. correspond to the variable names used in the annotations in figure 4. This generic template gets instantiated for every association in the model that is marked as a set.

```

context class::set_plus inv:
post:
  if set@pre.set_isElem( param1 )
    set.set_size = set@pre.set_size
  else
    set.set_size = set@pre.set_size + 1
  
```

Listing 1: OCL constraint template for adding elements to sets.

The instance for adding a connection to the phone record is shown in listing 2. Note that this invariant is not complete in the sense that it checks only that an element is added if it is not already in the set, but not, for example, that adding an already existing element to the set has other undesired side effects.

We noted before that the class diagram in figure 2 represents the conceptual model of the data entities in the system, but that those entities will not be implemented explicitly as classes connected directly by the shown associations. In the 3-tier architecture the actual data is stored in a database and individual records are only retrieved on demand with-

out populating the associations to other entities. For simplicity we assume that on the business logic tier customer data, including phone contracts, is managed by one component (`Customer Manager`), while monitoring call traffic and recording connection information is handled by another component (`Traffic Monitor`). On the database layer each of those components has a matching partner, encapsulating the actual database and providing an API for common queries.

```

context Phone::registerConnection inv:
post:
  if self@pre.contains( param1 )
    self.getNoOfConnections() =
      self@pre.getNoOfConnections()
  else
    self.getNoOfConnections() =
      self@pre.getNoOfConnections() + 1
  
```

Listing 2: The template instantiated for the Phone–Connection association.

If a customer dials a number to initiate a new connection, several things need to happen. First, his phone records are retrieved from the customer database in order to check that the phone has enough “credit” to start a connection. If the call is authorized and a connection is established, the traffic monitor must create a record of the call, including start- and end-times, connection types, etc. When the phone call terminates, that record is written to the traffic database. Writing back a call record corresponds to a call to `registerConnection()` on a `Phone` object. However, in the concrete implementation no explicit representation of the association between `Phone` and `Connection` exists. Thus, after having identified which constraints need to be checked on the conceptual model, step three of our approach must map the entities and associations from the conceptual- to the implementation model. The multi-tier implementation of the system, with proxies between the component managing the business logic and the database that ultimately manages the data, leaves us a lot of freedom to decide where to insert the assertion.

Assuming we have access to all code of the business logic and the wrapper component for the database, we can either insert the check between the wrapper and the actual database, or we can insert it between the wrapper and the logic component. This choice influences the level of detail of the information we can obtain when an assertion is violated; the fewer the components that are involved in servicing a specific request, the easier it is to pinpoint the component responsible for the failure. Since we focus on integration failures, we insert the checking code around the call from the `Traffic Monitor` to the `Traffic DB` components.

Once we have annotated the model, we generate the code that needs to be inserted in the selected locations. The insertion is done by means of aspect oriented programming (AOP) [14] techniques. This not only simplifies code manipulation, but also ensures that the assertions are inserted whenever the checked operations are called. Whether to use static or dynamic weaving techniques mostly depends on the availability of the code and possibly other concerns, which are outside the scope of this paper.

The OCL constraint in listing 1 can be translated into

a template for actual code straightforwardly¹. The `@pre` label indicates that some information has to be collected before the actual operation is called. When the template is instantiated for a concrete class, all the identifiers in listing 3 are replaced by the corresponding identifiers in the program matching the objects and operations denoted by them. For example, `class_instance` refers to an instance of the class implementing the `set.add` operation, `set` and `size` represent the accessors to the (conceptual) set of stored connection records and its size.

```
int pre_size = class_instance.set.size;
boolean pre_contains =
    class_instance.set.set_in( param1 );
```

Listing 3: Advice to be inserted before calling `set.add`.

