Demonic Testing of Concurrent Programs

Scott West, Sebastian Nanz, and Bertrand Meyer

ETH Zürich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. Testing presents a daunting challenge for concurrent programs, as non-deterministic scheduling defeats reproducibility. The problem is even harder if, rather than testing entire systems, one tries to test individual components, for example to assess them for thread-safety. We present *demonic testing*, a technique combining the tangible results of unit testing with the rigour of formal rely-guarantee reasoning to provide deterministic unit testing for concurrent programs. Deterministic execution is provided by abstracting threads away via rely-guarantee reasoning, and replacing them with "demonic" sequences of interfering instructions that drive the program to break invariants. Demonic testing reuses existing unit tests to drive the routine under test, using the execution to discover demonic interference. Programs carry contract-based rely-guarantee style specifications to express what sort of thread interference should be tolerated. Aiding the demonic testing technique is an interference synthesis tool we have implemented based on SMT solving. The technique is shown to find errors in contracted versions of several benchmark applications.

1 Introduction

The spread of multicore architectures has established concurrent programming as an increasingly indispensable part of software development, and causes an increasing need for suitable development tools. Of particular importance to industrial applications is support for debugging and testing of concurrent programs; such support is difficult to provide, however, because of the unpredictability and irreproducibility of thread scheduling, which makes interference between threads very difficult to discover. These difficulties have not stopped successful research into concurrent testing tools, e.g. [12, 23, 19, 14, 25]. The focus of this work is to address both the difficulty in finding concurrency bugs, while keeping the process reproducible and modular.

Modularity and reproducibility are, in particular, difficult challenges, as concurrent programs appear to be inherently non-modular and non-deterministic: independent threads carefully manipulate shared-data to work towards a common goal (multiplying a matrix, serving a web-page, etc.). To overcome these challenges, demonic testing as prescribed in this paper takes regular unit-testing of routines and contracts, such as preconditions, and uses them to determine whether a routine will fail in a concurrent setting. A high-level visualization can

be seen in Figure 1, where the program-state is exported during testing to a constraint-based synthesis tool along with a precondition. If the synthesis tool can violate the precondition with the actions of another thread, this indicates that the routine is vulnerable to interference.

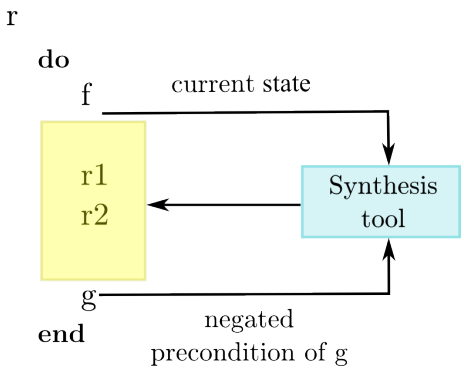

Fig. 1: Finding dynamic interference

The overall process of demonic testing can be seen in Figure 2. Given a set of classes and one of their routines s chosen for testing, we perform:

- **Class-to-domain transformation**: collect all supplier classes for the routine s, and convert them to an abstract domain description for synthesis tool.
- **Routine instrumentation**: instrument the routine s to serialize the state.

These two steps produce a *domain description* and an *instrumented routine* which are passed to the remaining two modules of the system:

- **Testing tool** This component runs the instrumented version of s with relevant test cases. Test cases may for example be obtained from an automatic testing tool, such as AutoTest [18].
- **DemonL (synthesis) tool** This component takes the domain description as input and dynamic state of s recorded by the testing tool and produces sequences of interfering actions.

If the demonL tool finds interference for a test case, the interfering instructions are given. If no such interference can be found then the test succeeds.

An evaluation of the technique shows that it can successfully catch 7 out of 8 selected bugs of the concurrency bug collection [6], which include known bugs in major applications, such as Apache and MySQL – entirely without threads. The implementation of the technique is available [11, 7].

