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

Wladimir Araujo (corresponding author)
Juniper Networks
700 Silver Seven Rd.
Ottawa, ON, Canada, K2V 1C3
T: +1 (613) 591-2725
F: +1 (613) 591-2799
waraujo@juniper.net

Lionel Claude Briand
University of Luxembourg
SnT centre
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg-Kirchberg
Luxembourg
lionel.briand@uni.lu

Yvan Labiche
Carleton University
1125 Colonel By Drive
Ottawa, ON, Canada, K1S 5B6
labiche@sce.carleton.ca
ABSTRACT

Design by Contract (DbC) is a software development methodology that focuses on clearly defining the interfaces between components to produce better quality object-oriented software. The idea behind 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. Using Java as the target programming language, this paper tackles such challenges by augmenting the Java Modelling Language (JML) and modifying the JML compiler to generate Runtime Assertion Checking (RAC) code to support DbC in concurrent programs.

We applied our solution in a carefully designed case study on a highly concurrent industrial software system from the telecommunications domain to assess the effectiveness of contracts as test oracles in detecting and diagnosing functional faults in concurrent software. Based on these results, clear and objective requirements are defined for contracts to be effective test oracles for concurrent programs whilst balancing the effort to design them. Main results include that contracts of a realistic level of completeness and complexity can detect around 76% of faults and reduce the diagnosis effort for such faults by at least ten times. We, therefore, show that DbC can not only be applied to concurrent software but can also be a valuable tool to improve the economics of software engineering.

Keywords

Design by contract, concurrency, object-oriented programming, Java, Java Modelling Language, verification, runtime assertion checking, fault detection, fault diagnosis.
1 INTRODUCTION

Including specifications of program behaviour together with the source code is not a new idea. Design-by-Contract (_DbC_) (Meyer, 1992) is one of the most elaborate software development methodologies that put such idea into practice, with Eiffel being the best-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 (Müller et al., 2006)\(^1\), 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/after a visible method\(^2\) execution. Behavioural subtyping (America, 1987, 1990; Leavens and Weihl, 1995; Liskov and Wing, 1994; Raghavan and Gary, 2000) is an integral part of DbC. A subtype automatically inherits the specification (contracts and invariants) from its super-types (Dhara and Leavens, 1996; Leavens, 2006; Leavens and Naumann, 2006). 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).

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 occurs

\(^{1}\) Meyer originally named them class invariants but we prefer the term object invariant since it is an invariant about an object.

\(^{2}\) Typically non-private methods are considered visible. However, this varies with the specification language. See section 3.1 for the specific meaning in the context of the Java Modeling Language.
even on correct programs with respect to concurrency control. Basically, interference for a method’s precondition happens when the Run-time Assertion Checking (RAC) 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 at 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. The second challenge is the specification and verification of thread-safety properties (i.e., which objects are safe to be accessed by an executing thread) using contract assertions. It is common practice to have methods in a concurrent class (informally) specify which locks the client is required to hold prior to executing such methods, which locks the method is going to (potentially) acquire during its execution, and, in some cases, the order in which they must be acquired to avoid deadlocks (Weikum and Vossen, 2002). From a specification perspective, these safety properties have been typically associated with preconditions, therefore allowing subtypes to strengthen (e.g., require more locks to be acquired) or weaken (e.g., require fewer locks to be acquired) the precondition. This may cause unexpected behaviour with respect to such safety properties as discussed in (Araujo et al., 2008). From a verification perspective, the instrumented version of a sequential system can be used in place of the original one and the results obtained during verification activities will be valid for the original system. This is not necessarily the case for concurrent systems, in which changes in the execution time of different threads due to the introduction of the RAC code may affect the execution paths a program takes, therefore preventing certain failures from happening whilst forcing others to occur.

Instrumenting contract assertions requires tool support. The Java Programming Language (Arnold et al., 2000) does not provide native support for 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) (Leavens et al., 2006; Leavens et al., 2009) as the specification language used to write contracts. JML allows the specification of properties from simple assertions (lightweight properties) about pointer null-ness to complete functional correctness of program components (strong properties). JML is a behavioural
interface specification language with which one can specify the syntactic and behavioural interface of a portion of Java code. It also includes notations for pre- and postconditions, invariants, and offers mechanisms for specification inheritance, thus providing support for DbC. JML has a Java-like syntax and JML specifications can even perform method calls. It also provides a rich set of model classes (i.e., classes that can only be used in specifications) to construct rich abstract descriptions of program behaviour. The JML toolset comes with a compiler that translates JML specifications into RAC code, producing Java classes instrumented with executable assertions. The JML compiler (Cheon and Leavens, 2002) produces RAC code that enforces DbC and behavioural subtyping.

One of the purposes of developing techniques to extend DbC to concurrent programs is to instrument such programs with oracles from contracts. Creating test oracles is time-consuming, error prone (Beizer, 1990; Staats et al., 2011), and represents a significant portion of the testing effort and probably even more so for concurrent programs. Automatically deriving test oracles from contracts through the generation of executable assertions is an attempt to address these issues. Crafting contracts, however, also requires effort and is subject to errors and this is why it must be supported by a specification language like JML enabling their definition at a higher level of abstraction. It is also important, from a practical standpoint, to determine what should be the level of detail and content of such contracts, i.e., the effort put into writing contracts, to enable such task to be as cost-effective as possible.

The JML compiler (jmlc) originally provided support for generating RAC code for sequential programs. One important contribution of our work is that we extended JML with new specification constructs to address the above-mentioned problems and modified the JML compiler to generate RAC code for these constructs as well as other concurrency related constructs (Rodríguez et al., 2005a) which had never been implemented in a RAC scenario (Araujo et al., 2011a).
In earlier work reported in conference venues, we addressed the objectives of (1) creating contracts specifying concurrency properties as well as functional properties in a concurrent environment (Araujo et al., 2008), implementing them in a compiler to generate instrumented code, and (2) verifying that an instrumented system can be used in place of its production version (Araujo et al., 2011a). In the current paper, based on a large-scale industrial case study involving a highly concurrent software product of the telecommunications industry, we assess whether such contracts can help detect a high number of faults and significantly reduce diagnosis effort. In summary, our goal is to provide a practical solution, supported by evidence, to the problem of applying Design by Contract to concurrent object-oriented software, with an emphasis on the Java programming language and JML.

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 introduces the concept of concurrent contracts and how they address the problems of interference and thread-safety, a summary of our previous work. It also briefly introduces JML, focusing on the constructs required to understand this work. Section 4 summarizes our previous work on implementing a concurrent RAC for JML and assessing the validity of system testing results performed with it. Section 5 presents an extensive industrial case study, which leads to a set of recommendations regarding the contents of concurrent contracts (section 6) so that they can be effective and efficient test oracles with respect to functional faults in the context of concurrent systems. We conclude in section 7.

2 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.
Atomizer (Flanagan and Freund, 2004) is a dynamic checker for Java programs. It checks for atomicity: “A method is atomic if its execution is not affected by and does not interfere with the concurrently executing threads.” Agrawal et al. (Agrawal et al., 2005) combine runtime and static analyses to check for atomicity. Both solutions have a common limitation intrinsic to dynamic checking: they require that the code be exercised to be checked. Atomicity checking relies on annotations to determine the set of locks protecting access to a variable (which is possibly flawed), or relies on lock inference algorithms. Lock inference requires multiple executions of a method (or block of code in general) being checked for atomicity, and therefore does not fit well with a RAC-based approach to verification since the latter depends on a predicate which is expected to yield an answer in every execution. Atomicity, therefore, is to be established prior to executing functional contracts. Unfortunately, atomicity does not prevent interference as we have defined it earlier and thus cannot be combined with RAC. A similar problem happens with pattern-based concurrent bug detection (Park et al., 2010).

Rodríguez et al. (Rodríguez et al., 2005b) present solutions to the problem of specifying lock acquisition and thread-safety properties in JML but fail to consider the issue of inheritance. Although they propose several constructs, none of them were implemented in the JML toolset. We implemented all the constructs we propose on the JML compiler and generate RAC code for them (Araujo et al., 2011a).

Jacobs et al. (Jacobs et al., 2005) 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 (possibly important) cases. Our approach solves this issue with the introduction of safepoints (Araujo et al., 2008).

Nienaltowsky and Meyer (Nienaltowski and Meyer, 2006) present an interesting approach regarding the use of contracts in a concurrent environment. They target SCOOP (Arslan et al., 2006), an extension of the Eiffel language to provide support for concurrency. They do not consider specification inheritance nor conduct any experiment.
Greenhouse et al. (Greenhouse et al., 2005) describe a series of annotations related to the specification of the concurrent behaviour of a Java program. Their annotations are similar to those of (Rodríguez et al., 2005b) with respect to locking properties and member ownership, and thus suffer from the same limitations. They do not present a construct to state the thread-safety of an object. Their work focuses on the static checking of such properties with an emphasis on the evolution and refactoring of concurrent programs. They do not present a solution to the verification of functional properties in combination with concurrency properties.

Qadeer and Wu (Qadeer and Wu, 2004) 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. Their approach may miss some faults but it never reports false positives, exactly as ours. They focus on data races only, whilst our approach covers deadlocks as well. Their approach suffers from one limitation: it does not allow a developer to specify under which conditions objects are expected to be thread-safe, which leads to false positives since environmental assumptions cannot be taken into account by the checker.

