Correct and Scalable Invariant Handling for Executing Software Systems

Sebastian Wilms
Johannes Kepler University
Linz, Austria
sebastian.wilms@jku.at

Alexander Nöhrer
Johannes Kepler University
Linz, Austria
alexander.noehrer@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT

Errors in software systems often lead to invalid system states, which are detectable through class invariants during the systems' execution. The benefits of invariant checking are well documented. Unfortunately, state-of-the-art approaches require extensive manual overhead for non-localized invariants because multiple versions of an invariant may be needed (re-writing problem). Moreover, invariant checking requires that triggers are placed in the code which is unscalable if done exhaustively or unreliable if optimized manually (trigger placement problem). And, to complicate matters, trigger placement and re-writing must be continuously adapted during system evolution. This paper presents a novel approach for invariant checking where engineers merely write a single version of the invariant and the approach automatically, correctly, and scalably handles the invariant checking. Our approach instruments the code to observe execution events and uses a constraint checker to analyze their consequences. The incremental nature of our approach poses only a small, constant execution overhead. A proof of concept tool for Java exists through which our approach's correctness and scalability were validated on three case studies (up to 50KLOC in size) and 46 invariants.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification—*Class invariants, Programming by contract*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors*

General Terms

Verification

Keywords

Invariants, Design-by-contract

1. INTRODUCTION

Invalid system behavior can be disastrous, especially if life is at risk or the failure causes significant loss of any kind. Design by contract [1] recognizes that incorrect behavior and invalid states correlate directly in many cases and invariant checking is an effective means for detecting invalid systems states[2]. In object oriented languages, valid states are typically defined in form of class invariants. Consider an actual industrial example of an invariant in a drone control system whereby a drone's position must always be within its assigned airspace. Such an invariant is critical because a drone's failure to remain inside its assigned airspace implies risk of life. The ability to detect such an invariant violation quickly and correctly is thus a fundamental necessity.

State-of-the-art solves this problem by injecting invariants into the systems' source code such that they are being checked alongside with the systems' execution. This requires deciding on the proper placement of invariant triggers to ensure that the invariant is checked at the right times during a system's execution (a human activity). Unfortunately, present approaches do not guarantee correctness except for localized invariants whose scope is limited to a single class or method. Yet, many interesting invariants are non-localized and span multiple classes such as the drone invariant example discussed above which involves a Drone and an Airspace. This invariant obviously needs to be checked every time the location of the drone changes. For example, during method calls that modify the location fields of a drone. Unfortunately, this is not enough because if the drone is assigned to another airspace then the drone's location may not change but due to its airspace change it may find itself violating the invariant. The invariant thus also needs to be checked whenever the drone's airspace changes. And there may be other locations.

One dilemma of invariant checking is that failure to identify all trigger locations in the system's code implies failure to detect all invariant violations and hence invalid system states (trigger placement problem). A trivial solution is to place invariant triggers throughout the entire code but this would lead to unscalable executions. Most state-of-the-art thus limited the locations where invariant triggers are placed but to the best of our knowledge no approach is able to decide this correctly and automatically. Another dilemma is that each location may require a different way of checking the invariant (invariant re-writing problem). For example, from the perspective of a drone, we need to check the drone's location with respect to its single airspace but from the perspective of the airspace we need to check all drones that are

within it whenever, for example, the airspace's boundary changes. Both invariant versions express the same invalid state but they have to be written differently because they focus on different changes - a change in the drone's location versus a change in the airspace's location. Finally, there is the evolution problem in that both the invariant trigger placement and the invariant rewriting are affected by code evolution. Those problems are thus not one-time problems whose cost might be rationalized as acceptable. State-of-the-art rarely acknowledges that invariant trigger placement are rewriting are recurring costs as code changes may require invariant changes and/or changes in trigger placement.

This paper contributes a fully automated, correct, and scalable approach to validating arbitrary, user-definable invariants. Contrary to most state-of-the-art, our approach does not place invariants within the code but rather observes the code's execution for cues to invariant checking. Specifically, our approach instruments the executing code to observed field (variable) changes and method calls and then uses an existing constraint checker to validate the implication of those changes and calls. Since exhaustive instrumentation would pose another kind of scalability problem, our approach was optimized to only react to those fields that are able to influence a given invariant. We found that the instrumentation has only a roughly 30% computational overhead and each invariant check has a constant computational overhead. Our approach's performance is thus in line with state-of-the-art and the improvement over the state-of-the-art is thus in on the quality side by avoiding both the trigger placement and re-writing problems. As input, our approach merely requires a list of invariants. The handling of the invariants and their continuous checking is fully automated. The correctness and scalability of our approach were validated on 3 case studies covering 46 invariants. Our current prototype is limited to Java and uses OCL as the invariant language. However, the kinds of instrumentation we performed are similar to the ones modern debuggers and profilers use. We thus believe that our approach should be readily applicable to other programming languages.

2. PROBLEM ILLUSTRATION

Let us focus next on the drone control system and our desire to detect a drone leaving its assigned airspace. For simplicity, we might want to think of an airspace as a cuboid with its boundary being defined in terms of a coordinate (x, y, z) and a *width*, *height*, and *depth*. The drones themselves have a position, also a coordinate (x, y, z). Listing 1 shows a possible implementation of the `Airspace` and the `Drone` classes including the invariant, which is implemented as a Java method. The invariant ensures that an assigned drone correctly references back to the airspace (bi-directionality) and the drone's location is within the bounds of that airspace. The invariant returns `true` if the class is in a valid state; otherwise it returns false. As is common in practice, triggers for invariant checks are inserted manually using the `assert` statement. The common notion of invariants regarding design by contract is that an invariant has to hold whenever an instance of its class is in a publicly visible state [3]. In short, an object is in a *publicly visible state* during the execution of a program, when no code from the class definition is currently executed on the object (i. e., none of its methods can be called from the other objects). Indeed, invariant placement seems trivial in this example. A brute force, naive approach would add the `assert` trigger to all method postconditions but this is generally perceived as computationally expensive and impractical. Some state-of-the-art limits the trigger placement but there are two main problems discussed next.

```
class Drone {
  double x, y, z = 0.0;
  Airspace airspace = null;
  Drone(Airspace airspace) {
  this.airspace = airspace;
  }
  setLocation(double nx, double ny, double nz) {
    this.x = nx; this.y=ny; this.z=yz;
  assert checkInvariant();
  }
  boolean checkInvariant() {
    return airspace != null && airspace.isInBounds(this)
          && airspace.drones.contains(this);
  }
}

class Airspace {
  Collection<Drone> drones;
  double x, y, z, width, height, depth;
  Airspace() {
    drones = new ArrayList<Drone>();
    setBounds(0, 0, 0, 0, 0, 0);
  }
   void addDrone(Drone d) {
    drones.add(d);
    d.airspace = this;
    assert checkInvariant();
  }
  void removeDrone(Drone d) {
    drones.remove(d);
    d.airspace = null;
    assert checkInvariant();
  }
  void setBounds(double x, double y, double z, double
       w, double h, double d) {
    this.x = x; this.y = y; this.z = z;
    this.width = w; this.height = h; this.depth = d;
    assert checkInvariant();
  }
  boolean isInBounds(Drone d) {
    return (d.x > x && d.x < x+width && d.y > y &&
        d.y < y+height && d.z > x && d.z < z+depth);
  }
  boolean checkInvariant() {
    for (Drone d : drones) {
      if !(d.airspace == this && isInBounds(d))
          return false;
    }
    return true;
  }
}
```

