# Using Contracts to Guide the Search-Based Verification of Concurrent Programs

Christopher M. Poskitt[1] and Simon Poulding[2]

[1] ETH Zürich, Switzerland, `chris.poskitt@inf.ethz.ch`
[2] University of York, UK, `simon.poulding@york.ac.uk`

**Abstract.** Search-based techniques can be used to identify whether a concurrent program exhibits faults such as race conditions, deadlocks, and starvation: a fitness function is used to guide the search to a region of the program's state space in which these concurrency faults are more likely occur. In this short paper, we propose that contracts specified by the developer as part of the program's implementation could be used to provide additional guidance to the search. We sketch an example of how contracts might be used in this way, and outline our plans for investigating this verification approach.

## 1 Introduction

Concurrency is often necessary if programs are to make the most efficient use of modern computing architectures. In particular, multiprocessor manufacturers have shifted focus from increasing CPU clock speeds to producing processors with multiple cores in the pursuit of better performance. For a program to realise this potential performance improvement it must be able to use more than one of the cores at the same time.

However, multi-threaded programs may exhibit concurrency-specific faults which are both difficult to avoid during development and to identify during testing. *Race conditions* can occur when more than one thread manipulates a shared data structure simultaneously, potentially resulting in the corruption of the data structure and consequential functional faults in the program. Race conditions may be avoided by an arbitration mechanism, such as locking, that controls access to shared resources. However, such mechanisms may give rise to non-functional faults such as *deadlocks* when a cycle of threads are each waiting to acquire a lock held by the next thread in the cycle and so none of them are able to proceed; and *starvation* when one thread is continually denied access to a shared resource as a result of the method of arbitration.

Verification of concurrent programs is thus complicated by the need to consider not only input data, but also the relative times at which key events occur, such as lock acquisitions and accesses to shared resources, across all of the program's threads. The temporal order of such events is referred to as an *interleaving* of the threads. The timings of such key events are not typically constrained and so each invocation of the program can have a different interleaving as a result of non-determinism in the hardware and software platform on which the program runs. Therefore verification techniques—both dynamic testing as well as more formal static approaches—must consider many possible interleavings in order to assess the likelihood of a concurrency fault.

A number of effective search-based techniques have been demonstrated for this purpose, each using a fitness function to guide the search algorithm to interleavings that could cause concurrency faults.

In this short paper, we propose that contracts provided by the developer could provide additional information with which to guide the search, and therefore improve the practicality of verifying concurrent programs using search-based techniques.

## 2 Background and Related Work

One approach to detecting concurrency faults is *model checking*, a static technique that builds an abstract model of the concurrent program from its design or implementation. The model is used to determine all possible states of the program and the valid transitions between these states. By exhaustively analysing all reachable model states for the existence of concurrency faults, all thread interleavings—not just those that occurred in a single invocation of the program—may be verified. However, since the set of model states is formed by a product of the states of each thread in the program, the number of states to be checked grows very quickly with the size of the program and the number of threads.

Alternatively, the set of possible interleavings can be explored during *dynamic testing* of the program itself by exerting control over the relative timings of key events in each thread. This can be achieved by inserting instructions that introduce delays around critical operations such as the acquisition of locks. However, as for the model checking, the number of potential interleavings that must be explored grows very quickly with the program size.

Search-based techniques can be used to locate counterexamples—specific thread interleavings (or equivalently, model states) in which concurrent faults arise—rather than an exhaustive exploration of the entire space. Such an approach does not guarantee the *absence* of such faults, but can demonstrate the *presence* of faults. If the metaheuristic algorithm can efficiently locate regions of the search space (i.e. the set of possible thread interleavings or model states) that potentially give rise to concurrency faults, this is a practical alternative to exhaustive exploration of all possible interleavings/model states.

An important factor that determines the efficiency of a search is the fitness function which guides the metaheuristic algorithm. For many types of concurrency faults, the fault either exists or it does not: for example, a deadlock cannot be 'partial'. Thus a fitness function based only on the existence of the fault itself would provide no guidance to the metaheuristic algorithm, and the function must instead utilise other information to identify interleavings/model states that are 'closer' to one that exhibits the concurrency fault.

For example, Godefroid and Khurshid [4], Alba et al. [1], and Shousha et al. [7], each describe the use of a genetic algorithm to locate deadlocks in models of concurrent programs; Staunton and Clark [8] describe an estimation of distribution algorithm using N-grams with the same objective. The fitness function used by Godefroid and Khurshid utilises the total number of transitions out of model states visited on the path to the current state. The rationale is that minimising this sum will guide the search to states with no outgoing transitions: such a state represents a deadlock. The fitness functions used by the other three algorithms all utilise the number of blocked threads (those waiting to acquire locks) in the current state as one of the metrics to guide the search. The rationale in this case is that the more threads that are waiting to acquire locks, the more likely a deadlock state is to occur.