VYRD (Elmas et al., 2005) is a tool to detect data races based on a trace refinement technique. A concurrent execution must be a refinement of a trace specification. FastTrack (Flanagan and Freund, 2010) is a precise (i.e., no false positive) dynamic race detection algorithm based on Lamport’s happens-before memory access relation. ToleRace (Ratanaworabhan et al., 2009) is a system to detect and tolerate data races. The detection mechanism is similar to ours (Araujo et al., 2011a), although they also implement techniques to tolerate races or report them otherwise. Differently than our approach, their tool does not require annotations. None of these approaches address the verification of functional properties. They also do not consider inheritance.

In (Le Traon et al., 2006) the authors 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 (Baudry et al., 2000b). Briand et al. (Briand et al., 2003) 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 summarize, except for our previous work (Araujo et al., 2008), (Araujo et al., 2011a) and (Araujo et al., 2011b), no existing work reports on a unified solution to the specification and dynamic verification of concurrent and functional properties. Furthermore, no empirical study rigorously assesses the effectiveness of contracts as test oracles in terms of fault detection and diagnosis based on a highly concurrent industrial system. This is the main contribution of this paper leveraging on our previous work. By doing so, we believe to present a viable solution to the problem of applying DbC to concurrent software and we provide credible empirical evidence of its cost-effectiveness.

3 CONCURRENT CONTRACTS

This section presents the problems and solutions of using contracts to specify behaviour and generating runtime assertion checking code for concurrent programs. It starts with a summary of JML, followed by a description of the problem of interference together with our solution. We only present an overview of our solution here and refer the interested reader to our previous publications for a complete argumentation: extensions to JML (Araujo et al., 2008), extensions to the JML compiler (Araujo et al., 2011a).

3.1 The Java Modeling Language

JML has a Java-like syntax and specifications, written in javadoc style, and can even perform method calls. In JML, the interface of a method is specified through a set of clauses. The most relevant clauses for this study are:

- **requires**: specifies a precondition.
- **ensures**: specifies a postcondition.
- **when**: specifies a wait condition.
• **signals**: specifies a predicate to hold if a given exception is thrown (i.e., the exceptional postcondition).

• **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 and what it ensures.

A method specification is composed of 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). In JML, the preconditions of a method, as well as arguments to the `\old` operator are evaluated in the method’s *pre-state*. The method postconditions 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” ([Leavens et al., 2006](#), p. 8).

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 method and constructor, with the exception of private methods marked with the `helper` modifier. These are methods used to establish intermediate states.

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.
• Operators < and <= are used to test the order of lock acquisition. A lock is greater than another if it was acquired later.

Figure 1 shows an example of a contract. It tells 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).

JML allows the use of model fields in specifications by using the modifier model in a declaration. Model fields are accessible only to specification code and are treated as regular Java fields. They are useful to support abstract specifications. Model fields can be specified on interfaces, something Java does not allow. Being specified on interfaces and classes, they are also inherited. Model fields are realized by concrete classes through the represents clause, which maps (in)directly model fields to a set of Java fields. Similar to model fields, ghost fields are also specification only, can be declared on interfaces and classes. However, ghost fields are explicitly assigned values within a method’s body instead of using the represents clause.

A method is pure if it is declared with the modifier pure or is a member of a pure class or interface. Pure methods must terminate and are not allowed to have side-effects (i.e., they cannot assign to fields that exist in the pre-state). Pure constructors must terminate and can only assign to the object being constructed. Only pure methods are allowed in assertions (e.g., in the requires and ensures clauses of a method specification).

