On the Effectiveness of Contracts as Test Oracles in the Detection and Diagnosis of Race Conditions and Deadlocks in Concurrent Object-Oriented Software

Wladimir Araujo
Juniper Networks and Carleton University
Ottawa, Canada
waraujo@juniper.net

Lionel C. Briand
Simula Research Laboratory and University of Oslo
Lysaker, Norway
briand@simula.no

Yvan Labiche
Carleton University
Ottawa, Canada
labiche@sce.carleton.ca

Abstract— **The idea behind Design by Contract (DbC) is that a method defines a contract stating the requirements a client needs to fulfill to use it, the precondition, and the properties it ensures after its execution, the postcondition. Though there exists ample support for DbC for sequential programs, applying DbC to concurrent programs presents several challenges. We have proposed a solution to these challenges in the context of Java as programming language and the Java Modeling language as specification language. This paper presents our findings when applying our DbC technique on an industrial case study to evaluate the ability of contract-based, runtime assertion checking code at detecting and diagnosing race conditions and deadlocks during system testing. The case study is a highly concurrent industrial system from the telecommunications domain, with actual faults. It is the first work to systematically investigate the impact of contract assertions for the detection of race conditions and deadlocks, along with functional properties, in an industrial system.**

Keywords- Design by contract, concurrency, object-oriented programming, Java.

I. INTRODUCTION

Including specifications of program behaviour together with the source code is not a new idea. Design-by-Contract (DbC) [1] is one of the most elaborate software development methodologies that put such idea in practice, with Eiffel being a well-known example of a programming language that supports it. Following DbC principles, a method defines a contract stating the requirements a client needs to fulfill to use it, the precondition, and the properties it ensures after its execution, the postcondition. Contracts can be treated as logical assertions (contract assertions) about the state of a program at a certain point. A program can be instrumented with code that checks the validity of the assertions at runtime and upon failure throws an exception indicating where it happened. DbC also defines object invariants, properties that must hold in all visible states of an object. The visible states of an object are the states just after object construction, just before a visible method execution, and just after a visible method execution. Behavioural subtyping [2-5] is an integral part of DbC. A subtype automatically inherits the specification (contracts and invariants) from its super-types [6]. The effective precondition of a method is the disjunction of all the inherited preconditions and the method's declared preconditions. The effective postcondition is the conjunction of all inherited postconditions for which the associated precondition is satisfied and the method's declared postconditions if associated preconditions are satisfied. The effective class invariant is the conjunction of all inherited class invariants with the object's declared invariant. This guarantees that a subtype can be properly used in place of its super-type(s).

The Java Programming Language [7] does not provide native support to DbC. It only provides basic support for assertions through the assert keyword, which simply causes an exception to be thrown in case a given Boolean expression evaluates to false. This work uses the Java Modeling Language (JML) [8, 9] as the specification language used to write contracts. JML includes notations for pre- and postconditions, invariants, and offers mechanisms for specification inheritance, thus providing support for the Design-by-Contract paradigm. JML has a Java-like syntax and specifications can even perform method calls in assertions. It also provides a rich set of model classes (i.e., classes allowed only in specifications) that enable the construction of rich abstract descriptions of program behaviour, such as data structure model classes, which can be used to model abstract properties of concrete data structures in a concise way.

The JML toolset comes with a compiler [10] that translates specifications into runtime assertion checking (RAC) code producing Java classes augmented with executable assertions. The process of adding RAC code to a Java class is called instrumentation. The resulting class is called the instrumented class. The JML compiler [10] produces RAC code that enforces behavioural subtyping, i.e., RAC code for all applicable invariants and preconditions is executed upon entering a method, and RAC code for all applicable postconditions and invariants is executed upon exiting a method.

Most work on DbC focused on sequential programs, and applying DbC to concurrent programs presents several challenges. The first challenge is interference, the product of multiple threads of execution modifying and accessing shared state. Interference is present even on correct programs with respect to concurrency control. Basically, interference with respect to the precondition happens because assertion checking code is evaluated at a point in time after which other threads are allowed to modify the objects referenced in such assertions but prior to the point in which these objects are accessed by the method in question. This causes RAC code to report errors for correct methods and vice-versa. The problem is analogous with respect to postconditions and invariants [10, 11]. The second challenge is the specification

and verification of locking related properties using contract assertions (the use of locking policies is a common deadlock avoidance technique). The third challenge is the specification of thread-safety properties in the presence of inheritance. These properties state which objects are safe to be accessed by the currently executing thread, i.e., there are no other threads accessing such objects. Locking requirements had so far been associated with preconditions. This causes problems. These challenges and their solutions are described in detail in [11] and summarized in section B.

This paper focuses, based on a large scale industrial case study, on assessing if and in which conditions contract based assertions are good replacements for manually coded test oracles for concurrent systems. Given the space restrictions, we address in this paper only the aspect related to concurrent faults even though the study is conducted with contracts including specifications for functional properties.

In our case study, we systematically apply DbC to a highly concurrent industrial system and measure the effectiveness of contract-based assertions at *detecting* and *diagnosing* race conditions and deadlocks, two important types of concurrent faults. We conclude that contracts are very good at detecting such faults. Moreover, contracts are extremely helpful in reducing the diagnosis effort for the faults they detect since faults are located at most one method away from the detection point. To the best of our knowledge, this is the first study to systematically apply contract assertions that combine the specification of functional and concurrent properties to the detection and diagnosis of race conditions and deadlocks in a highly concurrent industrial system and under realistic conditions (actual faults, realistic test suites).

The following section discusses related work. It is followed by a brief introduction to JML, a summarized description of the challenges in applying DbC to concurrent programs together with our solutions, and an empirical assessment of the validity of the use of RAC code in a concurrent environment to conduct system testing. Section IV describes the case study with attention to the methodology and reports the main results of this work. We conclude with a summary and future work.

II. Related Work

Verification of concurrency properties of programs can be divided into three kinds of approaches. Static checking uses the source code only (usually augmented with some annotations) to check the validity of certain properties. Dynamic checking uses only information available during runtime execution of the program under test. There are also approaches that combine both techniques. Our work concentrates on dynamic checking.