The remainder of this paper is structured as follows. In Section 2 an overview of the approach and a running example are introduced. The overall technique

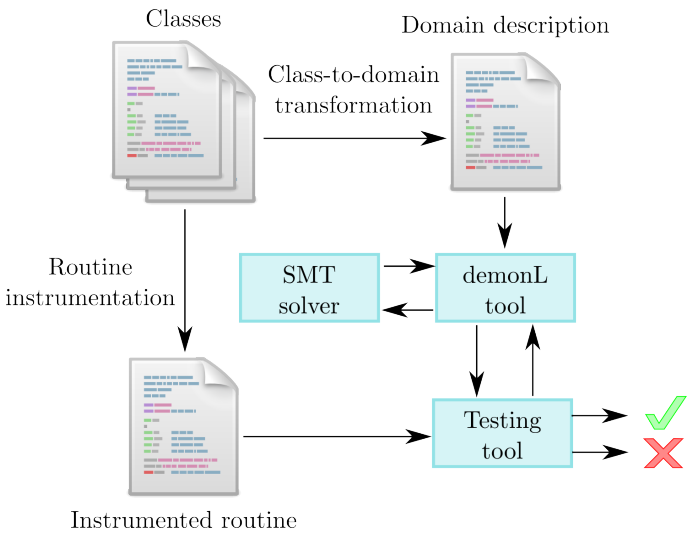

Fig. 2: Overview of the system architecture

is described in detail in Section 3, including the foundational concepts and the transformation of classes into the language of the demonL. We evaluate the technique in Section 4. Discussion on related work follows in Section 5, and we conclude in Section 6.

2 Example testing run

To provide intuition for the demonic testing technique, this section introduces the running example.

The technique works with any language that can carry contracts, including C# (.NET code contracts) [5], Java (JML) [15], D, and Eiffel [17]; in the following we use the Eiffel syntax. Routine contracts in Eiffel are specifications in the form of pre- and postconditions as part of `require` and `ensure` clauses, respectively. The `old` keyword indicates that the value following it will be considered from the pre-state of the routine execution.

To apply the technique to non-Eiffel programs, a first step of translation into Eiffel is currently required. This requirement is a property of our current setup, not an intrinsic limitation of the method of demonic testing introduced in this article, which can be applied to any language supporting contracts.

Example The Eiffel class `IDLE_COUNTER` in Figure 3 represents a collection of idle workers. The idle-counter can increase and decrease the number of idle workers. The `wait_for_idle` routine will decrease the number of idle workers if there are any, otherwise it waits on a condition variable until there are more idle threads. The objective is to test the routine `wait_for_idle` for usage in a concurrent setting.

```
class IDLE_COUNTER
feature
  num_idlers: INTEGER
  increment
    do ... end
  decrement
    require
      non_zero: num_idlers > 0
    do ...
    ensure
      num_idlers = old num_idlers - 1
    end
end
```

```
wait_for_idle
  do
    if num_idlers = 0 then
      mutex.lock
      if num_idlers = 0 then
        -- release locks on mutex
        -- and wait on condition
        non_zero.wait (mutex)
      end
      mutex.unlock
    end
    decrement
  end
```

Fig. 3: Work distribution example

Demonic testing uses the dynamic state at a given program point to analyze statically whether concurrent interference at that point could cause a fault, see Figure 1. We consider a *fault* to exist if the next instruction to be executed could have its precondition violated due to other threads modifying shared state. Part of the response from the static analysis is a sequence of instructions that would move the program into a state that would cause a fault. These instructions represent other threads that may give rise to a failure in the program.

Example While running a unit-test of the `wait_for_idle` routine in Figure 3, the tool instruments the routine `wait_for_idle` with calls to the synthesis tool. For a given test case, the tool reports whether or not interference could be found that will lead to a failure of the routine. In this example, the tool reports that an extra call to `decrement` immediately before the existing call to `decrement` will cause a violation.