3.2 Contracts and Concurrency

In a previous paper (Araujo et al., 2008), we addressed the problem of interference by combining the use of safepoints (a language construct we introduced to demarcate the safe places to evaluate contracts) with thread-safety requirements (language constructs we introduced to describe predicates specifying conditions for accessing shared objects), as defined and illustrated below (Figure 1). 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 LinkedQueue instance to fully execute method insert(), which does not acquire such a 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;
    /*@
    @  requires head == last;
    @  assignable \nothing;
    @  ensures \result == null;
    @  also public normal_behavior
    @  requires head != last;
    @  assignable head, head.next.value;
    @  ensures head == \old(head.next) &&
    @  \result == \old(head.next.value);
    */
    public synchronized Object extract() {
        synchronized (head) {
            //@requires_safepoint:
            Object x = null;
            LinkedNode first = head.next;
            if (first != null) {
                x = first.value;
                first.value = null;
                head = first;
            }
            //@ensures_safepoint:
            return x;
        }
    }
}
```

**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 is not due to erroneous concurrency control on the part of the implementation either of the client or the provider. Interference can also happen between the contract evaluation point (pre- and post-state) and the method entry and exit points. Since interleaving occurs outside the method body,
this is called \textit{external interference}. The previous case, where interleaving occurs inside
the method body is called \textit{internal interference}.

A \textit{safepoint} is any point inside the method body where it is safe to evaluate contract
assertions. A \textit{precondition safepoint} is a point where it is safe to evaluate preconditions
and invariants, and the pre-state predicates of postconditions. A \textit{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. If no precondition (resp.
postcondition) safepoint is explicitly specified for an execution path, it defaults to the
method pre-state (resp. post-state). This ensures the semantics of evaluating preconditions
once at the beginning of the method and postconditions once at the end of the method is
maintained for concurrent programs. The \texttt{requires\_safepoint} and \texttt{ensures\_safepoint} labels demarcate those safepoints. At the precondition safepoint
in Figure 1 (line 13), all the objects referenced by both requires clauses (lines 2 and 6)
and the contents of the \texttt{old} statements in the \texttt{ensures} clauses (lines 8-9) are properly
protected by locks. At the postcondition safepoint (line 21), the field \texttt{head}, present in the
\texttt{ensures} clause at lines 8-9, is properly protected by a lock. Since \texttt{result} refers to
local variable \texttt{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 \texttt{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. In case
the method does not return a value, the \texttt{ensures\_safepoint} label can be placed at the
end of a block or just before the method returns.
Figure 2: Method declaration exemplifying the use of thread-safety specification clauses.

We also solved (Araujo et al., 2008) 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 list 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 the current thread must or must not hold before or after a method execution. Figure 2 shows such an example (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 to \( (\text{connected} \lor \neg \text{connected}) \land r.isRequest() \)). 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 

\texttt{ensures\_thread\_safe} clause specifies that the object returned by the method must be thread-safe on the method’s post-state. The \texttt{*\_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 $\texttt{result}$ on the post-state. Precondition safepoints prevent interference related to field \texttt{connected}. As these are the only possible sources of interference, we conclude that combining safepoints and thread-safety predicates guarantees \texttt{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. A complete argument is presented in (Araujo, 2010; Araujo et al., 2008).

\section{RUNTIME ASSERTION CHECKING, INTERFERENCE FREEDOM AND TEST VALIDITY}

As seen in section 3.2, freedom from interference is required to ensure that the functional facets of contracts are evaluated correctly. Establishing freedom from interference, therefore, depends on the correct placement of safepoints within a method body (the determinant of internal interference), and on the correct and complete specification of the concurrent facet of each contract (the determinant of external interference). The correct placement of safepoints is dealt with in section 5, as part of the case study execution. This section deals only with the external interference component.

Another significant issue is the practical use of RAC with industrial systems. The correct and complete specification of the concurrent facet of contracts is not sufficient to ensure that a system instrumented with RAC code can be used as a valid replacement of the original system during system testing activities. Equivalence between the instrumented system and its original version is required. This equivalence must enable the conclusion that the absence of failures in the instrumented system when subject to a test suite implies the absence of failures in the original system when subject to the same test suite. This
equivalence must also enable the conclusion that the occurrence of a failure in the original system implies the occurrence of a failure in the instrumented system. However, the manifestation of such failure is expected to be different in the two systems: the instrumented system is expected to raise an assertion error due to a contract violation.

We investigated the equivalence between the instrumented and the original version of a system (Araujo et al., 2011a). The equivalence is determined based on the observable behaviour of the system (a factor named indistinguishability) and on the use of resources (a factor named runtime overhead). These two factors were considered independently. The system used for the execution of such experiment is the one used for this paper’s case study and is described in section 5.2. The runtime overhead was analyzed with respect to the consumption of three resources: CPU, memory and persistent storage. Though the impact of instrumentation is, as expected, significant, the load on the system can be adjusted to guarantee resource usage conditions by the instrumented version similar to those of the production version. As a result, instrumented applications can be executed in the same environment as the one used by the original version. Indistinguishability was analyzed based on the behaviour of the instrumented and the original versions of the system in the presence and absence of faults. Furthermore, a systematic analysis of the instrumentation process concluded that no thread interleavings present in the original version of the system are prevented from occurring in the instrumented version. The two versions were thus deemed indistinguishable. Combined with the runtime overhead result above, the instrumented system is deemed equivalent to the original one for system testing purposes.

The equivalence result guarantees that concurrent faults manifest themselves equally in both versions of the system. It is then possible to study the completeness and correctness of the concurrent facet of contracts. Contracts are defined by humans and, therefore, can be incorrect and incomplete. An incorrect contract, for the purposes of this discussion, is one that specifies more restrictive conditions for the establishment of thread-safety than necessary for the establishment of freedom from interference. In this case, the contracts will raise assertion violations for correct executions (i.e., false positives). This is not a serious problem, although counter-productive, since developers can identify those cases
and adjust the contracts accordingly. An incomplete contract, however, has the potential of allowing the execution of functional contracts in a non-thread-safe environment, therefore threatening the validity of any results obtained during testing with the instrumented system. Incompleteness can be caused either by an oversight from the contract developer or by an intrinsic limitation of the RAC and the specification language it uses.

In (Araujo et al., 2011b), we determined experimentally (through fault injection) that the contracts can detect all concurrent faults (race conditions and deadlocks), and thus any completeness issue would be simply a case of design oversight. We also determined that the concurrent facet of contracts could be considered largely complete to begin with and could then be updated in later iterations to detect the remaining concurrent faults. We can therefore state that the execution of the functional facet of contracts in a concurrent system, which also contains contracts with a concurrent facet, can be largely guaranteed to be free from external interference.

Combined with the equivalence result regarding the instrumented version of the system, freedom from external interference guarantees the interpretability of the results we provide in section 6 regarding our case study described in the next section.

5 CASE STUDY

Up to this point, providing an overview of results reported elsewhere, we have shown that DbC can be applied to concurrent programs by defining specialized specification constructs (section 3.2) and by showing that the results of executing tests on an instrumented program can be considered equivalent to those on the original version of such program and free from external interference (section 4). The missing step and main contribution of this paper is to investigate whether contracts, when applied to concurrent software using our JML extensions, can effectively be used to detect and diagnose functional faults.

As shown in (Briand et al., 2003), the content of the contracts has a significant influence on their ability to detect and diagnose faults in sequential programs. Following a similar
approach, in the context of concurrent software, we answer the following research questions which are part of the overall objective stated above: (Q1) Are there limitations regarding the type of faults contracts can detect in a program? (Q2) What is the relation (if one exists) between the effort spent in designing contracts (i.e., their level of detail) and their ability to detect and diagnose faults? (Q3) Is the complexity of the methods being specified a factor in such relation? (Q4) Are there faults that cannot be detected by contracts irrespective of their level of detail? (Q5) Based on such relations and the categories of undetectable faults, what recommendations can be offered to practitioners to balance the effort in contract design with the yield in terms of fault detection and diagnosis efficiency?

We go about answering these questions by defining precise measures for observability and diagnosability, the quantities that reflect how well faults can be detected and diagnosed, respectively, as in (Briand et al., 2003). We then categorize contracts based on the features of the specification language they use (e.g., the use of quantifiers). These categories determine the effort to design a contract. We also define a fine grained measure of contract complexity independent of the above categories, which can be used as a surrogate measure to assess the effort in designing a contract. Using these measures we execute an experiment consisting of injecting real faults in an industrial system and measuring the observability and diagnosability of such faults. This is achieved by instrumenting the system with contracts of different types while recording relevant data on contract and method complexity. Differently from the work of Briand et al. (Briand et al., 2003), we inject faults that were once present in the system instead of using program mutation. By applying objective and focused fault selection criteria, we enable the investigation of the above research questions with a focus on system testing and in a highly realistic setting. Another obvious difference is the focus on concurrent programs.

This section begins by reviewing the concepts of observability and diagnosability (section 5.1). It follows with a description of the test bed used to conduct this study (section 5.2). The experimental methodology is described in sections 5.3 (measure definitions) and 5.4 (experimental procedure). Section 6 reports on the results of our experiments.
5.1 Background on Observability and Diagnosability

This section is an introduction to the concepts and measures of observability and diagnosability, which will be used in our industrial case study. For a complete exposition, see (Briand et al., 2003; Le Traon et al., 2006).

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 effort required to diagnose 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. Our proposed distance definition is adapted from (Briand et al., 2003) to make it more precise by resolving ambiguities (flagged below) and is a contribution of the current paper. This, like any model, is a simplification of reality since expert developers frequently use shortcuts to diagnose a failure. Such simplification, however, is necessary to perform a rigorous, large-scale study of diagnosability.