Flanagan and Freund [12] describe Atomizer, a dynamic checker for Java programs. Atomizer checks for method atomicity. Agrawal et al. [13] describe a combination of runtime and static analysis to check for atomicity. Atomicity checking relies on annotations provided by the programmer to determine the set of locks protecting access to a variable (which is possibly flawed) or the lock inference algorithms used in their place require multiple executions of a method (or block of code in general) being checked for atomicity to make a determination, which does not fit well with a RAC-based approach to verification, in which a predicate is expected to yield an answer in every execution. Atomicity is to be established prior to executing functional contracts. Nevertheless, interference can cause a contract to evaluate erroneously even in atomic methods. Therefore these solutions do not contemplate the joint dynamic verification of functional and concurrent properties. A similar problem happens with pattern-based concurrent bug detection, as reported by Park et al [14]. Furthermore, they only report results for small programs or for programs with a small number of threads (< 20).

Rodríguez et al. [15] describe a variety of constructs in JML for dealing with several aspects of concurrency properties. They present solutions to the problem of specifying lock acquisition and thread-safety properties but fail to consider the issue of inheritance. Although they propose several constructs, none of them were actually implemented in the JML toolset and assessed empirically. We implement all the constructs we propose on the JML compiler and generate RAC code for them [16].

Jacobs et al. [17] present a methodology based on object and thread ownership in which a thread must own an object to access any of its fields. This implies that preconditions and postconditions only refer to thread-safe fields. In other words, the internal behaviour of the object cannot be specified in several important cases.

Nienaltowsky and Meyer [18] present an interesting proposition regarding the use of contracts in a concurrent environment. They target SCOOP [19], an extension of the Eiffel language to provide support for concurrency. The SCOOP model prevents data races by design but does not address deadlocks. They do not consider specification inheritance nor conduct any experiments or case studies.

Greenhouse et al. [20] describe a series of annotations related to the specification of the concurrent behaviour of a Java program. Their annotations are similar to those present in [15] with respect to locking properties and member ownership, and thus suffer from the same limitations. They do not present a specialized construct to state the thread-safety of an object. They do not present a solution to the verification of functional properties in combination with concurrency related properties.

Qadeer and Wu [21] describe a technique to translate a concurrent program into a sequential program, which is, then, analyzed by a checker to detect data races. Their approach has been applied to multithreaded C programs; ours focuses on object-oriented programs. They focus on data races only, whilst our approach covers deadlocks as well. Their approach does not allow a developer to specify what objects are expected to be thread-safe in which conditions, which leads to false positives.

Elmas et al. [22] describe VYRD, a tool to detect data races based on a trace refinement technique. A concurrent execution must be a refinement of a trace specification. Flanagan and Freund [23] describe FastTrack, a precise dynamic race detection algorithm based on Lamport's *happens-before* memory access relation. Their system works

by instrumenting Java bytecode to record an event stream of memory and synchronization operations for offline analysis. They apply it to the Eclipse framework and report detecting real faults. The fault selection and injection procedure is not described precisely. Ratanaworabhan et al [24] describe ToleRace, a system to detect and tolerate data races. The detection mechanism is similar to ours [16] but their tool does not require annotations. All the above works do not address the verification of functional properties and do not consider inheritance.

In [25] Le Traon et al. describe how to use contracts to generate assertion code. The authors propose metrics to evaluate the benefits of instrumenting contracts. They define vigilance and diagnosability and apply them to several case studies. The experiments are, however, limited to small programs in which faults are introduced via program mutation [26]. Briand et al. [27] clarify the concept and metric of observability (as a replacement for vigilance) and diagnosability. Although carefully designed, their experiment is performed on a small system through mutation analysis. Both studies are restricted to sequential software.

To the best of our knowledge, no existing work reports on an industrial case study where a unified solution to the specification and dynamic verification of concurrency and functional properties is rigorously assessed in terms of concurrent fault detection and diagnosis, and under realistic industrial conditions.

III. CONCURRENT CONTRACTS

This section begins with a brief description of JML and its fundamental constructs. It then briefly describes the challenges in applying DbC to concurrent programs together with our solutions, describing the constructs we introduced.

A. The Java Modelling Language

The Java Modeling Language (JML) has a Java-like syntax and specifications can even perform method calls. JML specifications are delimited by the strings `/*@` and `@*/` or by the remainder of lines following `//@`, being treated as comments by the Java compiler. Specifications can be written as annotations in '.java' files (javadoc style).

In JML, the interface of a method is specified through a set of clauses. The most relevant for this study are:

- **`requires`**: specifies the conditions that need to be satisfied by the method caller.
- **`ensures`**: specifies the properties that this method guarantees to its caller.
- **`when`**: specifies an enabling condition (the method blocks until this condition is met).
- **`signals`**: specifies a condition that is guaranteed to hold if a given exception is thrown.
- **`signals_only`**: constrains the exceptions that can be thrown when a condition for exceptional behaviour is satisfied.
- **`normal_behaviour`**: specifies the conditions in which a method returns normally and what it ensures.
- **`exceptional_behaviour`**: specifies the conditions in which a method throws an exception.

Invariants are specified using the **`invariant`** clause. Invariants must hold in any publicly visible state, i.e., prior to and after the execution of any instance methods. JML provides a rich set of native operators for defining complex specifications, the most relevant for this study being:

- `\old(e)`: used in post-conditions to refer to the value of expression e in the pre-state of the method.
- `\return`: the return value of a method. Its type is the same as the method return type.
- `\lockset`: returns the set of locks held by the current thread.
- Operators `<` and `<=`: used to test the order of lock acquisition. A lock is greater than another if it was acquired later.
- `\max(s)`: returns the largest lock in set s according to the ordering defined by the operator above.