Listing 1: Airspace and Drone Implementation

2.1 Invariant Re-Writing Problem

The first problem we observe is that the developer has to write the invariant twice (manually) in this example. Both invariants, the one in the `Airspace` class and the one in the

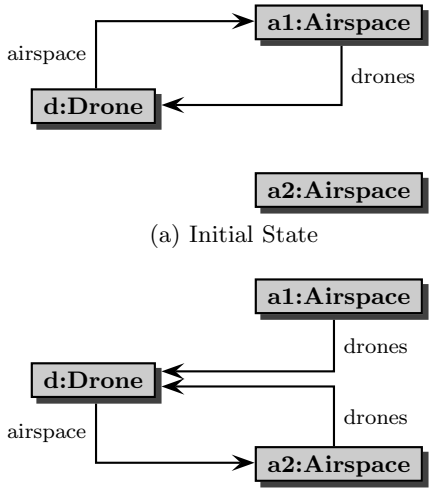
`Drone` class, check for the same illegal state but they have to do so differently (notice the different, albeit similar `checkInvariant` methods in the Drone and Airspace classes in Listing 1). Usually this is achieved by re-writing the invariant to fit the perspective of a class. To the best of our knowledge, no existing approach is able to automate the rewriting problem and it is easy to see that manual re-writing can be error-prone and, worse, seems redundant. However, without this re-written invariant, it would be hard or computationally very expensive to detect invariant violations caused by changes in a drone.

2.2 Invariant Placement Problem

The second problem we can observe, is that the developer has to decide on the proper trigger locations for the invariant checks. In Listing 1, triggers are identifiable through the `assert` statement and we see that most methods include one as the final statement. Invariant checking happens during systems executionS whenever such a statement is reached. An unnecessary trigger reduces execution performance and may even lead to false invariant detection (i.e., when the class is not in a publicly visible state). Exhaustive invariant checking is thus not desirable (see Section 6 for more details). A missing trigger, on the other hand, is also not desirable we may fail to identify an invariant violation. In state-of-the-art it is thus necessary manually to decide where to place triggers and even in this simple example it is not trivial as our Listing 1 has an error. Let us consider more closely the bi-directionality between `Airspace` and `Drone`. Figure 1 shows two possible states of a drone in context of two airspaces. The initial state shown in Figure 1a is perfectly fine and does not violate the bi-directionality of the invariant. After calling `a2.addDrone(d)`, it transitions into the state shown in Figure 1b, which is broken since a drone per our definition cannot be in two airspaces at once and our implementation of `addDrone` is missing the removal from the old airspace. Using our manually inserted assertions, we are unable to detect this error at this time, since the invariant will only be checked for the airspace `a2` (note that it was `a2`'s `addDrone` method that got called). After all, airspace `a2` is in a valid state thereafter but not airspace `a1`. Due to the nature of the implementation, the invariant is only checked in the airspace `a2` and the violation in airspace `a1` goes unnoticed until a method call on `a1` finally triggers the necessary invariant check. This may happen much latter or never. Correct invariant placement thus requires two triggers inside the `addDrone` method - one to check the new airspace and the other to check the old airspace. Manual invariant trigger placement is thus not only time consuming but also error prone. Moreover, the trigger placements need to be maintained and evolved with the code (e.g., when methods are added or modified). For example, if we add a new method `setAirspace` in `Drone` then we need to consider whether to place a trigger there also.

3. VISION AND RESEARCH QUESTIONS

Our vision is that the developer defines single versions of invariants only and the approach fully automatically, scalably, and correctly handles the invariant placement and rewriting - even when the code evolves. Instant feedback should be provided as soon as an invariant is violated. Our vision is thus that invariant checking is performed live during the systems' execution. Next we introduce our approach



(a) Initial State

(b) Broken State after Change

Figure 1: Example Instantiation of the Drone Example

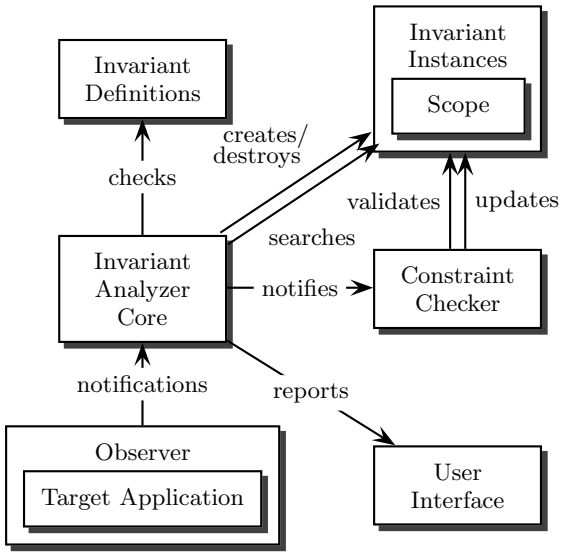


Figure 2: Architecture Overview of our Approach

to invariant checking and in the evaluation we answer the following questions:

1. Does our approach avoid the manual invariant placement problem?
2. Does our approach avoid the manual re-writing problem?
3. Is our approach guaranteed correct?
4. Does our approach scale?
5. Does our approach handle code evolution?