The actual assertion to be checked is inserted after the method call. It is formulated as a simple condition that throws an exception when it does not hold. To weave these code fragments into the application they must be defined as advice in an aspect. Listing 4 shows the template code for the assertion.

```
if ( pre_contains &&
    class_instance.set.size != pre_size )
    throw new ConstraintViolatedException();
if ( !pre_contains &&
    class_instance.set.size != pre_size+1 )
    throw new ConstraintViolatedException();
```

Listing 4: Advice to be inserted after calling `set.add`.

5. TOWARDS A GENERAL SOLUTION

In the previous sections, we have shown how we can automatically detect component integration failures by mapping high-level, informal system requirements to code-level assertions. Our work highlights several new research problems. Here we discuss the main challenges: automating the different phases of our approach, defining modeling primitives and verifying model consistency, and providing complete taxonomies of failures to support automatic failure identification. We also point to research results from different areas that may be useful in addressing these problems.

Our main goal is to provide a set of tools to automatically derive the code assertions required for detecting failures and triggering healing mechanisms at runtime. To automate the process, we need to define a format for the information that on the one hand side makes it easy for software engineers to express the extracted properties, and on the other hand side is suitable to be directly fed to the tools. Extracting useful information from the requirements document requires *understanding* the semantics of the requirements document. Therefore, the extraction process can only be fully automated if the requirements specifications are expressed in a formal language, otherwise it requires preprocessing by software engineers. Once the information is expressed in a suitable notation, it can be automatically transformed into code annotations, by referring to a model of the relationships between system specifications, detailed design, and actual code.

¹For our example we use Java, but the approach is not limited to any specific language.

Figure 4 shows an example of annotations for the `set` ADT. This format is convenient to write for the software engineer, who extracts the relevant information from the requirements document, and it is formal and expressive enough to be automatically processed and translated into code-level assertions.

The steps of modeling the desired properties in an abstract, system-independent way and to then generate code from the specification through a multi-step transformation are similar to the general approach put forward by the model driven architecture community [18]. In particular, the step of translating from a platform-independent model to a platform-specific model in MDA is related to our separation of annotations in the conceptual and the implementation models.

The main problem shared with the MDA community is the consistency between models and all other artifacts in a dynamically evolving context. It is not yet realistic to assume that a complete executable system can be generated from an abstract model. Thus, when models or other artifacts are modified to adapt to new or evolving requirements, all related artifacts need to be modified as well. Some work has already been done on the problem of model evolution, incremental transformations and synchronizing models (for example [4, 12]). Some modeling tools, like Together [25], support the direct reflection of code changes in the corresponding UML model. Even though the approaches discussed in the context of MDA are problem specific, the general ideas put forward can probably help us address the problems of determining which steps can be automated, and how to keep our models and the implementation consistent.

Another important research question is what kind of failures our approach can possibly detect. To this end, a taxonomy that relates failure types to design artifacts is desired. Additionally, and more fundamentally, we would also like to assess to which extent our approach guarantees to catch *all* occurrences of a specific failure in a system. Aspect oriented techniques should assure complete coverage, but we would like to provide a sound theoretical foundation for this claim in the context of our work.

The classes of failures one can detect, and the precision of the detection mechanism are influenced by the decision procedures used to detect failures. The decision procedure has to be applicable to large, possibly infinite input domains. Thus we cannot use comparison based oracles. To be able to check the outcome of an operation or action sequence for all possible inputs, the decision procedure must be given in the form of a “program” or assertion that can be applied to any state or execution sequence. Ideally, the oracle would be precise enough to detect any possible failure. Unfortunately, even if a precise check is theoretically possible, it might be too computationally expensive. In cases where a precise check is not possible, the oracle needs to work on some kind of abstraction that is easier to obtain. In our example we used the size of the set as a measure to check whether or not the `add` operation had changed the set. However, partial oracles may not detect all failures.