Bhattacharya et al. [2] use hill climbing and simulated annealing to identify potential race conditions through dynamic testing. A simulator is used to execute the program as this allows control of the thread interleaving by injecting timing delays while removing other sources of non-determinism. The fitness function is based on the timing between write accesses to the same memory location by different threads. The rationale is that by reducing the gap between write accesses, a race condition is more likely to arise.

## 3 Using Contracts to Guide Search

### 3.1 Our Proposed Use of Contracts

The search-based techniques discussed in the previous section used a fitness function to efficiently guide the search to regions of the space of thread interleavings/model states in which concurrency faults are most likely to occur. Nevertheless, some of the fitness functions return values from only a small range of discrete values. For example, the metric of the number of blocked threads—used in the functions that guide the search to deadlock states during model checking—takes only integer values between 0 and the total number of threads in the program. Thus many states may have the same fitness, and the search is provided with no guidance as to how to choose between them in order to reach a state that has, in this example, more blocked threads.

We propose that there is opportunity to provide additional guidance to the search by utilising developer-specified contracts. The objective would be to improve the efficiency of the search algorithm by incorporating additional information into the fitness function.

We do not envisage contracts taking the form of exhaustive specifications. Instead we propose the use of formal contracts of the type used in the *Design by Contract* approach to software development. Examples of this type of contract are the preconditions, invariants and postconditions specified in the Eiffel programming language using the **require**, **invariant**, and **ensure** constructs respectively [5]; and equivalent constructs in the Java Modeling Language (JML) [3]. Such contracts do not necessarily change the semantics of a program unless the developer chooses to enable runtime checking for contract violations (for example, in order to localise bugs in development builds), and there is no requirement on them to be exhaustive. However they do document the behaviour of the program intended and assumed by the developer in a form that might not otherwise be easily inferable from the program code itself. It is this information that we believe could be used to guide the search.

For example, a contract could document the assumption that the developer has made as to how other threads will access the shared resources used by a particular section of code. The search could then attempt to locate thread interleavings that break this assumption using a fitness function derived automatically from the contract. The rationale would be that if the assumption made by the developer can be invalidated, it is likely that such interleavings could give rise to concurrency faults.

Alternatively, a contract could be used to guide the search to particular *data* states that increase the likelihood of concurrency faults. This possibility is motivated by the observation that most of the existing fitness functions consider only metrics related to interleavings—such as the number of blocked threads or time between access to a shared memory location—but not the data that, for example, satisfies guards on code that performs operations likely to cause concurrency faults. This additional information in the fitness function could be used to guide the selection of input data as well as thread interleavings.

Our proposal to use developer-specified assumptions in search-based algorithms contrasts with their use in more analytical approaches. For example, the approach of [10] uses an SMT solver to construct sequences of interfering instructions that drive a program under test to break the assumptions.

## 3.2 An Example

As a motivating example, we consider a program written in concurrent Eiffel with SCOOP (for Simple Concurrent Object-Oriented Programming). SCOOP [5, 6] is an experimental object-oriented concurrency model which has contracts as a central concept, making it an interesting starting point for our work.

An object in the model can be declared with a special type using the keyword **separate**, meaning that applications of routines to it may occur on a different *processor* (an abstraction of threads, physical cores, etc.), and that calls to commands (i.e. routines that do not return results) are executed asynchronously. Every object belongs to exactly one processor; no other processor can access its state. Calls to **separate** objects are only allowed if the current processor *controls* the processors owning those objects; this is guaranteed if they are passed as arguments, in which case they are automatically and exclusively locked for the duration of the routine's execution.

SCOOP supports pre- and postconditions for routines (preceded by **require** and **ensure** respectively). In a sequential setting, preconditions are (optionally) checked before executing the routine. In a concurrent setting, preconditions are interpreted as *wait conditions*. That is, the execution of the routine is delayed until simultaneously the precondition is satisfied and the processors handling the **separate** objects controlled.