4. APPROACH

The basic principles of our approach follows previous work on design model consistency checking [4, 5, 6]. An architectural overview is depicted in Figure 2. The core is the *Invariant Analyzer Core* which coordinates all other components. It receives notifications about object creations, method calls,

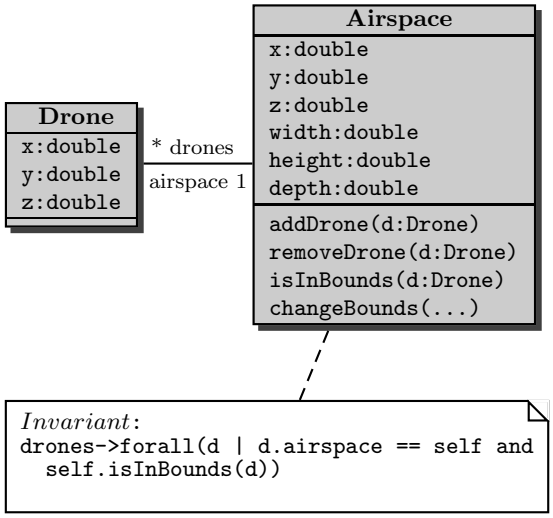


Figure 3: Single Invariant for the Drone Example

field modifications, and object destruction as these notifications are potentially state changing events and thus require our attention. These events are generated by instrumenting the execution environment (presently done in Java as a proof of concept). Upon receiving such notifications, depending on the type of notification, invariants are instantiated and evaluated, which will be explained in detail in Section 4.1. *Invariant Definitions* represent the invariants provided by the developer. Invariants are expected to have a context, which is a reference to a class for which the invariant must hold. As we have seen above, there are as many contexts as there are versions. In our approach the developer may choose any arbitrary version (hence and arbitrary context).

For example, the developer may want to define an invariant for Airspace as in Figure 3 where the context of the invariant in Figure 3 is the `Airspace` class as indicated by the dashed line. The reverse invariant for `Drone` does not have to be defined explicitly but is implied. Since a class might be instantiated multiple times into objects and these objects might be in different states during execution (e. g., in Figure 1 we see two instances of `Airspace` and their fields obviously differ), we require a separate evaluation for each instance. In our approach, we handle this by defining *Invariant Instances* to mirror class instances. That is, for each invariant definition there are as many invariant instances as there are instances of invariant definition's context (e. g., there would be two invariant instances for the two `Airspace` objects in Figure 1 but no invariant instance for the `Drone` object since no invariant was defined with `Drone` as its context). Each invariant instance is tasked with checking its invariant definition on the object it is assigned to. We refer to a context element as the object for which an invariant instance is responsible for. *Invariant Instances* also contain a change impact *Scope*, which is a set of references to object fields that affect the validity of the invariant. We define the individual elements of a *Scope* as *scope elements*. How the *Scope* is computed is explained below but the key point is that the scope identifies which notification events (e. g., field modifications, see above) should trigger the re-validation of its invariant instance. The *Constraint Checker* is responsible for validating the *Invariant Instances* and also keeps the scope up-to-date. Information about the instances and their validation results are reported to the *User Interface*.

4.1 Notification Scenarios

As previously mentioned, our approach reacts to events in an observed target application. Events of interest to us are state-changing activities that occur during the execution of the system. These are the creation of a new object, the modification of a field, or the destruction of an object. Usually multiple notifications about such field modifications, creations, and destruction get collected as the object transitions between publicly visible states (recall Section 2). Such transitions are simply executions that include method calls of a given object in question where we define an *Interval* to be the time between such publicly visible states. Invariant checking should occur at the end of each such interval. Next we will discuss each of the notifications that trigger the *Invariant Analyzer Core* to take a look at the notifications collected since the last trigger event and how it reacts to them. Algorithm 1 shows a brief overview how we determine which invariant instances need to be instantiated or re-validated. CS denotes the set of changes collected during a given interval. The *type* of a change may either be a field modification, object creation or deletion. The set of all currently managed invariant instances is referred to as II and SE denotes the set of all scope elements in any of the individual invariant instance's scopes (i. e. $\bigcup_{ii \in II} ii.scope$). Please note that the depicted algorithm is overly simplified and does not depict optimizations.

4.1.1 Creation of a New Object

We need to keep track of the objects existing in the target application (referring to Lines 11–15 in Algorithm 1). For example, in Figure 1 we illustrated a scenario where there were two objects of type `Airspace` and an object of type `Drone`. The problem of incorrect invariant placement discussed above were due to the fact that existing state-of-the-art often fails to keep track of every object to ensure that the invariants of all affected objects are evaluated – not just the invariant of the one who experienced a state change. As a consequence the creation of an object is something our approach needs to keep track of. The triggering notification for this scenario is when the constructor of the object in question is exited. Upon receiving this event the *Invariant Definitions* is searched to find all invariants whose context element is the type of the newly created object it is assignable to. For every match found a new *Invariant Instance* is created and immediately validated by the *Consistency Checker*. For example, when object `a1` is created than an instance of the invariant in Figure 3 is created because the invariant's context element is `Airspace`, which is the same as the created object's type. We denote this invariant instance as `Invariant[a1]`. Similarly, another invariant instance is created for `a2` – denoted as `Invariant[a2]`. However, no invariant instance is created when Drone d is instantiated because there is no invariant with this context element (recall that our approach does not require invariant re-writing).

4.1.2 Destruction of an Object

In an object orientated environment, objects have a limited lifetime. The disappearance of an object will not change the outcome of any invariant instance,instead it causes an

Algorithm 1 Analyzing Changes and triggering Invariant Instance validations

Input: a set of changes CS

1: $II_e \leftarrow \emptyset$ {the set of invariant instances to validate}
2: $II_c \leftarrow \emptyset$ {the set of newly created invariant instances}
3: $II_r \leftarrow \emptyset$ {the set of removed invariant instances}
4: **for all** $change \in CS$ **do**
5: **if** $change.type =$ `MOD` **then**
6: **for all** $se \in SE$ **do**
7: **if** $se.object = change.object \wedge se.field = change.field$ **then**
8: $II_e \leftarrow II_e \cup se.iinstances$
9: **end if**
10: **end for**
11: **else if** $change.type =$ `CREATE` **then**
12: **for all** $i \in \{x \in I \mid change.object.type \preceq x.context\}$ **do**
13: instantiate new invariant instance ii of invariant i with context element $change.object$
14: $II_c \leftarrow II_c \cup \{ii\}$
15: **end for**
16: **else if** $change.type =$ `DELETE` **then**
17: **for all** $ii \in \{x \in II \mid x.contextElement = change.object\}$ **do**
18: $II_r \leftarrow II_r \cup \{ii\}$
19: **end for**
20: **end if**
21: **end for**
22: $II_e \leftarrow II_e \cup II_c$
23: $II \leftarrow II \cup II_c$
24: $II \leftarrow II \setminus II_r$
25: report creation of invariant instances II_c
26: report removal of invariant instances II_r
27: **for all** $ii \in II_e$ **do**
28: validate invariant instance ii and update its scope
29: report new result and scope of ii
30: **end for**

invariant instance to become obsolete. Although an invariant may ensure that an object with certain properties exits in a collection and deleting it may violate the invariant, prior to the deletion a notification stating that the contents of the collection changes will ensure that the related invariant instances are re-validated. Thus, upon receiving a deletion notification the *Invariant Instances* get searched for instances associated with the object to be destroyed and the *Invariant Analyzer Core* takes care destroying the associated invariant instances (as shown in Lines 16–20 in Algorithm 1). The destruction event is thus a house-keeping event that notifies us of objects that no longer need to be monitored.