B. Contracts and Concurrency

In previous work [11, 28], we solved the problem of interference by combining the use of safepoints with thread-safety requirements. A safepoint is a point inside the method body at which it is safe to evaluate precondition or postcondition predicates together with invariants. Fig. 1 shows an example of their use. The method specification is composed of two specification cases separated by the keyword **`also`** (each with a precondition and the corresponding *expected postcondition*, the postcondition to be established if the precondition is satisfied), which simply tell that the head of the list will move to the next element and the method will return the value of what used to be the first element of the list if the list is not empty (lines 5-9), and returns **`null`** otherwise (lines 1-4). In JML, the preconditions of a method (i.e., the **`requires`** clauses), as well as arguments to the `\old` operator in postconditions are evaluated in the method's *pre-state*. The method postconditions (i.e., the **`ensures`** clauses) are evaluated in the method's *post-state*. "The *pre-state* of a method call is the state just after the method is called and parameters have been evaluated and passed, but before execution of the method's body. The *post-state* of a method call is the state just before the method returns or throws an exception; in JML we imagine that `\result` and information about exception results is recorded in the post-state" ([8], p. 8).

Although straightforward, this specification is not correct in a multi-threaded environment without safepoints. Suppose that `extract()` is invoked by thread 1 and in the method's pre-state, `head` references the same object as `last` (i.e., the list is empty). Suppose, also, that thread 2 pre-empts thread 1 right after thread 1 acquires the lock on this to fully execute method `insert()`, which does not acquire such lock for performance reasons. The postcondition of `insert()` specifies that `head` is not referencing the same object as `last`, i.e., the list is not empty. Once thread 1 resumes execution and acquires the lock on `head`, it will return the first element of the list, violating the postcondition of `extract()` for an (expected) empty list, i.e., that it should have returned **`null`**.

```
public class LinkedQueue {
    protected /*@ spec_public @*/ LinkedNode head;
    protected /*@ spec_public @*/ LinkedNode last;
    //@ public invariant head.value == null;
1   /*@ public normal_behavior
2     @  requires head == last;
3     @  assignable \nothing;
4     @  ensures \result == null;
5     @ also public normal_behavior
6     @  requires head != last;
7     @  assignable head, head.next.value;
8     @  ensures head == \old(head.next) &&
9     @    \result == \old(head.next.value);
10    @*/
11    public synchronized  Object extract() {
12        synchronized (head) {
13            //@requires_safepoint:
14            Object x = null;
15            LinkedNode first = head.next;
16            if (first != null) {
17                x = first.value;
18                first.value = null;
19                head = first;
20            }
21            //@ensures_safepoint:
22            return x;
23        }
24    }
25}
```

Figure 1. Method `extract()` of class `LinkedQueue` using safepoints to avoid internal interference.

This is an example of interference in the context of DbC. This problem is not specific to Java or JML. Any object-oriented language in which the scenario we described above is realizable and provides support for DbC via runtime assertion checking (RAC) is prone to this problem. It is important to emphasize that such problem arises due to the combination of DbC and the program under execution. It is not due to erroneous concurrency control on the part of the implementation either of the client or the provider. The case, as above, where interleaving occurs inside the method body is called *internal interference*. Interference can also happen between the contract evaluation points (pre- and post-state) and the method entry and exit points. Since interleaving occurs outside the method body, this is called *external interference*.

A *safepoint* is any point inside the method body where it is safe to evaluate precondition, postcondition and invariant predicates. A *precondition safepoint* is a point where it is safe to evaluate preconditions and invariants, and the pre-state expressions of postconditions. A *postcondition safepoint* is a point where it is safe to evaluate the expected postconditions and the invariants. Any method execution path (from the pre-state to the post-state) can have only one precondition safepoint and only one postcondition safepoint to maintain the semantics of DbC as for sequential software. If no precondition (resp. postcondition) safepoint is explicitly specified for an execution path, it defaults to the method pre-state (resp. post-state). In a precondition safepoint, all preconditions, invariants and pre-state expressions are required to be safely evaluated. In a postcondition safepoint, the postconditions and all invariants are required to be safely evaluated. The **`requires_safepoint`** and **`ensures_safepoint`** labels demarcate those safepoints. At the precondition safepoint in Fig. 1 (line 13), all the objects referenced by both **`requires`** clauses (lines 2 and 6) and the contents of the `\old` statements in the **`ensures`** clauses (lines 8-9) are properly protected by locks. At the postcondition safepoint (line 21), the field `head`, present in the **`ensures`** clause at lines 8-9, is properly protected by a lock. Since `\result` refers to local variable `x`, which in turn points to an object no longer referenced by the list, it is also thread-safe at the postcondition safepoint. Finally, the object invariant can be safely evaluated both in the pre- and postcondition safepoints since it refers to `head`, which is properly locked in both places. The postcondition safepoint must be the **`return`** or **`throw`** statement. Additionally, the return (or throw) expression must be side-effect free.

We also solved in [11] the issue of thread-safety specification by detaching these properties from preconditions while considering interference and inheritance issues. Thread-safety properties are specified using the **`requires_thread_safe`** and **`ensures_thread_safe`** clauses of a method specification. Such clauses specify a set of objects to which access is required to be thread-safe. An object is considered to be thread-safe if it is local to the current thread (i.e. no other thread has a reference to it) or access to it is protected by a lock. Thread-safety properties can also be specified by referring explicitly to the locks a method must or must not hold before or after its execution via the **`requires_locked`**, **`requires_unlocked`**, **`ensures_locked`** and **`ensures_unlocked`** clauses, respectively. Fig. 2 shows an example of their use (lines 8-10) in combination with safepoints. The **`requires_thread_safe`** clause specifies that object `r` must be thread-safe in the method pre-state. This is necessary because the effective precondition, accounting for normal and exceptional behavior of the method is `r.isRequest()` (the disjunction of preconditions from both specification cases simplifies the terms `connected` and `!connected`), which is not simply **`true`**. In this situation, safepoints alone cannot guarantee the thread-safe observation of this predicate since `r` is external to the provider. Once such object is thread-safe, predicates involving it can be checked at precondition safepoints since they will not change between the method pre-state and the safepoints. A similar discussion can be made for postconditions and thus the **`ensures_thread_safe`** clause specifies that the object returned by the method must be thread-safe on the method's post-state. The **`*_thread_safe`** clauses guarantee freedom from interference with respect to `r` from the method pre-state up to the precondition safepoint and with respect to `\result` on the post-state. Precondition safepoints prevent interference related to model (i.e. specification-only) field `connected`. As these are the only possible sources of interference, we conclude that combining safepoints and thread-safety predicates guarantees `sendAndWait()` and its contract are interference-free. In general, the combination of thread-safety requirements on data to be observed by the provider and the client with safepoints (for safe evaluation of predicates referring to internal state) is required to guarantee freedom from interference.