Suppose we have a simple program that has a bounded buffer, on which we can store integers and from which we can consume them—provided that respectively, the buffer is not full or empty. We give possible implementations of a `store` routine below, using both the Eiffel SCOOP model (left) and Java (right). Both implementations involve waiting if the buffer is full. In the SCOOP version, an (asynchronous) execution would first wait for the **separate** `a_buffer` object to become available and for the precondition to hold; then, the buffer is locked, a new element is pushed, and the lock released. The Java version is intended to do the same, but when waiting for buffer space to become available (with the call to `buffer.wait()`), relies on an (unspecified) consumer object notifying this `store` thread that it has consumed an element from the buffer.

```
store (a_buffer: separate
    BOUNDED_BUFFER [INTEGER];
    an_element: INTEGER)
  require
    a_buffer.count < a_buffer.size
  do
    a_buffer.put (an_element)
  ensure
    not a_buffer.is_empty
    a_buffer.count = old a_buffer.
        count + 1
  end
```

```
public void store(BBuffer<Integer>
     buffer, int element) {
  synchronized(buffer) {
    while (!(buffer.size() <
            buffer.maxSize())) {
      try {
        buffer.wait();
      } catch
       (InterruptedException e) {}
    }
    buffer.push(element);
  } }
```

We have only given fragments of the whole programs, but already, with the precondition in the SCOOP version, we can infer a "region of interest" in the state space, i.e. a region where concurrency bugs may be more likely to reveal themselves. In our example, this region involves states in which the buffer is approaching its bound. A poor design of the SCOOP program might, for example, lead to a call of `store` waiting for an unacceptably long time, e.g. if consumers are starved of access to a full buffer. In the Java version, threads that are blocked because of a full buffer may never be awoken, for example, if the implementation of consumers fails to notify threads when the buffer is no longer full. These bugs would not be observed outside of that region of interest, and with a sufficiently large bound on the buffer, naive testing strategies might not encounter them. The information provided by the precondition should be incorporated into the fitness function to guide the search towards this region of interest, perhaps by converting the contract's Boolean condition to a metric similar in nature to the branch distance [9] used in other forms of search-based testing. Note that the precondition is essentially exposing information that is present in the Java program, but would be more difficult for search to extract in that form.

This is a simple motivating example, and though illustrated with SCOOP, we hope that the approach will generalise to other concurrent object-oriented languages by allowing routines to be annotated with some notion of contract. Furthermore, the example we considered used a *functional* precondition. We are also interested in how search might be guided by a contract language offering *non-functional* preconditions, such as expressions about patterns of access or deadlock-free resource usage.

## 4 Conclusions and Next Steps

In this short paper, we have proposed the use of contracts in concurrent programs for guiding search-based techniques towards regions of the state space where concurrency faults may be more likely. We placed our proposal within the context of the state-of-the-art, and sketched an example in a concurrent object-oriented programming model to discuss how contracts might be exploited.

Our next step is to empirically evaluate our proposal on realistic software with preconditions expressed in JML, the wait conditions of SCOOP, or another suitable language. We also plan to investigate whether and how search-based techniques can benefit from simple non-functional contracts in the code.

**Acknowledgements.** This work is funded in part by ERC grant agreement no. 29138, CME: Concurrency Made Easy, and by EPSRC grant EP/J017515/1, DAASE: Dynamic Adaptive Automated Software Engineering.

## References

1. Alba, E., Chicano, F., Ferreira, M., Gomez-Pulido, J.: Finding deadlocks in large concurrent Java programs using genetic algorithms. In: Proc. 10th Annual Conference on Genetic and Evolutionary Computation. pp. 1735–1742 (2008)
2. Bhattacharya, N., El-Mahi, O., Duclos, E., Beltrame, G., Antoniol, G., Le Digabel, S., Guéhéneuc, Y.G.: Optimizing threads schedule alignments to expose the interference bug pattern. In: Proc. 4th International Symposium on Search Based Software Engineering. pp. 90–104 (2012)
3. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: Formal methods for components and objects. pp. 342–363 (2006)
4. Godefroid, P., Khurshid, S.: Exploring very large state spaces using genetic algorithms. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 266–280. Springer (2002)
5. Meyer, B.: Object-Oriented Software Construction. Prentice Hall, 2nd edn. (1997)
6. Nienaltowski, P.: Practical framework for contract-based concurrent object-oriented programming. Ph.D. thesis, ETH Zürich (2007)
7. Shousha, M., Briand, L.C., Labiche, Y.: A UML/MARTE model analysis method for uncovering scenarios leading to starvation and deadlocks in concurrent systems. IEEE Transactions on Software Engineering 38(2), 354–374 (2012)
8. Staunton, J., Clark, J.A.: Searching for safety violations using estimation of distribution algorithms. In: Proc. 3rd International Workshop on Search-Based Software Testing. pp. 212–221 (2010)
9. Tracey, N.J.: A Search-Based Automated Test-Data Generation Framework for Safety-Critical Software. Ph.D. thesis, The University of York (2000)
10. West, S., Nanz, S., Meyer, B.: Demonic testing of concurrent programs. In: Proc. 14th International Conference on Formal Engineering Methods. LNCS, vol. 7635, pp. 478–493. Springer (2012)