4.1.3 Field Modification

In object-oriented systems, the state of an object is defined by the values of its fields – e. g., the location value of a `Drone` instance. Usually, the state of an object will change over time. This is only possible in response to code execution – e. g., a method call such as `setBounds` in `Airspace` which modifies the fields `x`, `y`, `z`, etc. Whenever fields change, the system state may change and become invalid. Hence, the invariant needs to be re-validated at these times to check for possible invalid state changes. Since invalid states are allowed for as long as the object is not in a publicly visible state [3] it is not desired to react immediately to field modifications. A field modification is thus a state changing event but it should not be used as a trigger for invariant checking. Similar to existing approaches, we use the transition to a publicly visible state, i. e. method calls for this purpose, to trigger the actual invariant check. Since multiple field modifications may occur between publicly visible states, all these field modifications must be queued up. Upon reaching a publicly visible state, the *Invariant Instances* get searched for instances that are affected by the queued field modifications. The *Invariant Analyzer Core* determines which invariant instances require re-validaton by comparing the invariant instance's scope with the queued field modifications. Recall that the scope contains a set of all fields whose change should trigger a re-validation. Thus, if at least one field modification is contained in an invariant instance's scope then this invariant instance is affected by the modification and requires re-validation. How the scope is computed is discussed below in Section 4.2.

Note that not all intervals between publicly visible states require invariant re-validations (i.e., this is something a brute force approach might do). A fair amount of methods merely query the state of an object without actually modifying it (e.g., "getter" methods); or they might modify a field that is not in the scope of any invariant instance. In such a case whichever invariant did hold beforehand will hold afterward. In the case of non-state changing methods, existing approaches for invariant checking offer the ability to mark such methods as simple queries, instructing it to not generate / perform any invariant checks after said method was called (see 6). However, using such markers is another source for human introduced errors. During the evolution of a software system a simple query may become a method that does change the state of an object. Not removing the markers in such a case may cause errors to go unnoticed. Using our approach it is not required to use markers or any manual input as it will detect state-changing methods automatically during run-time. In fact, this is quite simple because a non-state-changing method will not cause a field modification during its execution.

As an example of how our approach reacts to a field modification, consider the method `addDrone` from Listing 1 executed on an `Airspace` object `a2` as described in Problem 2.2. When this method is called then the statement `drones.add(d)` is executed which modifies the collection drones in object `a2`. In response, the *Invariant Analyzer Core* would receive the notification about a field modification in `a2.drones`. And since `a2.drones` is in the scope of the invariant instance `Invariant[a2]`, it is affected by this change. However, since the method has not finished executing yet, `a2` is not in a publicly visible state. Triggering an invariant validation at this point would result in a violation, which is in fact a false violation. Indeed, after the second statement `d.airspace = this`, this violation will resolve itself, showing the need to prevent invariant checking during intermediate invalid states. Hence, our approach collects all notifications until the method is exited and the *Invariant Analyzer Core* would react to the notifications about field assignments for `{a2.drones, d.airspace}` (i.e., these are the two field changes that occurred in that method). The *Invariant Analyzer Core* would discover that `Invariant[a2]` requires re-validation because its scope contains `a2.drones`. The *Invariant Analyzer Core* thus notifies the *Constraint*

Checker to re-validate it. Since `Invariant[a2]` is still valid, the *Invariant Analyzer Core* would move on to the next notification. There, the *Invariant Analyzer Core* would discover that `Invariant[a1]` requires re-validation because its scope contains `d.airspace`. As we know, `Invariant[a1]` is violated now as was illustrated and discussed in Figure 1. Since the change happened in `a2` and `a1` remained unchanged, many existing approaches would miss the violation to `a1`. Our approach, on the other hand, detects this problem because we have an invariant instance dedicated for every object that needs one. However, we have yet to reveal how invariant instances built up the scope. This is discussed next.

4.2 Constraint Checker

The *Constraint Checker* is used to validate and re-validate *Invariant Instances* and update their *Scope*. The scope of an Invariant Instance is simply the list of fields that affect its validity. Since the validity of an invariant instance is based on the fields it investigates, the scope is the list of all fields that the invariant instance required during validation. Our approach thus also observes field read accesses in addition to field modifications. Whenever the *Constraint Checker* validates an invariant instance, all fields it accesses are recorded. The *Scope* is thus the union of all field accesses it recorded which we can be build up by observing the execution of the invariant. This observation and therefore the *Scope* includes field read accesses from the invariant itself and also from all methods that are used within the invariant, such as `isInBounds` from our example. This is necessary since a change introduced to one of those fields may cause the method to return a different value and therefore possibly change the result of the invariant instance. For example, `Invariant[a1]` accessed during its validation for the initial state in Figure 1a `a1`'s location (`x`, `y`, `z`, etc.) and compared it with each drone's location (`d.x`, `d.y`, `d.z`); and it made sure that `d.airspace` is pointing back towards `a1`. The set of accessed fields is thus: `{a1.x, a1.y, a1.z, a1.width, a1.length, a1.depth, a1.drones, d.x, d.y, d.z, d.airspace}`. Note that `Invariant[a1]` did not access fields such as `a2.x` or `a2.drones`. These fields are accessed by `Invariant[a2]`. So, after calling `a2.addDrone(d)`, which was described in Problem 2.2, the fields `a2.drones` and `d.airspace` get modified: hence the notification being `{a2.drones, d.airspace}` as was discussed inSection 4.1. The field `a2.drones` was previously accessed during the validation of `Invariant[a2]`. Hence, this instance must be re-validated. Moreover, the field `d.airspace` was previously accessed during the validation of `Invariant[a1]`. That instance must also be re-validated. Our approach thus simply and elegantly solves the flaws of the local invariant checking problem without requiring any invariant re-writing (discussed in Section 2).