The starting point of the diagnosis is the method in which the failure was detected: either the caller of the method with the violated method precondition or the method with an internal assertion or a post-condition violation. We do not use internal assertions since contracts are completely defined in terms of pre-conditions, post-conditions and invariants, both in the functional and the concurrent facets. The search proceeds then from the beginning of the method in which the failure 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. We assume a method is investigated only once. The method to be investigated is determined according to the dynamic type of the target object, not its static type (ambiguity 1). A method call is not
explored if it is certain, based on method arguments and syntactical constraints, that the particular execution path leading to the fault did not execute it (ambiguity 2). For instance, a method called in an else block of an if-then-else statement is not investigated if we ascertain that the then block was executed. The behaviour described by ambiguity 1 above is implied but it needed to be explicitly described. The behaviour covered by ambiguity 2 is a fundamental addition to the diagnosis flow to reflect the behaviour of a developer exploring multiple execution branches: when sure, a developer only investigates one branch, but when in doubt, alternate branches are investigated. This additional effort must be accounted for.

To illustrate this process, consider an execution depicted by the UML sequence diagram of Figure 3. Assuming an assertion violation occurs when evaluating the precondition of method $e()$, 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.

![UML Sequence Diagram](image)

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

An alternate measure of diagnosability is the number of statements located between the contract immediately preceding the faulty statement and the contract detecting the failure for a particular execution: the *diagnosis scope* (Le Traon et al., 2006). This comprises all statements suspected of containing the fault from the failure observation point (i.e., the
assertion violation) to the satisfied contract immediately preceding the faulty statement according to the specific execution that uncovered the failure. A good estimate of the diagnosis scope is the number of method lines of code (MLOC) for all methods included in the diagnosis scope. MLOC counts the number of non-blank and non-comment lines of code inside method bodies.

Both measures may not reflect accurately the effort a developer spends in diagnosing a failure. They do not precisely take into account the complexity of the methods being investigated (e.g., complexity in terms of control flow). McCabe’s Cyclomatic Complexity (VG) ([McCabe, 1976]) is a well known measure of method complexity. It basically indicates the number of execution paths a developer needs to mentally traverse to completely understand a method. Other metrics combine control flow complexity and computational complexity ([Weyuker, 1988]). With the exception of computationally intensive software (e.g., numeric processing), which is not the target of this study, VG is a simple measure expected to correlate with the effort in determining if a fault is located inside a particular method body or not.

We define the weighed distance measure as the number of execution paths to investigate, from the detection point (where the failure occurred) to the location of the faulty statement according to a diagnosis flow. The diagnosis flow is identical to the one used for the distance measure of ([Briand et al., 2003]). The weighed distance is then the sum of the VG values of the methods of a diagnosis flow. For instance, assuming methods a(), b(), c(), d(), e() and f() in Figure 3 have VG values of 2, 5, 3, 4, 2 and 5, respectively, the weighed distance for diagnosis flow [a, b, c] is 10 and 17 for [e, f, a, b, c].

All above measures are expected to correlate (see section 6.1.3) and differ only in terms of granularity. These are surrogate measures of the diagnosis effort, the effort to diagnose a fault, which is affected by the programmer’s skills. More investigation is required to determine which one is a better fit for determining diagnosis effort in terms of man-hours for example. To enable their comparison, we adopt the definition of global diagnosability ([Le Traon et al., 2006]): \( \Delta = 1 - \frac{\delta}{\delta} \), where \( \Delta \) is the global diagnosibility, \( \delta \) is the absolute
measure of diagnosis effort (distance, weighed distance, diagnosis scope), and $\theta$ is the maximum value of the diagnosis effort for an execution thread $T$. The value of $\theta$ can then be measured using the total number of statements (or methods, or sum of methods’ VGs) traversed on an execution thread as measured by diagnosis scope (or distance or weighed distance). The global diagnosability then varies between 0, for the case one must traverse the whole execution thread to diagnose a fault (i.e., no contract detects a fault), and 1, for the (asymptotic) case the faulty statement coincides with the one generating the assertion error.

5.2 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 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 a subsystem, called the router driver, which interfaces with Juniper’s E-series routers. It 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 us to explore all proposed constructs in section 3.2. With respect to code size, the router driver subsystem is composed of 54
classes and interfaces (33,509 LOC), all of which are used in a concurrent environment. Of these, 34 present concurrent behaviour. In many ways this can be considered a representative concurrent system in the telecom domain.

We use an actual test suite composed of a 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 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 was 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 test suite are confidential information of Juniper Networks. However, the key property for this study can be stated: there is at least one test case in the test suite that exercises each fault of the fault database in such a way that the fault manifests itself as a failure in the production system or as an assertion violation in the instrumented system if the fault is observable by 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 concerning the load factor is its ratio between program executions, 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.

5.3 Categories and Complexity of Contracts

To assess the effect of contract content on the observability and diagnosability of the system with respect to functional faults, the content of the contracts is restricted according to different rules that define a contract type. To clearly relate the effort in
designing contracts and their ability to detect and diagnose faults, a measure of contract complexity is defined as a surrogate measure of effort.

### 5.3.1 Contract Specification Types

Briand et al. (Briand et al., 2002; Briand et al., 2003) define three levels of precision of a contract for their experiment: at the highest precision, every distinct condition possibly resulting from a different set of inputs or system state is distinguished in the post condition; the intermediate precision only distinguishes conditions for the standard situation (expected execution) from exceptional situations that are also addressed by the method; the lowest precision just defines the ranges/enumerations of values expected as resulting from executing the method. These levels suit their work well since they consider only analysis contracts. The contracts to be developed for the router driver are at the design level, and normally include some information regarding the exceptional behaviour of a method, especially in Java since exceptions normally have to be declared in the method signature. Therefore, the intermediate precision level is not useful.

The four contract types we defined and used in our study are described below in ascending order of sophistication (see Table 1 for a summary) and exemplified next. We decided for a categorization based on the structure of the contract (the language features used) instead of precision of specification (behaviour specified) since the latter cannot be automatically checked. It is expected that contracts of higher sophistication require more effort to design since they enable the more detailed specification of method behaviour. The types are:

- **Basic**: defines ranges and enumerations for native types in pre/postconditions and invariants, checks for null-ness of references and simple properties of such objects. A simple property only involves pure (side effect free) methods (e.g., `isEmpty()` of the Java collection classes). No model fields are allowed in any specification. Expressions using quantifiers are also not used.

- **Elementary**: basic plus the use of model fields, model methods, and ghost fields. Interfaces also make use of model instance fields to specify some behaviour.
Model methods serve as convenience methods in this case to factor out computations that could otherwise be stated directly in the contract. They improve the contract’s readability.

- Intermediate: elementary plus quantification over elements of collections given as arguments, data members, and return values, plus the use of pre-defined modeling types. The predicate of such quantified expressions should only refer to basic properties of such elements specified through the use of their pure methods or ranges of values if native types.

- Advanced: no language restrictions. The use of specification-only pure classes (model programs (Leavens et al., 2009) are not supported by the RAC (Cheon and Leavens, 2003)) designed to simulate the correct behaviour of a class or cluster of classes is allowed as well as non-trivial model methods. For our purpose, a model method is considered non-trivial if the contract specifying its behaviour is required to be of type advanced to allow for complete behaviour specification (i.e. it is not just a convenience method).

Table 1: JML features allowed in contracts as a function of their type.

<table>
<thead>
<tr>
<th></th>
<th>Ranges, enumerations, nullness and pure methods</th>
<th>Model and ghost fields, model methods</th>
<th>Quantifiers and modelling types</th>
<th>Pure classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elementary</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Intermediate</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Advanced</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

The use of thread-safety related constructs is not subject to the above rules since they are necessary for the safe evaluation of the functional contracts regardless of their type.

The Basic type’s purpose is to enable creating contracts that state properties using only the existing code; the designer does not put effort in introducing new code (e.g., model fields and methods) or creating constructs to loop over objects (e.g., quantifiers). Therefore, Basic contracts are generally simple to design. Elementary contracts introduce the use of model fields. Although declaring a model field is quite simple, a subclass will require a represents clause to map the abstract model field to a concrete
implementation, which can be quite a complex task depending on the nature of the class. This is the reason why the use of model fields is tracked as part of a separate, more sophisticated type of contract than Basic. Intermediate contracts introduce the use of quantifiers. The rationale for having the use of quantifiers (both existential and universal) tracked in a separate contract type is the added effort required when specifying the expression itself, the predicate to be checked and enforcing the thread safety of all objects in the quantifier’s domain. Furthermore, it is quite common to use pre-defined modelling types to conform to the purity requirements of specifications (section 3.1). The use of such classes is also an added effort to contract design. Advanced contracts introduce the use of pure classes and non-trivial model methods, which have to be created solely for the purpose of type and method specification. These are used to simulate the visible behaviour of methods and, therefore, need to be pure (i.e. side-effect free) to be used within contracts. The added effort put in the design of such classes and methods is significant, which justifies tracking their use in a separate contract type.

We use method open() of interface Central (Figure 4) to illustrate the differences between the types of contracts (shown in Figure 5 and Figure 6). The basic contract for method open() refers only to parameters and exceptions. No postcondition is specified because the return value is an interface that uses model fields (not shown) and model fields are not allowed in basic contracts. Although exception ConnectionAllocatedException is not part of the method signature, it still appears in the contract since this exception is a child of RuntimeException and RuntimeException is not mandatory in method signatures. The elementary contract incorporates the use of model field peers. Also notice that the elementary contract is more detailed than the basic one since it can refer to model fields in the return value as well as instance fields on the interface. By adding an invariant that uses the universal quantifier to specify a property for each element of the peers collection, the contract becomes of the intermediate type. The advanced type of contract uses the Connection class (not shown) in its specification (the ghost field connections is a collection of all established connections). It is a specification-only pure class designed just to represent the connection between two peers. It is not actually used in the method’s implementation.
Notice that the complete behaviour of the method can be specified, i.e. the contract simulates all possible conditions both normal and exceptional and describes the post-state as a function of the pre-state to the maximum level of detail that a client of the method could expect. In this example, this means the precise conditions in which a connection is created and recorded as well as the conditions in which it fails to create a connection and the particular exceptions thrown for each condition.