Consider the following two examples: A partial oracle to test the outcome of a sorting procedure might just check that the output is sorted, not whether the output contains the elements of the input. A broken sort procedure might lose elements from the input, but the oracle would not detect that failure. An oracle that strictly checks only the

validity of a system state might not notice that the result of a given event sequence is not the expected state. For example, if we start in a valid state with an empty shopping cart, add an item, and end up with an empty shopping cart, the partial oracle would consider that state valid, even though it is not what we would expect. These two examples show some problems occurring when we use partial oracles or consider only state information. The broader question about the scope of our approach is still open.

In the larger context of self-healing software the failure detection approach we propose targets only the first phase in the autonomic adaptation cycle (see figure 1). To optimally exploit detected failures for fault diagnosis, it would be helpful if the monitored components were able to provide detailed information about themselves via some built-in reflection mechanism. In the context of this work we intend to explore how this “design for self-healing” could aid the fault diagnosis phase after a failure has been detected, and how software components can be designed to easily provide that information.

6. RELATED WORK

We observed in section 2 that the main focus of most research on self-healing systems has been on fault diagnosis and the ensuing adaptations, and relies on simple failure models. Failure detection has not received much attention as a research question in its own right yet. In the following we address related work on the fringes of the field of self-healing. We would like to note that most approaches cited in the introduction are prescriptive in the sense that with their simple failure models they often achieve a direct correspondence between fault and failure, and prescribe a fixed healing strategy. In contrast, we explicitly intend to decouple failure detection from fault diagnosis and repair to be more flexible with the possible reactions the the detected failures.

There are currently few self-management approaches that deal with functional problems, and none of them focus on failure detection. Candea et al. [3] propose a technique called *microreboot* to recover from problems that are due to internal faults. Their failure detection mechanism is simple, and does not attempt to actually locate the cause of the problems. An approach that attempts to deal with typical problems due to memory management and concurrency is presented by Qin and colleagues [22]. They instrument software with extra assertions and, whenever a specific assertion fails, they attempt to re-execute the same trace with a few changes to the environment and the memory allocation of the program. Both approaches are designed to deal with common failures, such as buffer overflows and memory leaks. They do not target problems arising from incorrect component implementations or uses, and they use heuristics to fix the problems without checking for the cause of the failure.

Skene and Emmerich [23] describe an approach to monitor requirements through a combination of MOF models and OCL invariant checkers. In their approach the service level agreements (SLA) are expressed in a MOF model, from which OCL constraints can be derived. Combined with the manual implementation of the actual monitoring code in the system, these invariants can be used to check if the execution violates the SLA. Stirewalt and Rugaber [24] propose to use OCL and automatic compilation to maintain invariants.

Their goal is to generate code from OCL invariants in a way that transparently integrates with programmer written code. The way they integrate their *mode components* into the system resembles the mechanisms employed by aspect oriented programming. These approaches contain elements similar to the technique we propose in this paper. However, they also rely on human effort to provide key part of the models they require for their approaches to work. One aim of our proposed technique is to reduce the required human involvement as much as possible.

Aspect oriented programming has been used by many researchers to implement various aspects of self-adaption [6, 11, 17]. Greenwood and Blair [11] uses AOP for monitoring and adaptation. In their example this works very well, because assertions and adaptations are tightly coupled. Thus, they can use a single aspect to handle both. Engel and Freisleben [6] use dynamic aspect weaving to implement various adaptations in an operating system kernel. In both cases, the failures being targeted are simple violations of performance or resource constraints.

In the domain of software testing there is an abundance of literature on designing or automatically deriving test oracles. Baresi and Young provide a detailed survey [1]. The ideas expounded there served as a starting point for our exploration of the differences incurred by the special needs of decision procedures for the autonomic feedback loop.

7. CONCLUSION

In this paper we presented an approach to augment models with information derived from user requirements to automatically generate assertions that check the systems adherence to these requirements. A detailed example showed how the technique can be used to generate self-checks that facilitate failure detection in the running system. This work is part of an effort to enable self-healing systems to deal with functional problems arising at the system and component integration level. The early results illustrated through the example are promising. The work described in this paper opens many new and interesting research questions.