4.2.1 First-Time Validation of an Invariant Instance

As we know, an *Invariant Instance* is created together with the object it monitors. The *Invariant Definition* is used as a template and references to `self` are replaced with the actual object the invariant monitors (i.e., `self` refers to the object for which the invariant instances was created for). After the creation, the instance is immediately validated and hence its *Scope* is set.

Imagine during the execution of our example that after the state shown in Figure 1b, at some point, a new `Airspace a3` is created. After the instantiation is complete, we collect all relevant invariants and instantiate them as well (e. g. `Invariant[a3]`). We validate those new invariant instances and store their scope. Since initially the drones collection is empty, no drone objects will be accessed. After the validation is completed we store the scope of the invariant instance and report the result. In this example the scope of `Invariant[a3]` would be `{a3.drones}` only because there are no drones assigned to the airspace yet.

4.2.2 Re-validation of an Invariant Instance

We already discussed that the re-validation of an invariant instance is triggered if an execution notification matches an element in its scope. During the re-validation of *Invariant Instances* the result can change but, even more interestingly, the *Scope* can change also. Consider the example given in Section 4.2.1 where the initial scope of `Invariant[a3]` contained only `{a3.drones}`. Now consider a call to `a3.addDrone(d)`. This would of course cause the invariant instance `Invariant[a3]` to be re-validated (among others, which would fail, as pointed out in Figure 1b), as indicated in Lines 5–10 in Algorithm 1). However, since the `drones` collection would not be empty anymore, the re-validation would also access the newly assigned drone. As a result, the *Scope* of `Invariant[a3]` would change to `{a3.drones, a3.x, a3.y, a3.z, a3.width, a3.height, a3.depth, d.x, d.y, d.z, d.airspace}`.

4.3 Implementation Details

Our prototypical implementation has been fully implemented in the Java programming language and is able to observe and check invariants for arbitrary programs running in a Java Virtual Machine (JVM). The implementation is according to Figure 2. The *Invariant Definitions* are written in OCL, the implementation includes a parser, which is also used for code-completion. In fact, the available types in a JVM may change over time due class loading / unloading. Thus, we are able to cope with lazy-loading in state-of-the-art run-time environments. A graphical *User Interface* is built on top of the Eclipse Platform. It offers an editor for defining invariants and visualizes the results of invariant checks and their scopes. For the *Observer* component we perform byte-code instrumentation of the loaded class. We already mentioned, that we are able to cope with lazy-loading most JVMs perform. Although the appearance of a new class does not change the result of any invariant instance, there may be invariants in the invariant storage for a type that does not exist yet. Therefore, additionally to the events mentioned beforehand, we also have to observe the loading / unloading of classes. Whenever the run-time environment loads a new class we treat it as if the invariants defined on that class have just been added.

Since none of the components of our approach are specific to Java and/or OCL, they could be adapted to work with other constraint languages or run-time environments. Indeed, we have two variations of our tool: one where the invariant checker uses byte code instruction to run within the process of the system being monitored and one where the invariant checker uses the Java Platform Debugger Architecture [7] to run in another process. There is a simple trade-off. The in-process implementation is much faster compared to the cross-process communication. However, the latter allows for "external debugging" where the invariant checker does

Table 1: Case Studies used for Evaluation

Application	LOC	#c	#m	#f	#i
ATM	572	13	8.769	1.846	17
jPacMan	1595	25	8.459	2.333	21
GanttProject	48405	1059	6.424	2.361	8

not have to execute on the same workstation as the system being monitored. Screen snap shots and a short tool demonstration can be viewed at http://www.sea.jku.at/tools/invariantchecker/.

5. EVALUATION

We empirically evaluated our approach to assess our research questions stated in Section **??** using three different open source applications as case studies. Details for our case studies are given in Table 1, like the lines of code (LOC), the number of classes (#c), the average number of methods per class (#m), the average number of object fields (#f) and the number of class invariants (#i) we used. The invariants we used were mostly defined by the original developers of these applications, either directly implemented in Java, or stated as annotations using JML. Others were derived by examining the applications source code or supplied test cases.

For evaluating our approach we had three major concerns:

1. Correctness: Is our approach able to detect all invariant violations?
2. Scalability: How does our approach scale up with increasing "size" of a system, especially increasing amount of objects and invariants?
3. Performance: How does our approach perform generally, i. e. to which degree does it slow down the "ordinary" execution of a system?

Recall that two additional research questions focused on whether our approach solved the rewriting and trigger placement problems. These are briefly discussed also though they are proven through the algorithm and its correctness. That is, our evaluation did not require manual rewriting or trigger placement but still functioned correctly and scalably as will be shown next. The correctness and scalability evaluation thus also demonstrates the correct solving of the rewriting and trigger placement problems.

To address these questions we executed the test cases shipped with the source code of the applications and analyzed the behavior of our approach. All time measurements stated in this section were performed on a standard PC with an Intel Q9550@2.83Ghz processor and 8GB of RAM.

5.1 Correctness

To verify correctness (research question 3), we compared our results with an excessive brute-force approach. This approach revalidates all the invariants at all times after each change from a publicly visible state, which is correct by definition. While it is computationally inscalable, the brute force approach does provide us with a golden standard for when invariants are violated. This is done by checking whether each violation, or more precisely each change of an invariant instances result, detected by the brute-force approach is also detected by our approach. Although this does not formally prove completeness, the extensive empirical tests involving 13,803 checks revealed no single case of incorrectness (see Table 2) and the large number of tests strongly support this.

Table 2: Invariant checks performed

System	# checks	instance changes
ATM	112	43
jPacMan	9386	7108
GanttProject	4305	1996

Table 3: Objects created during system run

Application	# Objects
ATM	112
jPacMan	63218
GanttProject	230643

5.2 Scalability

This subsection assesses the issue of scalability (research question 4). Typically, the scalability of a system is determined by complexity metrics like lines of code. However, for invariant checking this is not useful because such metrics reveal little about the number of class instances that are created or field manipulations that occur (e.g., a small system could trigger more invariant checks than a larger one). We thus opted to measure scalability using a more meaningful dynamic size – in particular, the number of class instances (ω) as they represent data/memory. Table 3 shows the overall amount of objects created by our case studies. The following evaluation will provide formal arguments and empirical evidence that our approach does scale and behaves linear in terms of computational complexity.