```
/**
 * The central establishes a connection between peers.
 * Peers are identified by unique names.
 * There are two ways to establish a connection: callback or rendezvous.
 * In the callback mode, a peer registers to receive a notification when
 * a connection is attempted. In the rendezvous mode both peers attempt
 * to open a channel and wait until both participants are present.
 */
public interface Central {
    /**
     * Opens a channel to the remote peer. The remote peer must be
     * registered to receive connection notifications.
     * @param aRemoteId the ID of the remote peer
     * @param aLocalId the ID of the local peer
     * @return the channel to be used to communicate with the remote peer
     * @throws UnknownPeerException if the central does not have a remote
     * peer registered with the given ID
     * @throws ConnectionRefusedException if the remote peer refused to
     * establish a connection
     * @throws ConnectionAllocatedException if a connection already
     * exists between the two peers
     */
    public Channel open(String aRemoteId, String aLocalId)
        throws UnknownPeerException, ConnectionRefusedException;
}
```

**Figure 4: Interface Central to demonstrate the different types of contracts.**

```
// Contract of Central.open()
behaviour
    requires !aRemoteId.equals(aLocalId);
    signals_only ConnectionRefusedException, ConnectionAllocatedException,
                 UnknownPeerException;
```
Figure 5: Basic, Elementary and Intermediate contracts for \texttt{Central.open()}.  

It is clear (by construction) that contracts of a higher sophistication type can specify all behaviours specified in one of a lower type. This implies that a fault detected by a contract of a lower sophistication is also detected by a contract of higher sophistication.
for the same method and type. It is also clear by looking at the preceding example that a more sophisticated contract requires more effort to write due to the extra level of detail needed to use the more sophisticated features of the language as well as the ability to distinguish more cases through the use of such features.

Having four different contract types implies that it is necessary to have four different versions of the system, each with contracts of one particular type, when running our experiments. If it is not possible for a method to have a contract of a particular type (e.g., because its behaviour can be completely specified by a contract of a lower sophistication type), a contract of a lower sophistication type is used instead.

5.3.2 A Measure of Contract Complexity

Although useful, the contract type alone is not necessarily a complete indication of the complexity of a contract and, therefore, the effort involved in designing it. For instance, a basic contract can contain a large number of terms referring to multiple fields and method parameters while an intermediate contract can contain a single quantified expression. It is clear that, in such a case, the effort necessary to design the basic contract is greater than the effort to design the intermediate contract. A measure of contract complexity is, thus, needed to complement the notion of contract type.

This measure depends on the structure of a contract which must be clearly defined. For the purposes of this measure, a contract is a set of clauses (e.g. a requires statement) and specification cases. A specification case relates clauses within a contract. A type is a set of contracts and invariants. Clauses are Boolean expressions that can contain the usual elements as well as quantifiers. A contract belongs to a type: class or interface. This is necessary to represent the inheritance of specifications in a type hierarchy.

We define a measure of contract complexity (\( Complexity(M) \)) according to a property-based approach for measurement definition (Briand et al., 1996). The notation is as follows: uppercase \( M \) denotes a contract, \( T \) denotes a class or an interface (i.e., a type), \( c \) denotes a clause and \( sc \) denotes a specification case (section 3.1); \(|c|_v\) denotes the number of constructs “\( X \)” of its argument (e.g., \(|c|_v\) denotes the number of disjuncts of clause \( c \)).
\(M^T\) denotes a contract for type \(T\); \(\leq\) denotes the subtype relation between two classes and/or interfaces: the left operand is a subtype of the right operand. Subscripts are used to differentiate methods, types, clauses or specification cases. We assert, based on intuitive arguments presented next, that a contract complexity measure needs to present the following properties:

1. Since a contract is composed of clauses (e.g., preconditions, postconditions, invariants), and each clause can be arbitrarily complex, we have \(c \in M \Rightarrow \text{Complexity}(M) \geq \text{Complexity}(c)\) and \(M_1 \supseteq M_2 \Rightarrow \text{Complexity}(M_1) \geq \text{Complexity}(M_2)\).

2. Logically equivalent clauses in different specification cases are considered to be designed independently. They only relate to each other via a specification case (section 3.1). Grouping clauses into specification cases increases the design effort as one needs to correlate two or more predicates: a precondition and the predicates to be satisfied as a consequence of the precondition, i.e., the postcondition. In other words, all things being equal, a contract with more specification cases is more complex.

\[|M_1|_{sc} \geq |M_2|_{sc} \land \forall c (c \in M_1 \iff c \in M_2) \Rightarrow \text{Complexity}(M_1) \geq \text{Complexity}(M_2).\]

3. Invariants, although defined only once per class or object, are an intrinsic part of every contract and must be accounted as such. Thus, \(\forall c \forall M^T (c \in T \land c \notin M^T \Rightarrow |M^T|_c = |T|_c)\).

4. Inherited specifications must be accounted as being part of the contract of the overriding method in the subclass, i.e., the contracts of the super-type are included in the contracts of the subtype. Therefore \(T_1 \leq T_2 \Rightarrow M_i^{T_1} \supseteq M_i^{T_2}\).

5. The complexity of a clause should reflect the effort the developer spent in producing it. One way of accounting for this effort is to count the number of alternatives the developer needs to consider. The complexity of each disjunct
(i.e., alternative) also needs to be a factor and a way to estimate it is to count the number of conjuncts in them. Therefore, \(|c_1|_{\lor \land} \geq |c_2|_{\lor \land} \land |c_1|_{\lor \exists} = |c_2|_{\lor \exists} \Rightarrow \text{Complexity}(c_1) \geq \text{Complexity}(c_2)\). The number of quantifiers is assumed to be identical in both clauses as they add their own complexity.

Following the approaches described in (Briand et al., 1996), the contract complexity measure is based on graph theory. In our case, this measure maps a contract to the control flow graph required to implement it in an imperative programming language. The idea is to transform a contract into an equivalent program and measure the complexity of that program (in fact, its control flow graph) using a well-established metric. We choose Java for obvious reasons but it could be easily defined for any other imperative programming language. We use the cyclomatic complexity of the resulting control flow graph as a measure of contract complexity.

A detailed translation based on the JML grammar could be easily (but laboriously) described. Since this is not the focus of this paper, we present instead a simplified definition of a contract (analogous to method specifications and a subset of type specifications restricted to invariants as defined in (Leavens et al., 2009)) and the resulting program in Table 2. The precondition, postcondition, wait_condition, invariant and predicate terminals in the simplified grammar are Boolean expressions. The non-terminal quantifier is also a Boolean expression. The mapping for expressions is trivial (except for a quantifier, which is described on the table). The implication operator is translated into its semantically equivalent using the primitive operators available in typical programming languages (\&\&, ||, and ! for Java). The equivalence operator is translated to the equality test operator (==).

**Table 2: Mapping from a JML contract (top, in EBNF format) to a Java program (bottom).** The contract definition is based on the structure above.