The semantics of specification inheritance on the concurrent aspect, i.e., the clauses defined under the **`concurrent_behaviour`** construct (line 8), is identical to the one of invariants (conjunction). The effective specification (in a subclass) of any of the new clauses is the

union of the argument set specified on the target object with the argument sets of its immediate supertypes. In other words, thread-safety specifications, like invariants, can only be strengthened by sub-types. Decoupling concurrency related properties from functional properties gives concurrent contracts their intuitive (expected) meaning. A complete argument is presented in [11, 28].

We also addressed in [28] the problem of lock acquisition order specification with the introduction of the **`lock_order`** clause to the specification of a type (i.e., it is analogous to an invariant). This clause takes a list of lock order expressions that must be satisfied. A lock order expression is in the form `l1 < l2` or `l1 <= l2` where `l1` and `l2` are instances of either `java.lang.Object` (for monitor locks) or `java.util.locks.Lock` (for the semaphore style ones). These expressions evaluate to **`true`** if the current executing thread acquires `l2` only after acquiring `l1`. The semantics of the **`lock_order`** clause is that each lock order expression must hold for every state it is in effect in the context of the current thread. A **`lock_order`** clause is *in effect* for a given state if such state is in the activation record of a method belonging to the type declaring such a clause or one of its subtypes. A clause that is in effect is evaluated at every attempt of the current thread in acquiring a lock.

IV. DETECTION AND DIAGNOSIS OF CONCURRENT FAULTS

The objective of this study is to evaluate the applicability of the concurrency related constructs in contracts as defined in section B. More specifically, if contract assertions can be effective test oracles to detect and diagnose concurrency related faults. The study is limited to detecting and diagnosing race conditions and deadlocks since available JML constructs do not support the specification of liveness or fairness properties. This section begins by reviewing the concepts of observability and diagnosability. It follows with a description of the test bed used to conduct this study. It then addresses the issue of the validity of the results obtained using RAC code in concurrent programs during system testing in place of the original (non-instrumented) program. The methodology is described next and then the results reported.

A. Background on Observability and Diagnosability

This section only presents the definitions and some basic facts on the concepts and measures of observability and diagnosability. For a complete exposition, see [25, 27].

The *observability* of a system (also called global observability) composed of a set of interconnected components is defined as the probability that a fault internal to a component is detected in the component itself (e.g., through assertion violations) or in any one of the other components.

Diagnosability is defined as the ease with which the causes of a failure can be isolated. It can be measured based on an estimate of the size of the diagnosis work to be done by measuring the *distance* between the location of the failure detection and the location of the faulty statements that caused it. Such distance can be defined as the number of methods investigated beginning at the detection point (where the failure occurred) to the location of the faulty statement according to a diagnosis flow (see below). This, like any model, is a simplification of the reality since expert developers frequently use shortcuts to diagnose a failure. Such simplification, however, is necessary to perform a systematic, objective study of diagnosability.

```
/*@
1   normal_behaviour
2    requires connected && r.isRequest();
3    ensures \result.isResponse();
4   also
5   exceptional_behaviour
6     requires !connected && r.isRequest();
7     signals_only NotConnectedException;
8   concurrent_behaviour
9     requires_thread_safe r;
10    ensures_thread_safe \result;
 */
public Message sendAndWait(Message r) throws … {
11   synchronized(in) {
12       synchronized(this) {
13           //@ requires_safepoint:
14           if(closed || remoteClosed)
15               throw new NotConnectedException();
16       }
17       out.put(r);
18       return in.get();
19   }
}
```

Figure 2. Method declaration exemplifying the use of thread-safety specification clauses.

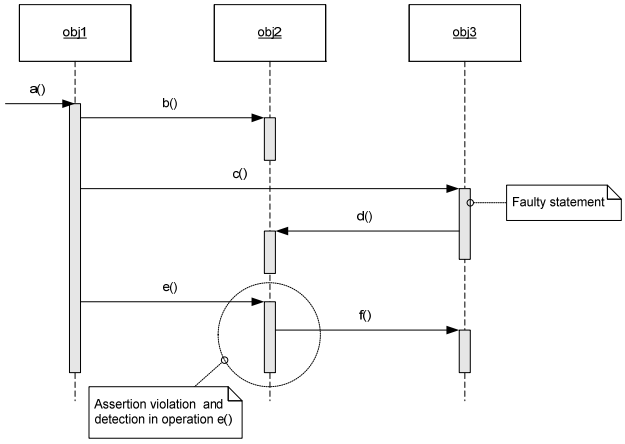

Figure 3. Diagnosability measure example: diagnosis flow as a sequence diagram.

The starting point of the diagnosis is the method in which the failure was detected. It is the caller of a method that had its precondition violated or the method that had a post-condition violation. The search proceeds then from the beginning of the method in which the fault was detected, recursively exploring all the methods called until the fault is uncovered or the end of the method is reached. In the latter case, the search proceeds to the caller method. A method is assumed to be investigated only once. The method to be investigated is determined based on the dynamic type of the target object, not its static type. A method call is not explored if it is certain that the particular execution path leading to the fault did not execute such method based on

method arguments and structural constraints. For instance, only one of either an if statement or an else statement needs to be investigated provided the conditional expression refers to only method parameters, their values are not changed by the method under investigation, and their values are known. It is explored otherwise. For instance, assuming an assertion violation occurs in the precondition of method e() in Fig. 3, the diagnosis flow is then the sequence [a, b, c] (method d() is not inspected since the faulty statement is discovered in a statement preceding its invocation). The distance is then 3. An assertion violation occurring in the post-condition of method e() would yield the diagnosis flow [e, f, a, b, c], instead, and thus a distance of 5.