The scalability of our approach depends on several factors and we investigated their complexity and impact on scalability. Whenever the invariant checking mechanism is triggered, two major operations are performed: 1) looking up the invariant instances that need to be (re)validated, if any (T_{lookup}) and 2) validating them (T_{eval}). For a single interval, we can state the following complexity:

$$T_{interval} = O\left(T_{lookup} + T_{eval}\right)$$

T_{lookup} is dependent on two factors, the number of collected change notifications (γ) and how long it takes to perform a lookup (t_{lookup}). Although not actually constant, we found that γ will remain relatively stable during execution. Although it could rise in larger systems, we found that it is in fact independent of the system size in practice because each method/interval typically only executes so many statements and methods in larger systems do not necessarily execute more statements than methods in smaller systems (we omit this data for brevity but below we also demonstrate the scalability of γ_r which is a random subset of γ). The individual lookup (t_{lookup}) is implemented as a simple hashtable lookup and thus constant.

$$\begin{aligned} \gamma &\ldots \text{number of changes} \\ t_{lookup} &\ldots \text{time needed for a single lookup} \rightarrow O(1) \\ T_{lookup} &= O\left(\gamma \cdot t_{lookup}\right) = O\left(\gamma\right) \end{aligned}$$

T_{eval} depends on how many of the notifications are actually relevant (γ_r), how invariant instances are affected by

Table 4: Maximum number of intervals

Application	# Intervals
ATM	90
jPacMan	63965
GanttProject	231974

the changes and the time required to perform the invariant checks (t_{eval}). γ_r are those changes that are in the scope of one the invariant instances or object creations triggering the creation of an invariant instance and will remain mostly constant similar to γ. Below we demonstrate that this is in fact true. The number of affected invariant instances depends on γ_r and how many invariant instances are affected per change (κ). Below we further decompose and analyze those factors. We demonstrate that both κ and t_{eval} are not strictly constants but will remain small and stable during execution.

$$\begin{aligned} \gamma_r &\ldots \text{relevant changes} \\ \gamma_r &\subseteq \gamma \\ t_{eval} &\ldots \text{time needed to validate a single invariant} \\ \kappa &\ldots \text{affected invariant instances per change} \\ T_{eval} &= O\left(\gamma_r \cdot \kappa \cdot t_{eval}\right) \end{aligned}$$

Overall, the complexity of our approach is linear. During execution our invariant checking algorithm will be triggered m times, where m represents the number of intervals, and since $T_{interval}$ remains mostly constant the overall complexity is linear.

$$T_{overall} = m \cdot T_{interval} = O\left(m \cdot (\gamma_m + \gamma_m \cdot \kappa \cdot t_{eval})\right)$$

Empirical evidence for the linear behavior is provided in Figure 4, showing that $T_{overall}$ did in fact increase linearly with m in our case studies. In this figure we normalized the number of intervals (0% being 0 and 100% the maximum) since they differed greatly in the different system runs. Table 4 shows the total number of intervals for each case study. The unusual high validation times in case of the ATM example is caused by two invariants iterating over a huge array. However, these invariants are only validated after initialization and $T_{overall}$ rises linearly from there on. Note that the complexity of the actual invariant is an external, user-defined factor that is not under the control of our approach. This high validation time is thus not significant because every invariant checking approach is equally affected by this. What is significant is that our approach exhibited linear run-time performance which is contrary to a brute force approach that would be exponentially complex.

5.2.1 *Relevant changes* γ_r

We claimed that γ_r remains mostly constant, i. e. is independent of the system size ω. We provide empirical evidence for this claim in Figure 5, showing that γ_r does not grow with increasing ω. Since we checked 13,803 invariants, we grouped ω in 5% increments and present their averages. We see that in average only very few changes trigger relevant invariant checking (summarized in Table 5 which shows the number of average relevant changes per interval m).

5.2.2 *Affected invariant instances* κ

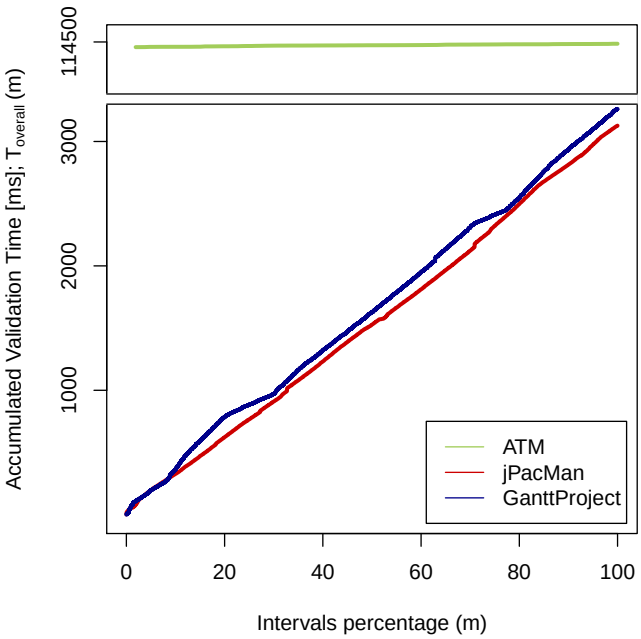


Figure 4: Relation between validation time and intervals

Table 5: Average changes per interval

System	Average Changes
ATM	0.867
jPacMan	0.083
GanttProject	0.015

The factor κ depends on the number of defined invariants r as well as their complexity. The more invariants exist, the more likely it is that one of them accesses a particular scope element. The number of defined invariants remains constant during a system's execution. One might think that might not scale during the execution due to more context elements ($o \cdot \omega$) instantiating over time. The factor o ($0 \leq o \leq 1$) describes how many of the existing objects are actual context elements of invariant instances. Figure 6 shows with increasing size (and thus the number of context elements) κ does not necessarily increase. We used a similar grouping as already explained for Figure 5.

5.2.3 *Validation time* t_{eval}

The time required to validate an invariant instance (t_{eval}) depends on the computational complexity of the invariant definition. Our approach does not optimize the checking of individual invariants and all approaches to invariant checking are affected by this factor. In only a few cases did we notice that the checking of individual invariants grows with the number of objects existing during the execution of a system. This usually occurs when using expressions that iterate over collections. However, we did find that not even t_{eval} actually depends on the number of existing objects ω – there were only a few exceptions as was already discussed. Empirical evidence thus suggests that the evaluation of an invariant is actually constant for most constraints because they tend to access a fixed amount of objects or a fixed amount of expressions, regardless of size. Figure 7 demonstrates that the time required to validate individual invariants is small and does not grow with an increasing dynamic system size.