```java
boolean contract() {
    if(!(invariant1 && invariant2 && ... ))
```

---
return false;
...
if(preconditioni) {
  if(!waitconditioni)
    return false;
  if(!postconditioni)
    return false;
}
...
return true;
}

boolean forall() {
  var decl;
  while(predicate1) {
    if(!predicate2)
      return false;
  }
  return true;
}

boolean exists() {
  var decl;
  while(predicate1) {
    if(predicate2)
      return true;
  }
  return false;
}

The contract complexity measure (CCM) of a contract is defined as the cyclomatic complexity (VG) of the Java program resulting from the mapping of the contract according to the rules in Table 2 minus the constant 1. The term -1 is necessary for an empty contract to have CCM equal to zero. Model methods with bodies and methods in specification-only pure classes can be referenced from clauses. Different methods perform different computations and should be accounted for according to their complexity. Therefore, \( CCM(m) = VG(m) \), where \( m \) is a model method or a method in a specification-only pure class. This translates to a very simple rule for measuring the complexity of a contract \( M \):

\[
CCM(M) = |\{c \in M\}| + \sum_{c \in M}(|c|_\vee + |c|_\wedge + 2|c|_\forall,\exists) + \sum_{(m: (\forall c \in M)(m \in c))} VG(m),
\]

where \( c \) iterates over clauses, \( m \) iterates over model methods referenced in clauses\(^4\). This rule can be used to compute the CCM of a contract without having to explicitly translate it to a Java program (note that \( m \) is already a Java program). Specification inheritance is

trivially incorporated considering that inherited specifications behave as if they were
defined in the overriding method (and type) (Leavens, 2006), which is reflected in the
CCM properties 3 and 4 above. For instance, the CCM of the advanced contract of
method open() of class Central can be computed by following the calculation
described in Table 3. It yields a CCM of 27.

<table>
<thead>
<tr>
<th>Term</th>
<th>CCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invariant 1:</td>
<td>3 (2 from the quantifier + 1 from the &amp;&amp;)</td>
</tr>
<tr>
<td>(forall Connection c; connections.contains(c); !peers.contains(c.peer1) &amp;&amp; !peers.contains(c.peer2))</td>
<td></td>
</tr>
<tr>
<td>Precondition 1:</td>
<td>2 (from the &amp;&amp;)</td>
</tr>
<tr>
<td>!aRemoteId.equals(aLocalId) &amp;&amp; peers.contains(aRemoteId) &amp;&amp; !connections.contains(new Connection(aRemoteId, aLocalId))</td>
<td></td>
</tr>
<tr>
<td>Postcondition 1:</td>
<td>3 (from the &amp;&amp;)</td>
</tr>
<tr>
<td>\result.connection.equals(new Connection(aRemoteId, aLocalId)) &amp;&amp; !peers.contains(aRemoteId) &amp;&amp; \result.connected &amp;&amp; connections.contains(new Connection(aRemoteId, aLocalId))</td>
<td></td>
</tr>
<tr>
<td>Exceptional postcondition 1:</td>
<td>0^5</td>
</tr>
<tr>
<td>ConnectionRefusedException</td>
<td></td>
</tr>
<tr>
<td>Precondition 2:</td>
<td>1</td>
</tr>
<tr>
<td>!aRemoteId.equals(aLocalId) &amp;&amp; connections.contains(new Connection(aRemoteId, aLocalId))</td>
<td></td>
</tr>
<tr>
<td>Exceptional postcondition 2:</td>
<td>0</td>
</tr>
<tr>
<td>ConnectionAllocatedException</td>
<td></td>
</tr>
<tr>
<td>Precondition 3:</td>
<td>1</td>
</tr>
<tr>
<td>!aRemoteId.equals(aLocalId) &amp;&amp; !peers.contains(aRemoteId)</td>
<td></td>
</tr>
<tr>
<td>Exceptional postcondition 3:</td>
<td>0</td>
</tr>
<tr>
<td>UnknownPeerException</td>
<td></td>
</tr>
<tr>
<td>Model methods and pure classes:</td>
<td>8 (6 from equals() + 1 from hashCode() + 1 from Connection())</td>
</tr>
<tr>
<td>Connection methods (constructor, hashCode() and equals()) referenced in the contract</td>
<td></td>
</tr>
</tbody>
</table>

5 This clause is equivalent to signals (Exception e) e instanceof ConnectionRefusedException and does not contain a Boolean operator. Its presence is accounted for as one clause (top row).

5.4 Experimental Design

The experiment is designed to enable the measurement of observability and
diagnosability of a program instrumented with contracts and determine how such
measures are influenced by the complexity and level of sophistication of contracts.
Moreover, the design takes into account the fact that such measurements are taken by
injecting faults into a concurrent system, which is inherently non-deterministic and thus presents challenges in reproducing such faults. We start by describing how contracts are designed, followed by the procedure to collect the data (fault selection, injection and measurement procedures) and conclude with a definition of success.

Contracts, according to the four types described in section 5.3.1, are specified for all methods of the classes and interfaces of the target system, to the maximum extent possible (i.e., describing the method’s behaviour with as much detail as possible based on the judgment of experimenter, an expert on the system). Having all contracts upfront, allows one to study their completeness from the start. Contracts are specified 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 negatively impacting the performance of the system.

In total, 1536 methods were specified. There is, on average, one contract per 17.6 LOC.

5.4.1 Experimental Procedure

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 experiment, including fault selection, to avoid biased results.

The observability and diagnosability of the instrumented version of the system is measured using faults re-introduced from earlier versions, 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 reproducible in the production system (i.e., detected by the standard test suite)
2. It was originally discovered during system testing
3. 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.

4. It was originally discovered in a version to which no significant new functionality was added to the router driver subsystem as compared to the version used for conducting this experiment.

Points 1 is self-explanatory. Point 2 is necessary to exclude faults reported by developers during development as our focus is system testing. Such faults are discovered during coding or unit testing. Point 3 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 feasible level. Faults located in directly connected clients of the router 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 since it is through those objects that clients interact with the router. Point 4 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. Following these criteria, we selected a total of 139 functional faults.

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 realistic.

There is still the risk that such faults do not represent the complete spectrum of possible types of functional faults. However, given the complexity of the system under test, 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 4 of the selection criteria above).

As described in the introduction of section 5, we want to analyze the limitations of the observability of a concurrent system instrumented with contracts and the characteristics of undetectable faults. We also want to determine the relation (if one exists) between the effort spent in designing contracts (measured in terms of contract type and complexity) and the global observability and diagnosability of a system instrumented with such contracts. With this information in hand, we want to offer practitioners recommendations to balance the effort in contract design with the yield in terms of global observability and diagnosability.

The experimental procedure below describes how to best collect the data specified above. It must be noted that it is more interesting to be able to detect a fault with a contract of lower sophistication type since these are generally easier to specify and reduce the analysis effort. The procedure reflects this preference. It is also of fundamental importance that a fault manifests itself in the same way in both the original and the instrumented version of the system, as otherwise it is not possible to guarantee that the absence of faults in the instrumented version implies the absence of faults in the original version. The procedure also incorporates the necessary steps to identify any discrepancies in fault manifestation between the different versions of the system.

The experimental procedure is as follows:

1. Select a fault satisfying the criteria above and inject it in the instrumented (starting with the lowest level of sophistication) and the production versions of the system.

2. Run both versions of the system through the test suite; the instrumented version should execute in a non-fatal assertion checking mode (i.e., assertion violations are simply logged instead of causing an exception to be raised). 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 (starting with the one instrumented with Basic contracts) through the test suite in regular mode (i.e., with assertion violations reported via thrown exceptions)
   a. If an assertion violation occurs, register the occurrence, record the type of contract that was able to detect the fault, and calculate the distance, weighed distance and diagnosis scope between the violated contract and the fault and go to step 4.
   b. If an assertion violation does not occur, update 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, inject the fault in the instrumented version of the system with a higher sophistication contract type and go to step 3. If there is no version of the system with a higher contract sophistication type, 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.

Regarding the iteration in step 3.b above, modifying a contract (or a set of contracts) to detect a specific fault cannot introduce incorrect specifications since the correct system in combination with the standard test suite can be used to determine the validity of the contract (the system is assumed correct prior to fault injection; see section 4). Such iteration will enable the determination of limitations in contract fault detection effectiveness (research question Q1) though in practice we can expect the effectiveness to be lower, to an extent depending on the developers’ skills. Furthermore, it also allows the determination of how incomplete the initial contracts are, thus contributing to answering research question Q5.

5.4.2 Definitions of Success

Success is assessed by the assertions’ detection rate of the injected faults and by the ease with which faults are diagnosed. Observability is measured by the fault detection rate and diagnosability by estimating the diagnosis effort in terms of the distance, weighed distance and diagnosis scope between the fault and the contract that detected it and
calculating the global diagnosability. The higher the observability and the diagnosability the more successful contracts are as test oracles.

6 RESULTS

The results and conclusions reported in this paper are based on a single though comprehensive industrial case study that is representative of concurrent systems in the telecommunications industry. This study should be replicated to allow for a wider generalization of results; our study methodology can be reused and possibly adapted for that purpose.

We first report on observability and second on diagnosability. We then summarize the results and provide guidelines for balancing contract design effort with attainable observability and diagnosability.

6.1.1 Observability: Descriptive statistics

This section addresses research questions Q1 and Q2. 107 (77.3%) of the 139 selected functional faults were detected by the instrumented version of the system through contracts of different types (Figure 7). In other words, 22.7% of the faults are undetectable. The figure shows the contribution of original and updated contracts depending on their type. For instance, Basic, unchanged contracts detected 31.8% of the faults, moving to Elementary contracts did not increase detection, increasing the sophistication of contracts to the Intermediate level added to the detection (4.5% more faults) while the Advanced level did not improve detection over the Intermediate level. Updating Basic contracts increased detection by 10.6%, leading to a total detection of Basic contracts of 42.4%.
These results allow us to make comparisons with previous studies. The overall detection rate of 77.3% is very close to the one reported in (Briand et al., 2003) (80%) and on the same ballpark of the one reported in (Baudry et al., 2000a) (87.5%). Despite such discrepancy, the same overall conclusion can be drawn: above a certain threshold, it is not worthwhile to invest in contract improvements (for observability purposes) since the detection rate does not improve significantly. In this study, the threshold seems to be contracts of intermediate sophistication since (1) the overall observability only increases by 1.5 percentage points with the use of advanced contracts and (2) the effort in designing an advanced contract is generally significantly higher than an intermediate one for the same method (assuming the method in question cannot be completely specified using an intermediate contract).

6.1.1.1 Observability: impact of updating contracts

Figure 7 shows that 47% of the detected faults were found without updating contracts (cumulative of contracts unchanged, i.e., 36.3 over 77.3). This can be interpreted as the ability of developers to produce contracts to be used as test oracles. This interpretation is valid because contracts were designed by an expert of the system with the source code
available, as described in section 5.3.1. Basic contracts were the ones that needed the least changes to detect additional faults since they are intuitive to design: 75% of faults detected by basic contracts are detected prior to changes (36.8 over 42.4). This should not be a surprise since they can only state predicates based on concrete fields and methods and cannot state properties of elements of collections, which would require the use of quantifiers. In other words, the language constraints make the contracts simple to write and unlikely to improve.