B. Target System and Test Bed Setup

The target system is the Service Activation Engine (SAE) component of the Session Resource Controller product line of Juniper Networks. It is basically a platform to design and deploy value-added services in an Internet Protocol network. It does so by converting service definitions specified as an abstract set of traffic controlling policies for a particular subscriber into device specific policies in the context of the interface such subscriber uses to connect to the network. The SAE currently supports various devices.

Our empirical study focuses on the subsystem that interfaces with Juniper's E-series routers. This subsystem, called the router driver, is responsible for responding to asynchronous notifications from the router regarding the state of each subscriber interface and managing traffic policies for each such interface. Due to the large number of subscribers a router supports, these requests are processed concurrently to maximize system performance. The router driver is responsible for the translation task above, the low-level communication with the router and to ensure correctness in the presence of concurrent processing. It does so by implementing a transactional infrastructure to guarantee ACID (Atomicity, Consistency, Isolation, and Durability) properties of transactions. The SAE is capable of managing approximately 520,000 active subscribers connected to multiple E-series routers. This amounts to executing approximately 1,500 transactions per second. The complex functionality of the router driver subsystem allows the use of complex functional specification constructs, and its high degree of concurrency with varied and intricate concurrency control patterns allows for all proposed constructs to be explored. With respect to code size, the router driver subsystem is composed of 54 classes and interfaces (33509 LOC), all of which are used in a concurrent environment. In many ways this can be considered a representative concurrent system in the telecom domain.

The standard test suite is an automated test suite composed of a sufficiently large number of test cases that are required to pass for a version of the SAE component to be released to production. Each test case exercises the SAE through its interfaces and the test case oracle (embedded in the test case) checks return values, parameters and exceptions of operations in its programmatic interface against expected values. It also checks the presence or absence of expected contents in the log files produced by the SAE, such as error messages related to the operation performed.

The test suite is built using a black box approach based on test plans derived from functional specification documents of SAE's features. Specific size and coverage parameters of the standard test suite are confidential information of Juniper Networks. However, the key property for this study can be stated: for every defect present in the defect database there is at least one test case in the standard test suite that exercises the fault caused by such defect thus causing it to manifest as a failure in the production system or as an assertion violation in the instrumented system if the fault is observable by the contracts. The test suite is executed in an environment that mimics production environments.

The scripts composing the standard test suite take as input several parameters that impact the load imposed on the overall system. Some of these parameters are the rate at which subscribers log in(out) to(of) the network, the total number of subscribers and types of services such subscribers have. All these parameters are abstracted as a *load factor* due to their confidential nature. The only property of interest for the experiments concerning the load factor is the ratio between them, i.e., if the load factor in one execution is double the value of another's then the overall load the first execution imposes on the system is double the other's. The load factor represents mainly the throughput of the system.

C. Runtime Assertion Checking

We modified the JML compiler to generate RAC code for the new constructs described in the previous section as well as for the existing constructs to enable their execution in a concurrent environment. A complete description of such modifications is presented in [16]. In this section, since this paper's main contribution is an industrial case study, we focus on investigating whether a program instrumented with RAC code is a valid replacement of the original program during system testing.

The target system is specified following the methodology described in section D. Ideally, the instrumented system should present the same external behaviour as the production system. We name this factor *indistinguishability*. The instrumentation techniques introduce extra processing steps and require more data to be stored for the purpose of verifying the validity of the contracts. Therefore, the instrumented system is expected to consume more resources (CPU, memory and persistent storage) than the production system. There should be a linear relation between the resource utilization of the original and instrumented versions to guarantee similar behaviour between them. We name this factor *runtime overhead*. These two factors are considered independently. Achieving satisfactory results, as described below, in both dimensions would allow us to conclude that an instrumented version of a concurrent system can be used, under practical conditions, during system testing to uncover faults.

1) Runtime Overhead

Program size is measured in two ways: class file size and permanent generation size. The class file size is the number

of bytes of the bytecode representation of a Java class or interface. The purpose of this metric is to determine the increase in the amount of persistent memory necessary to hold the instrumented program uncompressed. The permanent generation is the area of runtime memory of a JVM dedicated to holding the runtime representation of a class or interface. The purpose of this metric is to determine the increase in the amount of runtime memory necessary to load the class file into memory. Both metrics are important to understand the system requirements to execute the instrumented program in conditions equivalent to the original system.

Analyzing the incremental class file size (the size of the instrumented class file minus the size of the production class file) as a function of the number of methods in each class enables the derivation, through linear regression, of the following formula to determine the total class size (in kilobytes) of the instrumented version of the system based on the number of methods of the classes in the production version of the target system:

$$IncrementalClassSize(racCompiledClasses) \approx 8methodCount(racCompiledClasses) + 9|racCompiledClasses|$$
$$methodCount(\{C_1, \ldots, C_n\}) = \sum_{i=1}^{n} methodCount(C_i), \quad (1)$$

where $racCompiledClasses$ is the set of classes compiled with the JML compiler, and $methodCount(.)$ returns the number of methods in a given class. $IncrementalClassSize$ returns values in kB. The average magnitude of relative error observed between the calculated and the observed incremental class file size was 10%. The spearman correlation coefficient between the incremental class size and the number of methods was 0.99 indicating a strong dependency between them and, consequently, a small dependency (i.e., less than 1%) on the contents of the contracts and other unaccounted factors.

We also measured the increase in memory footprint (heap) and CPU utilization. The heap is the memory area in the JVM where all dynamically allocated objects reside. The heap needs to be carefully dimensioned to avoid not only failures in allocating objects but also to avoid excessive load on the JVM garbage collector (GC), which would cause a significant increase in CPU utilization, possibly reducing the amount of cycles available to execute the application itself.