As an important next step we are currently implementing an extension to the Eclipse Modeling Framework (EMF) [5] to experiment with the automatic translation of model annotations to assertions. An experimental validation of the ideas outlined in sections 3 and 4 should give us further insights into the problem and aid us in working through the presented research agenda.

8. ACKNOWLEDGMENTS

This work was partially supported by the IST PLASTIC project of the European Commission Sixth Framework Programme.

9. REFERENCES

- [1] L. Baresi and M. Young. Test oracles. Technical report, Department of Computer Science – University of Oregon, 1998.
- [2] D. Breitgand, E. Henis, and O. Shehory. Automated and adaptive threshold setting: Enabling technology for autonomy and self-management. In *Proc. of the 2nd Int. Conf. on Autonomic Computings*, 2005.
- [3] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery.

- In *Proc. of the 6th Symp. on Operating Systems Design and Implementation*, 2004.
- [4] A. Egyed. Fixing inconsistencies in UML design models. In *Proc. of the 27th Int. Conf. on Software Engineering (ICSE)*, pages 292–301, 2007.
- [5] Eclipse Modeling Framework. <http://www.eclipse.org/modeling/emf>. Accessed 2007-05-12.
- [6] M. Engel and B. Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD '05: Proc. of the 4th Int. Conf. on Aspect-oriented Software Development*, pages 51–62, New York, NY, USA, 2005. ACM Press.
- [7] J. Gao, G. Kar, and P. Kermani. Approaches to building self healing systems using dependency analysis. In *Proc. of IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2004.
- [8] D. Garlan, J. Kramer, and A. Wolf, editors. *WOSS '02: Proceedings of the first workshop on Self-healing systems*, 2002.
- [9] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In Garlan et al. [8], pages 27–32.
- [10] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In Garlan et al. [8], pages 33–38.
- [11] P. Greenwood and L. Blair. Using aspect-oriented programming to implement an autonomic system. In *Dynamic Aspects Workshop*, 2004.
- [12] D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Models in Software Engineering*, volume 4199 of *LNCS*, pages 321–335. Springer, 2006.
- [13] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.
- [14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 – Object-Oriented Programming*, pages 220–242, 1997.
- [15] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In L. C. Briand and A. L. Wolf, editors, *Future of Software Engineering (FoSE) 2007*, pages 259–268, 2007.
- [16] M. L. Littman, N. Ravi, E. Fenson, and R. Howard. An instance-based state representation for network repair. In *Proc. of the 19th National Conference on Artificial Intelligence*, pages 287–292, 2004.
- [17] L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis of component integration. In *ICECCS*, pages 292–301. IEEE Computer Society, 2005.
- [18] OMG. *MDA Guide*, version 1.0.1 edition, September 2003. Specification omg/03-06-01, downloaded from www.omg.org on 2005-09-29.
- [19] OMG. *UML 2.0 Superstructure Specification*, September 2003. Adopted specification ptc/03-08-02, downloaded from www.omg.org on 2005-04-05.
- [20] OMG. *Object Constraint Language*, May 2005. Available specification formal/06-05-01, downloaded from www.omg.org on 2006-12-11.
- [21] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, 2007.
- [22] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies — a safe method to survive software failures. In *Proc. of the 20th ACM Symp. on Operating Systems Principles, SOSP'05*, 2005.
- [23] J. Skene and W. Emmerich. Engineering runtime requirements-monitoring systems using MDA technologies. In R. D. Nicola and D. Sangiorgi, editors, *TGC*, volume 3705 of *LNCS*, pages 319–333. Springer, 2005.
- [24] K. Stirewalt and S. Rugaber. Automated invariant maintenance via OCL compilation. In L. C. Briand and C. Williams, editors, *Models in Software Engineering*, volume 3713 of *LNCS*, pages 616–632. Springer, 2005.
- [25] Borland Together. <http://www.borland.com/us/products/together/>.