The elementary contracts all had to be updated to enable them to detect additional faults (10.6% increment over Basic). This is because, to increase detection, they required the use of ghost fields to model behaviour spanning multiple objects, each of which were required to be in specific states. Such complex behaviour was not specified a priori since it was not intuitive to do so. A typical example is attempting to log a subscriber in (section 5.2) if, and only if, a default set of policies have been successfully applied on the interface terminating the traffic originating at a subscriber’s equipment. This behaviour is enforced implicitly by a set of objects but is not actually reflected in any concrete field of an object for direct verification anywhere. This is the type of behaviour that is not intuitive to capture since it requires a ghost field to be set in multiple objects (in this example by the objects executing the subscriber log in) so it can be read in different contracts (located in the object controlling the policy application in this instance). Model fields did not play a role since they are a convenience to model abstract behaviour already reflected in concrete fields and thus they do not enable the specification of any behaviour that is not possible to describe with basic contracts.

The vast majority (80%) of intermediate contracts were updated. This is due to the difficulty to determine which properties of collections are important to specify in contracts and the effort required in doing so. The hardest task is to state the subset of elements to which a predicate must hold in case such predicate specifies a relation to other elements of the same or of a different set. Let us take the example of a transaction, which is a set of policy objects to be applied on a router. For any object of type PolicyList present in a transaction, there must be at least one object of type RuleSet associated with it in the same transaction in case these objects are being installed on the
device. There are many properties like this that require nested quantification (two in this example) and not all of them were deemed necessary when specifying the initial contracts. Additionally, not all of them helped improve fault detection in our study.

We needed to update all the advanced contracts. Since this only resulted in a small increase in fault detection, we cannot provide a general explanation as we did for other types of contracts. However, we conjecture that, similarly to Elementary contracts, additional contracts useful for fault detection specify behaviours that are not intuitive to capture.

6.1.1.2 Observability: undetectable faults

As pointed out in (Baudry et al., 2000a), contracts describe some “important” behaviours but they are not expected to be a complete description of a method. The difficulty is therefore to determine which behaviour is important to specify. One aspect of this fundamental question is determining what contracts cannot detect (research question 4). Table 4 summarizes the reasons for failing to detect faults in our case study. Notice that these faults cannot be detected regardless of the effort employed in updating contracts. This conclusion is possible because the target system was assumed correct, the injected faults were known and the experimenter is an expert on the system.

<table>
<thead>
<tr>
<th>Reason for non-detection</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safepoint constraints</td>
<td>6.7%</td>
</tr>
<tr>
<td>Incorrect timeout</td>
<td>6.7%</td>
</tr>
<tr>
<td>Parsing error</td>
<td>13.3%</td>
</tr>
<tr>
<td>Missing functionality</td>
<td>20.0%</td>
</tr>
<tr>
<td>Internal method state</td>
<td>53.3%</td>
</tr>
</tbody>
</table>

Only 6.7% of the undetectable faults are a direct consequence of the constraints imposed by the use of safepoints (Araujo et al., 2008). These constraints prevent the specification of certain properties that would otherwise be able to detect such faults. These would be expressions referring to fields or parameters that would be subject to interference.
Using an incorrect value for a timeout scores the same: 6.7%. This type of error cannot be detected because it is a value that requires an agreement between two entities external to the program (e.g. processes, a process and the operating system) and any contract would just be referring to the same value the implementation refers to. If, however, the timeout is set by the program, then it is under its control and can therefore be checked. This type of occurrence is not considered undetectable and therefore excluded from this category.

Parsing errors of communication protocol messages cover 13.3% of the cases. They are undetectable because the contract would have to fully describe the actual parsing routine, which is not realistic since, in practice, contracts should remain at a higher level of abstraction than the implementation they specify. Notice that we are not stating that it is not possible to define contracts for the parser itself (it was not done so since this functionality resides outside the router driver subsystem) but that, from a client perspective, stating the acceptable contents of a message is equivalent to writing the code that drives the parser.

Missing functionality (e.g., unimplemented event handlers) accounts for 20% of the cases. Contracts cannot detect such faults unless an observable state change is required as a consequence of the missing statements in a method. For instance, if handling a particular type of message requires a subscriber to be logged out, a violation of such rule can be detected by a contract since it can specify which message types are acceptable when a subscriber is logged out. On the other hand, unobservable (or unpredictable) actions, like what happens while handling an event notifying the system of a configuration change, cannot be enforced to happen or not to happen: it would require some observable state change to be checked by such contract.

Internal method state (e.g., failing to check for null) is the top ranking reason (53.3%) for failing to detect a fault. It is impossible for contracts to detect such situations since there is no property observable in the pre- or post-state of a method that would indicate or control whether certain actions are performed. For instance, in the case of accessing a field that is allowed to be null and failing to check for its nullity inside a method is not a detectable fault since observing a null value for such field is not an unexpected situation.
6.1.1.3 Summary

In this section we studied the impact of the contract sophistication in the ability of the system in detecting faults (i.e. observability). We have split the faults in two groups: detectable (77.3%) and undetectable (22.7%) by contracts. For the former, we noticed that updating contracts over multiple iterations is required to achieve higher observability. We also noticed that there is little gain in investing in advanced contracts over intermediate ones (only 1.5 percent points increase). We also noticed that intermediate contracts are indeed required to achieve a significantly higher observability compared to other contracts of lesser sophistication. Regarding undetectable faults, we classified them into different groups to study their nature. Only 6.7% of such faults are undetectable because of the limitations of the contract specification language (JML), namely the safepoints construct. All the others are a consequence to the intrinsic nature of contracts as test oracles, namely that they can only observe states prior and post method execution, and that they should work at a higher level of abstraction than the method. The only remedy to such cases is to design manual test oracles to cover such behaviour.

6.1.2 Contract complexity vs. faults

We first attempt to identify a relation between the Contract Complexity Measure (CCM) (section 5.3.1) of a contract detecting a fault and the complexity of the method it specifies (research question Q3). A generic approach to determine the contents of a contract based on their complexity relates a structural property of contracts (the CCM) to a structural property of methods (MacCabe’s cyclomatic complexity or VG) and the global observability such contracts achieve. This allows one to derive guidelines for contract content (research question Q5) independently of the behaviour of a method. In other words, the goal is to determine what CCM value to expect based on the corresponding method’s complexity and the targeted observability.

Table 5 depicts the relation between the cyclomatic complexity (VG) of a method and the CCM of the most complex contract detecting a fault in the method. Methods are grouped in buckets of size 2 according to their VG (omitted buckets for methods whose contracts did not detect any faults).
Table 5: Method Cyclomatic Complexity Histogram and associated Maximum Detecting Contract CCM (omitted methods with frequency zero).

<table>
<thead>
<tr>
<th>Method VG</th>
<th>Method Occurrence Cumulative %</th>
<th>Contract CCM</th>
<th>Method VG</th>
<th>Method Occurrence Cumulative %</th>
<th>Contract CCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>71.3%</td>
<td>5</td>
<td>24</td>
<td>97.6%</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>84.6%</td>
<td>12</td>
<td>26</td>
<td>97.9%</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>89.8%</td>
<td>12</td>
<td>30</td>
<td>98.3%</td>
<td>24</td>
</tr>
<tr>
<td>8</td>
<td>91.6%</td>
<td>20</td>
<td>32</td>
<td>98.6%</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>93.4%</td>
<td>23</td>
<td>34</td>
<td>98.8%</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>94.3%</td>
<td>13</td>
<td>38</td>
<td>99.2%</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>95.1%</td>
<td>20</td>
<td>50</td>
<td>99.7%</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>95.8%</td>
<td>13</td>
<td>60</td>
<td>99.9%</td>
<td>35</td>
</tr>
<tr>
<td>20</td>
<td>96.9%</td>
<td>13</td>
<td>More</td>
<td>100.0%</td>
<td></td>
</tr>
</tbody>
</table>

Table 5 shows that there is no simple relation between VG and CCM. It shows that simple methods (VG ≤ 6) have simple contracts (CCM ≤ 12) that are capable of detecting faults. It also shows that very complex methods (VG > 50) can have very complex contracts (CCM > 30), which is intuitive. It also shows that the vast majority of contracts have a CCM smaller than 25. These observations do not allow any further conclusions regarding the relation between a method’s complexity and its contract complexity.

Our next research question is to investigate the relation (if one exists) between the effort spent in designing contracts (i.e., their level of detail) and their ability to detect and diagnose faults. Figure 8 shows the cumulative value of the fault detection percentage as contract CCM increases. It also shows the distribution of detected faults per CCM of the detecting contracts, which we will use to support some of our conclusions. For the proper interpretation of this graph one must keep in mind that a contract’s level of detail (and therefore its CCM) is at the will of the designer and, as a consequence, CCM cannot be used to classify methods in the same manner that VG is used. Specifically, VG is an intrinsic property of each method whereas CCM reflects the behaviour of the method that was deemed important by the designer, e.g., a method computing a statistical test may have a complex control flow (and therefore a high VG) though its postcondition may simply state that the result is in range [0,1] (i.e., very low CCM). Moreover, a contract may reflect the behaviour of the method being specified (which may have a low VG) in conjunction with other methods it calls (resulting in a high CCM).
The histogram in Figure 8 shows that 80% of the observed faults can be detected by the set of contracts with CCM up to 14. Contracts of CCM up to 20 detected 90% of detectable faults whereas a CCM up to 24 detected 96% of them. Given the low precision due to the relatively small population of faults, the CCM to attain the 96 percentile could be safely chosen as 30, the midpoint between 24 and 34, the limits of such interval. This threshold could be used as a guideline to a CCM upper-bound, i.e. when to stop specifying. It should not be used, however, as a goal for all methods since for most simple methods (i.e., those with VG ≤ 6) the CCM remains below 12 (see Table 5). It does apply, however, to most methods with VG > 6, for which contracts with CCM ≥ 20 are common, irrespective of the method’s VG. Methods with VG > 38 are very rare (less than 1%), however, due to their complexity, they are very likely to be faulty (we observed more than one fault in each such method) and, therefore, an effort in designing complex contracts for them is not only affordable but has a high chance of paying off.

The choice of the VG and CCM thresholds is ultimately based on the effort required to design the contracts, the available resources and the desired levels of observability. Figure 9 displays the cumulative distribution of detected faults over CCM values for different types of contract. 100% represents the total number of faults detected by contracts of a particular type. For instance, a value of 96% for intermediate contracts...
means a global observability of 73% (96% of 75.8%, the global observability achieved if all contracts are taken into account; see Figure 7) whilst it means an observability of 41% for basic contracts (96% of 42.4%; see Figure 7). It shows that 97.1% of faults detected by both basic and elementary contracts were found by contracts with a CCM up to 14. Given that basic contracts are much easier to design than intermediate ones, one may choose to restrict contracts to be of the basic type and, therefore, should define a CCM upper bound of 14.

Figure 9: Cumulative distribution of detected faults per detecting contracts’ CCM for different types of contracts.

6.1.3 Diagnosability

Any contract content recommendation should also be based on its effect on fault diagnosability. Table 6 compares the statistics for global diagnosability obtained from the three different measures of diagnostic effort (section 5.1). A Spearman correlation test reveals that the correlation coefficient between each pair of measures is greater than 0.99, thus indicating they are for all practical purposes equivalent measures of diagnosability. The table also shows similar results when restricted to the detected faults only.