Fig. 4 compares the heap utilization between the production and the instrumented versions of the target system subject to the same test suite. This is the actual system test suite and it consists of a ramp-up phase, in which subscribers are logged into the system (approximately the first 20 minutes), and a steady-state phase in which several operations are performed (the remaining time). The duration of the test is about two hours. The overall load factor was chosen as to cause 30% CPU utilization during the steady-state phase of the suite for the instrumented version of the system. This was necessary to guarantee that the CPU utilization would almost never go beyond 80% to ensure that CPU was never a contention point, thus preventing significant periods during which the CPU utilization is 100%. During such periods, there would be threads that could be scheduled to run but could not get cycles. This prevents the analysis of the relationship of the CPU usage between the instrumented and production versions of the system.

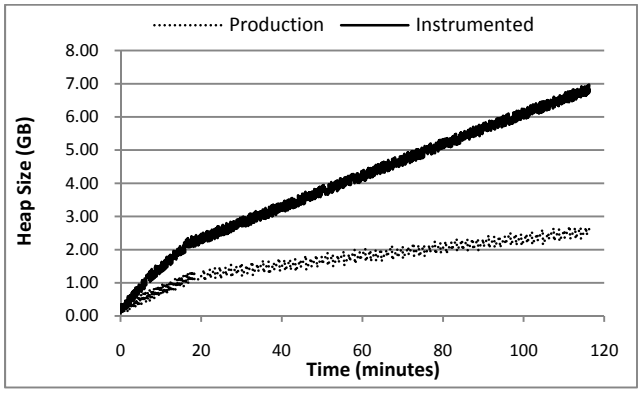

Figure 4. Comparison of the heap utilization between the production and the instrumented versions of the target system subject to the same test suite.

There is no easy formula to determine the ratio of heap utilization between the original and instrumented systems based on static parameters as with the permanent generation size, which is approximately 3.9 times the incremental class size. This is because several factors like the number of instances of each class as well as the number of threads in the system, which are essentially dynamic parameters, cannot be determined in a generic fashion for all applications since they depend on each application's structure. The important property, however, is that such ratio is bounded. Notice from the graph that although the heap size grows steadily in both versions, the ratio between them is almost constant. After the 20 minutes mark, the heap size of both versions grow linearly at a rate of 46.8MB/minute for the instrumented version and 13.5MB/minute for the production version; i.e. the instrumented version consumes heap at a constant rate of 3.47 times the production version. That is, there is a linear relation between the two versions of the system regarding their resource usage. This suggests that the instrumented version is presenting the same behaviour with respect to resource utilization as the production version whilst requiring more resources. It should be noted that, although the heap grows steadily, the system does not present a memory leak. The configured heap size is significantly higher than the range displayed in Fig. 4 and, therefore, the JVM does not attempt to collect garbage aggressively.

The overall CPU load that the instrumented version of the target system can handle is approximately 17.5 times smaller than that of the production system (based on the load factor in section 0). The immediate effect of this slowdown is on time-sensitive activities. This result implies that any absolute value used to detect improper behaviour like timeouts needs to be increased 20-fold to accommodate delays in the processing due to the instrumentation overhead, which typically consists of changing configuration properties.

Though the impact of instrumentation is expectedly significant, for example on the amount of CPU required to execute the instrumented code, we have seen that the load factor can be adjusted to guarantee resource usage conditions by the instrumented version similar to those of the production version. As a result, the amount of extra heap, program storage and permanent generation required by the runtime is kept at reasonable bounds thus allowing instrumented applications to be executed in the same environment used by the production version. Overall, the runtime overhead introduced by the instrumentation activity is deemed, from a practical standpoint, reasonable.

2) Indistinguishability

Indistinguishability is analyzed based on the behaviour of an instrumented version of the target system compared to its production version. Success is defined by the following:

1. The instrumented and the production versions of the target system pass the standard test suite.
2. Faults introduced in both versions cause the same failures when submitted to the standard test suite, i.e., the test cases that fail and succeed are identical for both versions.
3. A qualitative analysis of the instrumentation techniques give a high degree of confidence that thread interleavings present in the production version are also present in the instrumented version. This is omitted for lack of space. It can be found in [16].

To verify that both versions of the system behave the same in the absence of faults, contracts for the system were defined taking the behaviour presented by the code at a particular released version of the Service Activation Engine as correct (see section D and [11] for details). This became the instrumented version of the system. The standard test suite described above was then executed for two hours (the standard duration) and no (contract) assertion violations were observed.

To verify that both versions of the system behave the same in the presence of faults, the instrumentation was performed using a special compilation option that causes an error message to be printed on the error console (i.e. stderr on UNIX systems) instead of causing an assertion failure (the system is stopped manually after each failure, instead). This approach allows a fault to manifest itself identically in the instrumented and the production versions of the system. The faults introduced originate from Juniper's defect database and are related to the router driver subsystem. Moreover, the faults considered are only those found during the system test phase of the product and only through a period for which the feature set of the subsystem remained the same (i.e., contracts did not change). This amounts to a total of 139 faults split between functional and concurrent (deadlock and race conditions). All faults were reproduced in both versions of the system submitted to the standard test suite. Using functional faults increases the coverage of RAC functionality enabling more general conclusions than if restricted to concurrent faults.

There is still the question of timing, which is fundamental to race conditions. Given the instrumentation process causes significant slowdown to tasks (see section 1)) there is the possibility of some race conditions to be uncovered as well as others to be hidden. The only solution for this (in the context of testing) is to let the system run for a sufficiently long time performing a sufficiently varied set of tasks, which is the same approach used for uncovering race conditions and deadlocks in production systems. Therefore the instrumentation does not change the testing procedure with respect to uncovering concurrent faults.

The instrumented system presents the same observable behaviour as the production system since the outcome of the standard test suite to which both versions are submitted is the same in the presence and absence of faults, and that no thread interleavings present in the production system are artificially removed by the instrumentation process. Therefore, the instrumented system is deemed indistinguishable from the original for system testing.