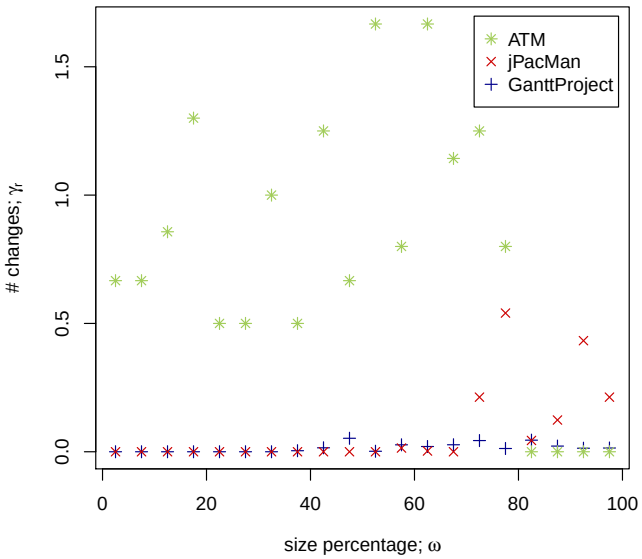


Figure 5: Relevant changes γ_r depending on system size ω

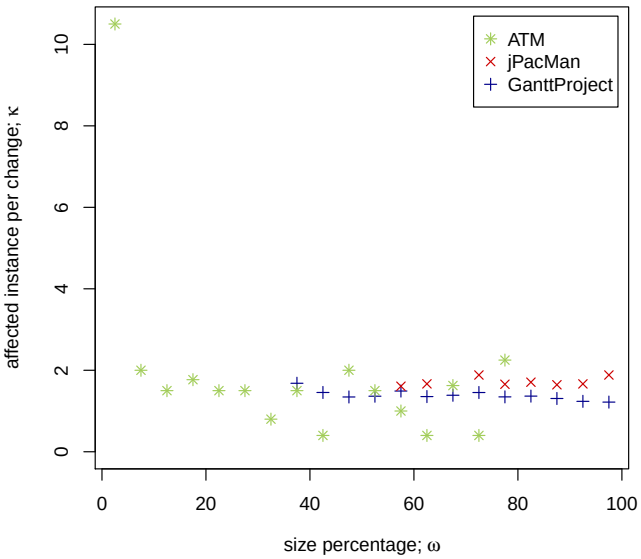


Figure 6: Affected invariant instances κ per change with increasing size ω

5.3 Invariant Placement, Re-Writing, and Evolution

The manual invariant placement and re-writing problems (research question 1 and 2) are avoided implicitly through our algorithm. Since for all our case studies we never manually decided on invariant trigger placement and given that the approach was correct always we conclude that our approach indeed avoids the manual invariant placement problem. The correctness evaluation also implicitly demonstrates that invariant triggering locations never have to be maintained whenever the code evolves since manual placement is no longer necessary - i.e., if our approach functioned correctly for all given case study systems and all given invariants, we have strong support in claiming that it would function correctly with other versions also. The same applies to the re-writing problem because we had a range of non-localized invariants yet our approach was able to correctly validate them without requiring re-writing.

5.4 Performance

While our approach does scale, it does not mean that there is no performance overhead. During the evaluation we did recognize a noticeable performance decrease compared to a system run without our invariant checker. Some of this overhead is caused by the invariant checks themselves. But, we can safely ignore this overhead since other approaches would have this overhead also. What distinguishes our approach from most existing ones is that also require the runtime instrumentation of executing system which causes additional overhead. On average we measured an overhead of $\sim 200\%$ (i.e., the program executes with 1/3 the performance it normally does) which we believe is already acceptable in most situations. However, we must mention that the additional overhead may violate performance requirements (though any existing approach likely would do that also). Currently, we are working on a more efficient solution, which is not yet fully implemented but first measurements promises an overhead of only $\sim 30\%$.

5.5 Threats to Validity

We see the biggest threat to the validity of the evaluation in the chosen example applications. One might argue that the applications used are small. However, the largest system with nearly 50KLOC is far from trivial and our empirical evaluation did not reveal any scalability issues. Hence we believe that larger systems would perform similarly well. Another argument could be that the chosen case studies were not representative because they represented open source systems only. However, open source systems are commonly used for these studies and since they cover different domains we believe they are representative. Finally, one might argue that the invariants could have been even more diverse / complex. We certainly believe that it is possible to find more complex invariants. However, the 46 invariants we used were not defined by us but were available through the systems. They are thus representative of the kinds of invariants their developers cared about. Given that nearly all invariants of all three systems scaled, we believe that invariants in general would scale quite well (note that invariant scalability is a problem for all approaches and not limited to our approach).

6. RELATED WORK

Since our approach is in the category of run-time monitoring, it shares some commonalities with existing approaches. In particular it is most closely related with design by contract [1, 3] that enables invariant checks during run-time. This was first introduced in Eiffel [8], and was an integral part of the programming language. The Java Modeling Language [9] (JML) is a language aimed at providing support for documenting contracts written in Java and provides tool support for creating run-time checks of these contracts. An approach for the .NET family is provided by Microsoft [10, 11]. Contrary to our approach those approaches weave the code for invariant checks into original code, thus all contracts need to be defined prior to execution and known at compile time. Our approach on the other hand is decoupled from the original application (no code is generated at compile time to enforce invariants), enabling users to modify an invariants

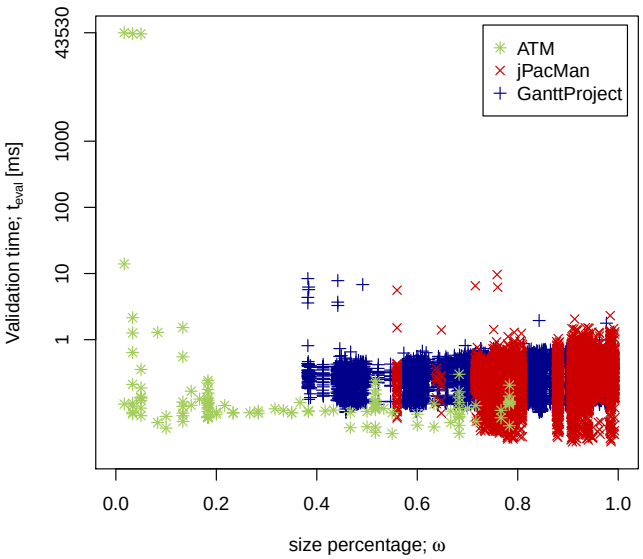


Figure 7: Validation time t_{eval} depending on number of existing objects ω

definition at run-time, add new ones, etc. Other approaches like shown in [12] do not alter the existing code, but instead make use of design patterns and generate new classes that include the invariant checks. When executing the program one may chose whether to use classes that include the checks or the ones without. Those approaches tend to be inefficient, in a sense that they may perform checks even if the state did not change (what we defined as the invariants scope). Furthermore, those approaches suffer from the flaw of local checks as well as the re-writting problem, as described in Section 2, since they only introduce the checks in the classes for which they were defined (or subclasses thereof). Our approach on the other hand observes an applications execution from a global perspective, not solely reacting to events occurring in the object itself. Our approach is thus able to handle arbitrary invariants and is not limited to localized invariants.