Undetected faults comprise undetectable faults (22.7% of all faults, section 6.1.1) and the faults not detected by contracts of a particular type. The increase of the diagnosability mean with the increment in contract sophistication if the population includes the undetected faults (Table 6, column 1) demonstrates an overall diagnosability improvement with increased contract complexity. The converse behaviour if the population does not include the undetected faults (Table 6, column 2) demonstrates that the incrementally detected faults are harder to diagnose.

**Table 6: Statistics for global diagnosability for contracts of different types obtained through varied measures of diagnostic effort.**

<table>
<thead>
<tr>
<th>Diagnosis Scope</th>
<th>All Faults</th>
<th>Detected Faults Only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Standard Deviation</td>
</tr>
<tr>
<td>Basic</td>
<td>0.41</td>
<td>0.52</td>
</tr>
<tr>
<td>Elementary</td>
<td>0.51</td>
<td>0.53</td>
</tr>
<tr>
<td>Intermediate</td>
<td>0.70</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td>Basic</td>
<td>0.42</td>
</tr>
<tr>
<td>Distance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elementary</td>
<td>0.52</td>
<td>0.52</td>
</tr>
<tr>
<td>Intermediate</td>
<td>0.72</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td>Basic</td>
<td>0.41</td>
</tr>
<tr>
<td>Weighed Distance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elementary</td>
<td>0.51</td>
<td>0.52</td>
</tr>
<tr>
<td>Intermediate</td>
<td>0.71</td>
<td>0.60</td>
</tr>
</tbody>
</table>

Global diagnosability does not reflect one important detail: the percentage of faults located in the same method as the detecting contract. Table 7 displays these percentages per contract type; the second column considers only the detected faults for a particular contract type whilst the first considers all the faults, including the undetected ones. It shows that the vast majority of faults, irrespective of the contract type, are detected by the postcondition of the method containing the fault. This aligns with the results from Briand’s study (Briand et al., 2003). Further comparisons of the numerical results is impossible since they used a variant of the distance measure that is not equivalent to the one used in this work. Furthermore, as with global diagnosability, increasing contract sophistication increases the number of faults detected in the same method as the contract postcondition, but the incrementally detected faults are more likely (although not a lot more) to be detected by a contract of another method.

**Table 7: Percentage of faults located in the same method as the detecting contract.**

<table>
<thead>
<tr>
<th></th>
<th>All Faults</th>
<th>Detected Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>35%</td>
<td>82%</td>
</tr>
</tbody>
</table>
As with observability, diagnosability improves significantly with the successive increment in contract complexity (see Figure 10), thus allowing one to establish a cause and effect relation between observability and diagnosability and, therefore, to estimate the gains in diagnosability based on the effort spent on contract design. This aligns generally with the theoretical diagnosability model in (Le Traon et al., 2006).

![Figure 10: Global diagnosability as a function of global observability.](image)

### 6.1.4 Discussion and Summary

This case study arrived at the following conclusions:

1. Overall, observability increases with the increase of contract sophistication.
2. The use of advanced contracts (those containing specially crafted classes to model abstract behaviour) does not increase the observability significantly if compared to intermediate contracts (1.5% increase only, based on Figure 7).
3. Observability is limited to about 76% if contracts are restricted to the intermediate type, whereas it is limited to about 42% for basic contracts.
4. Achieving higher observability is only possible with the use of manually coded test oracles targeted at the behaviours displayed by undetectable faults by contracts (Table 4).

5. A system with contracts with maximum CCM values of 20 and 30 can detect 90% and 96%, respectively, of all faults.


7. More complex methods with VG in range [7, 38] are the ones that should have contracts with CCM between 20 and 30.

8. Methods with VG > 38 are rare and likely faulty thus offering a high chance of payoff to contracts. Therefore, their contracts should be unrestricted in terms of CCM.

9. The three measures of diagnostic effort presented in this paper are statistically equivalent.

10. Global diagnosability has a linear relation to global observability, allowing one to determine the relative effort in diagnosing faults as a function of the targeted observability, which is a function of the contract complexity.

Based on these conclusions, to use contracts as a test oracle in a concurrent system, each contract should possess the following properties to be both effective and efficient (not unnecessarily complex):

A. Be of type intermediate if required to specify the method’s behaviour but never of type advanced for methods with VG ≤ 38.

B. Have a CCM ≈ 12 if the method has a VG ≤ 6. There are cases in which invariants are inherently complex due to other methods in the class, thus providing simple methods with a CCM close to 12 solely through invariants. In such cases, the recommendation is to still specify preconditions and postconditions based on properties unrelated to invariants such as predicates involving parameters and return values. Simply not providing a contract for a method in this case will certainly leave faults related to parameters and return values go undetected.
C. Have a CCM \( \approx 30 \) for methods with VG in range \([7, 38]\). This threshold targets an observability around 75% and thus a diagnosability of around 0.70 (Figure 10). This means that it will reduce the diagnosis effort to only 30% compared to a system without contracts. More importantly, however, is that in the worst case, i.e., a complex fault only detectable by an intermediate contract, the average effort in diagnosing such fault will be less than 9% of the execution thread (a diagnosability of 0.91 from Table 6), thus showing a tenfold improvement compared to an un-instrumented system.

D. For methods with VG > 38, their contracts can have a larger CCM than mandated by the requirements above whilst having manual oracles focusing on the undetectable behaviours, namely message parsing, time dependent actions, or that manipulate a lot of state kept in the context of the method (i.e., local variables).

The CCM and VG thresholds above should be chosen based on the availability of resources to design the contracts and the desired global observability and diagnosability. The graph in Figure 9 can be used to relate the desired observability to the CCM threshold. The VG upper limit (38 in this analysis) should be chosen based on actual data collected for the target system (the 99 percentile for this study, as shown in Table 5). These guidelines are a starting point and can only be deemed generic based on the assumption that our target system is representative of industrial concurrent systems. Replications of our study are of course necessary.

Even if one follows these guidelines, there is a chance that detectable faults will still go undetected. This is evidenced by the significant number of contracts (about 53%) that required improvement to detect additional faults (section 6.1.1.1). However, given that the guidelines previously discussed were based on data collected after contracts were updated, which caused CCM increases, it is expected that the need to update contracts will be significantly lower in practice. In a similar manner to standard methodologies in which test cases (and test oracles implicitly) are updated as new faults are discovered, contracts should also be updated. This will increase the system’s global observability and, consequently, its global diagnosability. More research is required to evaluate the continued use (i.e. across multiple evolving releases) of contracts as test oracles in
industrial systems to identify the gains in comparison to current methods based on manually coded test oracles.

The biggest threat to the generality of these recommendations is whether the structural properties of the chosen system are typical of a larger class of concurrent industrial systems. The method cyclomatic complexity distribution is expected to vary across systems. The effect of such a variation is unknown but it is expected that the recommended VG thresholds above would need to be adjusted. This would likely affect the upper end (38 in this case) to the 99 percentile of the actual target system. The CCM thresholds should, however, be less sensitive to such changes since even simple contracts can detect faults in highly complex methods.

7 CONCLUSION

Different solutions to the problem of applying Design by Contract to concurrent software have been proposed. A complete solution requires being able to (1) specify the behaviour of interest in the form of contracts in a suitable language, (2) translate such contracts into runtime assertion checking code, and (3) embed such code into the program being specified (instrumentation). One important use of executable contracts is to serve as automated test oracles. For any result obtained with the instrumented system to apply to the original one, both systems must be behaviourally equivalent, i.e., the runtime assertion checking code must not produce any side-effects that would cause faults present in one system not to manifest in the other. Contracts are also required to be effective at detecting and diagnosing concurrent faults such as deadlocks and race conditions as well as functional faults in a concurrent environment. Having addressed all the other points in previous work (Araujo et al., 2008) (Araujo et al., 2011a) (Araujo et al., 2011b), this paper focuses on the detection and diagnosis of functional faults.

We identified the challenges of applying DbC to concurrent software as well as the solutions we proposed in previous work. We also summarized the results of our previous work in demonstrating the ability of a RAC compiler enhanced with our concurrent contracts extensions to produce instrumented programs that can be used in place of their
production versions (i.e., they are equivalent) during the system test phase, thus allowing one to conclude that the occurrence of faults in one version corresponds to the occurrence of faults in the other. Finally, we also recapitulated the results of our previous work stating that concurrent contracts are very effective in guaranteeing freedom from interference, which, combined with the equivalence result, ensures the validity of the results obtained.

In this paper, the effectiveness of concurrent contracts as test oracles was evaluated with respect to functional faults. Effectiveness was characterized in terms of two measures: the global observability, which is the probability of oracles detecting faults, and the global diagnosability, which is the average effort to diagnose faults normalized with the size of an execution thread. This study was performed with a representative concurrent industrial software system following a precise and objective experimental methodology, which is also a contribution of this work. We categorized contracts based on their level of sophistication and defined a measure of contract complexity named Contract Complexity Measure (CCM). We also defined a procedure for injecting faults and obtaining the measures above from the detecting contracts, which may require updates to detect such faults.

Likely the most significant contribution of this work was to create a set of clear and objective guidelines that contracts should satisfy to be effective test oracles, balancing the availability of resources to spend in designing contracts with the payoff in increased fault detection and decreased diagnosis effort. Such guidelines are based on structural properties of the contract and the code being specified. We used MacCabe’s cyclomatic complexity as the code structural property and the CCM as the contract structural property. By following such guidelines, we showed a concurrent system instrumented with concurrent contracts can detect about 75% of all functional faults and that the global effort to perform fault diagnosis is reduced by about 70%. More importantly, however, is that the diagnosis effort is reduced by about 90% for the faults that are detected by contracts, a tenfold improvement over systems without contract instrumentation. The guidelines were derived assuming our case study is representative of concurrent industrial systems. Replications of this study are of course necessary.
This work focused on Java as the system programming language and the Java Modeling Language (JML) as the language to define contracts. We believe, however, that our contributions are applicable to other object-oriented programming and specification languages. Having addressed the problems of specification and instrumentation of concurrent programs, and showing that concurrent and functional faults can be detected by instrumented concurrent software, we showed how Design by Contract can be effectively applied to concurrent object-oriented systems.

8 REFERENCES


made easy, in: Meyer, B., Schiper, A., Kohlas, J. (Eds.), Dependable Systems: Software,
Computing, Networks - Research Results of the DICS Program. Springer Verlag,
Heidelberg, Germany, pp. 82-102.

systems designed by contracts, Proceedings Seventh International Software Metrics
USA, pp. 272-284.

components using a genetic analogy, ISSRE 2000 International Symposium on Software

York.

Briand, L.C., Labiche, Y., Sun, H., 2002. Investigating the use of analysis contracts to
support fault isolation in object oriented code, Proceedings of the 2002 ACM SIGSOFT
international symposium on Software testing and analysis. ACM, Roma, Italy, pp. 70-80.

Improve the Testability of Object-Oriented Code. Software - Practice and Experience 33,
637-672.

measurement. IEEE Transactions on Software Engineering 22, 68-86.


Language. Iowa State University, Department of Computer Science.


Elmas, T., Tasiran, S., Qadeer, S., 2005. VYRD: VerifYing concurrent programs by
runtime refinement-violation detection, 2005 ACM SIGPLAN Conference on
Programming Language Design and Implementation, PLDI 05. ACM, Chicago, IL,
United states, pp. 27-37.

multithreaded programs, ACM SIGPLAN/SIGACT Symposium on Principles of
Programming Languages, pp. 256-267.