D. Experimental Methodology

Contracts are specified for all methods of the classes and interfaces of the target system to the maximum extent possible and without modifying the code. This restriction is of fundamental importance to obtain realistic results as in practice the code would not be modified to facilitate contract specifications at the expense of performance or simplicity. An example would be increasing the scope of a lock by covering more statements in the method body to satisfy thread-safety requirements so that a more precise predicate can be stated. Doing so has the potential of impacting the performance of the system. In total, 1536 methods were specified with contracts containing concurrent facets. Each contract typically contains two clauses, for both pre- and post-state predicates. There is, on average, 1 contract per 17.6 LOC.

The target system with contracts is compiled with the RAC compiler and is then called the instrumented system. All contracts were designed prior to executing any experiments, including fault selection (see below), to avoid biased results.

The observability and diagnosability of the instrumented version of the system is measured using injected concurrency related faults, which may be detected through assertion violations. The faults to be injected are real and retrieved from Juniper's bug database according to the following criteria:

1. It is a concurrency related fault (deadlock or race condition)
2. It is reproducible in the production system (i.e. detected by the standard test suite)
3. It was originally discovered during system testing
4. It is located in the router driver subsystem or on a directly connected client so that the failure is detected due to the erroneous behaviour of the router driver subsystem
5. It was originally discovered during a period of time in which no significant new functionality was added to the router driver subsystem.

Points 1 and 2 above are self-explanatory. Point 3 is necessary to exclude faults reported by developers during development as our focus is system testing. Such faults are discovered during coding and developers have the habit of filing reports to keep track of their development activities.

Point 4 is necessary to limit the scope of the study to the subsystem we selected for our empirical work and keep the effort of the study to a reasonable level. Faults located in directly connected clients are eligible since some locks need to (or must not) be acquired prior to executing operations in the router driver subsystem. It is expected that such faults be detected by the contracts of the methods in the interface objects. Point 5 is required so that the contracts used to specify the subsystem remain valid (i.e., they do not need to be changed) in order to inject a fault present in an earlier version of the system. This is not merely a matter of effort in contract updating but a requirement to allow for the proper analysis of the results: the target system remains the same throughout the experiment, with the exception of the injected fault.

The experimental procedure is as follows:

1. Select a fault satisfying the criteria above and inject it in the instrumented and the production versions of the system.
2. Run both versions of the system through the standard test suite; the instrumented version should execute in a non-fatal assertion checking mode. If both versions of the system exhibit failures on the same test cases, proceed to step 3. Otherwise go to step 4.
3. Run the instrumented version of the system through the standard test suite in regular mode (i.e., with assertion violations reported via thrown exceptions)
   a. If an assertion violation occurs, register the occurrence and calculate the distance between the violated contract and the fault and go to step 4.
   b. If an assertion violation does not occur, update the contracts, if possible, to detect the fault and restart step 3.
   c. If it is determined that the fault cannot be detected through a contract violation, record this occurrence and go to step 4.
4. Go to the next fault and go to step 1. If there are no more faults, stop.

The decision to retrieve faults from the bug database serves two purposes: it eliminates the human factor in the fault selection process and it ensures that the faults are representative of realistic faults.

There is still the risk that such faults do not represent the complete spectrum of possible types of concurrency related faults. However, given the complexity of the system under test with respect to concurrency control (i.e. a transactional system responding to asynchronous events from devices and users with a high degree of parallelism), the fact that the system has been through multiple releases to a variety of customers and is operational in several networks supporting many different scenarios, it is reasonable to state that the vast majority of faults in the system have already been found. This conclusion is only possible because the feature set of the system under test did not change over the period (releases) in which the faults were discovered (see point 5 of the selection criteria above).

Regarding the iteration in step 3.b above, it is an error free task to modify a contract (or a set of contracts) to detect a specific fault since the correct system in combination with the standard test suite can be used to determine the validity of the contract. Such iteration will enable the determination of an upper bound in contract fault detection effectiveness though in practice we can expect the effectiveness to be lower, to an extent depending on the developers' skills.

Success is defined by the ability of the instrumented system in detecting the injected faults and by the ease in diagnosing it. The first factor is measured by the system observability and the second by the size of the diagnosis effort in terms of the distance between the fault and the contract that detected it. The higher the observability and the lower the diagnosis effort, the more successful contracts are as test oracles for concurrency related faults.

E. Results

A total of 10 faults satisfied the experimental criteria defined above. All faults were detected by contracts, thus amounting to 100% observability. Table I summarizes the results. Race condition faults are detected by method specification clauses in the concurrent facet. Faults caused by lock ordering issues are detected by the `lock_order` type specification clause.

TABLE I. SUMMARY OF CONCURRENT FAULTS DETECTED BY CONTRACTS WITH AND WITHOUT UPDATES.

	Contract unchanged	Contract updated
Race condition	6	0
Lock order	2	0
Race condition & lock order	1	1

Table I allows us to derive two important conclusions. First that the vast majority of concurrency related faults (80%) are related to race conditions since only 20% of them are exclusively associated with lock ordering issues. One must notice that this does not mean that deadlocks represent only 20% of concurrency related faults since deadlocks can be caused by race conditions in the evaluation of wait conditions. This only means that the effort to specify race condition related behaviour is significantly more likely to offer a better return in terms of fault detection than the effort spent on specifying lock ordering behaviour. The second conclusion we can draw is that concurrency related contracts written by a well-trained person (the first author was responsible for designing and implementing the majority of the subsystem under test) are rarely incorrect or incomplete since only one fault required a contract to be updated to enable its detection. This is likely due to the simplicity of the concurrency clauses (compared to functional clauses) since one simply specifies if an object is expected to be thread-safe or if a lock is expected to be (or not to be) acquired by a thread as well as having a separate facet dedicated for such clauses. This allows us to conclude that the likelihood for contracts written based on design information to detect faults is very high (90% in this study).