Run-time verification approaches also share some similarities with our approach. Usually run-time verification techniques are used to ensure a programs correct behavior [13], whereas our approach is concerned with the correct state of the underlying data structures. But, in both cases the programs are instrumented to observe events during runtime and react accordingly. In run-time verification temporal logic is used to define behavioral properties that a program has to fulfill and a sequence of events is checked whether it fulfills the given properties. The temporal logic formulas are transformed into code that represent a so called *monitor* (usually representing some kind of state machine). Events occurring during program execution are fed to the monitor. The monitor transitions into an error state indicates faulty behavior of the program. Approaches tend to use linear temporal logic (LTL), or a variation thereof, for defining behavioral properties [14, 15] or provide their own temporal logic language [16, 17, 18]. Most weave the generated monitors directly into the application code and provide instant feedback during run-time if violations occur [16, 19, 18], but there also examples of techniques that record execution traces and may be used to analyze whether the system run was correct. To some extent, run-time verification techniques may be used to check class invariants as we do in this paper (i. e. as a property that needs to be checked after certain execution steps), but, to the best of our knowledge, users need to know the concrete behavior of the program to instruct after which events the invariant checks should be executed - this is just another variation of the invariant trigger problem. Using our approach, this is not necessary, users only need to define the invariants and the algorithm will determine whether invariant checks are required or not at arbitrary points during the execution.

Finally, Rosu et al. introduced a new programming paradigm based on run-time monitoring [20, 21].. Although run-time verification approaches are usually concerned with verifying the correct behavior of a program, there are some similarities with our approach and their work shows a detailed comparison of existing approaches for run-time monitoring. They define various criteria for the various tools that differentiate target language, constraint (invariant) language, the time invariants are observed (immediately or after some post processing), or the level of intrusiveness on the observed system. Our approach follows along the line of most run-time monitoring approaches that detect invariants immediately when they occur. The other criteria are roughly irrelevant. While our prototype is implemented for Java and uses OCL constraints, our approach should not be limited to either.

7. CONCLUSION AND FUTURE WORK

This paper introduced a novel automated approach for class invariant checking that tremendously reduced the user effort required to handle invariants. Existing approaches tend to be incomplete due to local checks and error-prone due to manual invariant placement and invariant re-writing. Our approach overcomes these weaknesses by observing the system and automatically responding to execution notifications. We demonstrated that our approach scales and outperform brute-force invariant checkers. The future vision is to provide additional support for helping a developer understand and fix invariant violations in the system. We believe that the additional data we collect during an invariant's evaluation, like the method calls involved in the violation, the scope elements with their values, as well as the changes that caused the defect, is useful for debugging purposes. Furthermore, proposed fixing actions for data structures similar to the ones explained by Reder [22], can also be used for debugging purposes, as shown in [23].

Acknowledgments

The work was kindly supported by the Austrian Science Fund (FWF): P23115-N23.

8. REFERENCES

[1] B. Meyer, "Applying "design by contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[2] J.-M. Jézéquel and B. Meyer, "Design by contract: The lessons of ariane," *IEEE Computer*, vol. 30, no. 1, pp. 129–130, 1997.

[3] B. Meyer, *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[4] A. Egyed, "UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models," in *ICSE*. IEEE Computer Society, 2007, pp. 793–796.

[5] A. Reder and A. Egyed, "Model/Analyzer: A Tool for Detecting, Visualizing and Fixing Design Errors in UML," in *ASE*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 347–348.

[6] A. Egyed, "Instant consistency checking for the UML," in *ICSE*, 2006, pp. 381–390.

[7] Oracle Corporation, *Java Platform Debugger Architecture*, 05 2013. [Online]. Available: http://docs.oracle.com/

[8] B. Meyer, *Eiffel: The Language*. Prentice-Hall, 1991.

[9] G. Leavens and Y. Cheon, "Design by contract with jml," *Draft, available from jmlspecs. org*, 2006.

[10] Microsoft Corporation, "Code contracts." [Online]. Available: http://research.microsoft.com/en-us/projects/contracts/

[11] M. Fähndrich, M. Barnett, and F. Logozzo, "Embedded contract languages," in *SAC*, S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, Eds. ACM, 2010, pp. 2103–2110.

[12] B. A. Malloy and J. F. Power, "Exploiting design patterns to automate validation of class invariants," *Softw. Test., Verif. Reliab.*, vol. 16, no. 2, pp. 71–95, 2006.

[13] Y. Falcone, K. Havelund, and G. Reger, "A tutorial on runtime verification," *Summer School Marktoberdorf*, 2012.

[14] K. Havelund and G. Rosu, "Monitoring java programs with java pathexplorer," *Electr. Notes Theor. Comput. Sci.*, vol. 55, no. 2, pp. 200–217, 2001.

[15] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *ECRTS*. IEEE Computer Society, 1999, pp. 114–122.

[16] M. d'Amorim and K. Havelund, "Event-based runtime verification of java programs," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[17] H. Barringer, D. E. Rydeheard, and K. Havelund, "Rule systems for run-time monitoring: from eagle to ruler," *J. Log. Comput.*, vol. 20, no. 3, pp. 675–706, 2010.

[18] D. Drusinsky, "The temporal rover and the atg rover," in *SPIN*, ser. Lecture Notes in Computer Science, K. Havelund, J. Penix, and W. Visser, Eds., vol. 1885. Springer, 2000, pp. 323–330.

[19] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in *VMCAI*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds., vol. 2937. Springer, 2004, pp. 44–57.

[20] G. Rosu *et al.*, "Monitoring-Oriented Programming," 01 2014. [Online]. Available: http://fsl.cs.illinois.edu/index.php/Monitoring-Oriented_Programming

[21] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Rosu, "An overview of the mop runtime verification framework," *STTT*, vol. 14, no. 3, pp. 249–289, 2012.

[22] A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *ASE*, M. Goedicke, T. Menzies, and M. Saeki, Eds. ACM, 2012, pp. 220–229.

[23] M. Z. Malik, J. H. Siddiqui, and S. Khurshid, "Constraint-based program debugging using data structure repair," in *ICST*. IEEE Computer Society, 2011, pp. 190–199.