Regarding diagnosability, all faults have a distance measure equal to 1, meaning that a fault is either located immediately preceding the detecting contract or in the same method that detected the fault in its post-state, in such a way

that no other methods needed to be investigated to determine the cause of the fault. A typical example of the former case is missing to acquire a lock via a **`synchronized`** block prior to calling the method with the detecting contract. A typical example of the latter is missing to make a method **`synchronized`**. This result may seem surprising as one would expect at least some cases of nested method calls with the innermost method contract detecting the failure of the outermost method in acquiring (or ensuring the release of) a lock to occur. Such cases would have a diagnosis effort with distance greater than 1. This is likely due to the small number of injected faults.

Despite the lack of data regarding diagnosability of concurrency related faults in the absence of contract instrumentation, it is a well-known fact that such faults are difficult to diagnose. The use of contracts as test oracles is therefore clearly expected to help since our results show that faults are located very close to the detecting contract.

Though we used all the system-level concurrency faults we could use in our industrial system, our fault sample remains small and the above results will need to be confirmed by further studies.

V. Conclusion

We described the challenges involved in defining contract assertions describing both functional and concurrent properties in concurrent systems and presented a solution implemented with an extended version of the Java Modeling Language (JML). Using an industrial concurrent system as case study and actual system-level faults, we systematically analyzed the use of contract assertions as test oracles to detect and diagnose *concurrency related* faults. Results clearly show that assertions were effective at improving system observability and diagnosability since they were able to detect all faults and that such faults were located in the immediate vicinity of the assertion detecting them. Future work will attempt to extend these results to functional faults and replicate these results on other systems.

References

[1] B. Meyer, "Design by Contract," *IEEE Computer,* vol. 25, pp. 40-52, Oct. 1992 1992.

[2] B. H. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems,* vol. 16, pp. 1811-1841, 1994.

[3] P. America, "Inheritance and Subtyping in a Parallel Object-Oriented Language," in *European Conference on Object Oriented Programming*, 1987, pp. 234-242.

[4] P. America, "Designing an Object-Oriented Programming Language with Behavioural Subtyping," in *REX School/Workshop on Foundations of Object-Oriented Languages*, 1990, pp. 60-90.

[5] G. T. Leavens and W. E. Weihl, "Specification and Verification of Object-Oriented Programs Using Supertype Abstraction," *Acta Informatica,* vol. 32, pp. 705-778, 1995.

[6] K. Dhara and G. T. Leavens, "Forcing Behavioural Subtyping Through Specification Inheritance," in *International Conference on Software Engineering*, 1996, pp. 258-267.

[7] K. Arnold*, et al.*, *The Java Programming Language*, 3rd ed. Reading, MA: Addison-Wesley, 2000.

[8] G. T. Leavens*, et al.*, "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes,* vol. 31, pp. 1-38, 2006.

[9] G. T. Leavens*, et al.* (2009, JML Reference Manual. Available: http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html

[10] Y. Cheon and G. T. Leavens, "A Runtime Assertion Checker for the Java Modeling Language (JML) " presented at the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, 2002.

[11] W. Araujo*, et al.*, "Concurrent Contracts for Java in JML," presented at the 19th International Symposium on Software Reliability Engineering, ISSRE, Seattle, WA, United states, 2008.

[12] C. Flanagan and S. N. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs," in *ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 2004, pp. 256-267.

[13] R. Agrawal*, et al.*, "Optimized Run-Time Race Detection And Atomicity Checking Using Partial Discovered Types," presented at the 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, CA, United states, 2005.

[14] S. Park*, et al.*, "Falcon: fault localization in concurrent programs," presented at the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, Cape Town, South Africa, 2010.

[15] E. Rodríguez*, et al.*, "Extending JML for Modular Specification and Verification of Multi-threaded Programs," presented at the 19th European Conference on Object-Oriented Programming, ECOOP 2005, Glasgow, United kingdom, 2005.

[16] W. Araujo*, et al.*, "Enabling the Runtime Assertion Checking of Concurrent Contracts for the Java Modeling Language," presented at the 33rd ACM/IEEE International Conference on Software Engineering (ICSE '11), Honolulu, HI, United states, 2011.

[17] B. Jacobs*, et al.*, "Safe concurrency for aggregate objects with invariants," in *IEEE International Conference on Software Engineering*, 2005, pp. 137-147.

[18] P. Nienaltowski and B. Meyer, "Contracts for concurrency," in *International Symposium on Concurrency, Real-Time and Distribution in Eiffel-like Languages*, 2006.

[19] V. Arslan*, et al.*, "SCOOP - concurrency made easy," in *Dependable Systems: Software, Computing, Networks - Research Results of the DICS Program*. vol. 4028, B. Meyer*, et al.*, Eds., ed: Springer Verlag, Heidelberg, Germany, 2006, pp. 82-102.

[20] A. Greenhouse*, et al.*, "Observations on the assured evolution of concurrent Java programs," *Science of Computer Programming,* vol. 58, pp. 384-411, 2005.

[21] S. Qadeer and D. Wu, "KISS: Keep it simple and sequential," presented at the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementaion (PLD'04), Washington, DC, United states, 2004.

[22] T. Elmas*, et al.*, "VYRD: VerifYing concurrent programs by runtime refinement-violation detection," presented at the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 05, Chicago, IL, United states, 2005.

[23] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," *Commun. ACM,* vol. 53, pp. 93-101, 2010 2010.

[24] P. Ratanaworabhan*, et al.*, "Detecting and tolerating asymmetric races," *SIGPLAN Not.,* vol. 44, pp. 173-184, 2009 2009.

[25] Y. Le Traon*, et al.*, "Design by contract to improve software vigilance," *IEEE Transactions on Software Engineering,* vol. 32, pp. 571-86, Aug. 2006 2006.

[26] B. Baudry*, et al.*, "Building trust into OO components using a genetic analogy," presented at the ISSRE 2000 International Symposium on Software Reliability Engineering, Los Alamitos, CA, USA, 2000.

[27] L. C. Briand*, et al.*, "Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code," *Software - Practice and Experience,* vol. 33, pp. 637-672, June 2003 2003.

[28] W. Araujo, "Assessing the Effectiveness of Design Contracts as Test Oracles in the Detection of Faults in Concurrent Object-Oriented Software," Ph. D. Doctoral Thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, 2010.