There are two ways to respond to a warning by this method: either to modify the behaviour of the program so that it is not vulnerable to this kind of interference, or to refine the specification and only allow certain interference. In the case of Figure 3, synchronization instructions could be introduced to prevent concurrent access of the shared data.

Example The developer can express that the interference found in Figure 3 does not occur by limiting the interference that can be generated. For example, the restriction could be: `num_idlers >= old num_idlers`. This specification disallows the environment from removing idle workers; upon retesting no violations are reported.

3 Demonic testing

This section presents the founding concepts and implementation of demonic testing as well as the approach to handling common synchronization primitives in a thread-free and modular way.

Demonic testing takes classes annotated with traditional contracts and rely-specifications and uses static analysis in combination with the state from runtime to indicate where there may be errors due to concurrent executions.

3.1 Application of rely-guarantee reasoning

The rely-guarantee formalism [13] provides a framework to express and reason about interference in concurrent programs. The interaction between a component and its environment is included in the component's specification, allowing compositional reasoning about concurrent programs.

The formalism proposes an extension of the usual Hoare logic specification (P, Q) of a routine s with precondition P and postcondition Q, to a four-tuple (P, R, G, Q) which additionally contains a *rely*-condition R and a *guarantee*-condition G. The new conditions are binary predicates on states and describe the state changes that the environment (other threads) is allowed to make. A routine s satisfies its specification if, starting in a state satisfying P, under environmental interference adhering to R, s only makes state changes allowed by G, and finishes in a state satisfying Q.

The demonic testing approach uses a subset of the rely-guarantee concepts, namely the *rely*-conditions and the notion of *stability*, to specify interference generation for concurrent programs. The rely-specifications are manually added to the method under test, expressed as a postcondition with the tag `rely`. The `rely` tag indicates that this is only for demonic testing.

The concept of stability allows us to ascertain whether a routine can operate correctly in spite of the interference described by the rely-specification. Formally, the stability of a state-predicate p with respect to a rely-condition R is given as:

$$stable(p, R) \equiv \forall \sigma, \sigma'.\ p(\sigma) \wedge R(\sigma, \sigma') \rightarrow p(\sigma')$$

With the notion of the rely-condition one can express the goal of the testing strategy in the following terms: given the rely-condition R of a routine s under test, try to create interference that would drive the program to violate the precondition pre of some call in the body of s.

Example In the running example, we have the following stability formula for the precondition of decrement:

$$num\_idlers(\sigma(this)) > 0 \wedge num\_idlers(\sigma'(this)) \geq num\_idlers(\sigma(this)) \rightarrow num\_idlers(\sigma'(this)) > 0$$

Demonic testing distinguishes itself from other techniques of program verification by the usage of a dynamic program state to reduce the need for program specification. The goal given to the demonL tool is merely the negation of the *stable* predicate, $\exists \sigma, \sigma'.\ p(\sigma) \wedge R(\sigma, \sigma') \wedge \neg p(\sigma')$. In demonic testing, this is formula simplified by two assumptions:

1. that the routine is correct without interference, and

2. that the test-cases driving the routine constitute the inputs on which it is expected to work correctly.

The first point allows us to assume $p(\sigma)$, the second allows us to remove σ as a quantified expression, as it is given by the dynamic state. This leaves solving only $\exists\sigma'.\ R_\sigma(\sigma') \wedge \neg p(\sigma')$, where R_σ is the rely condition specialized to the concrete program state. Since the rely condition is specialized, it doesn't have to handle cases that never arise in normal program execution; this lowers the amount of required annotation. Additionally, there is no specification required for typically difficult to specify cases, such as loop variants and invariants.

3.2 The domain description language

Program synthesis constructs a program that satisfies a given specification. Demonic testing uses program synthesis to construct interference, actions performed by other threads, which indicates errors in concurrent programs.

Facilitating the demonic testing approach are a language and tool: *demonL*. In the same spirit as the verification language Boogie [2], demonL serves as an intermediate language to express the allowable types of interference. The input to the tool consists of two parts: a domain and a goal.

The domain language is as follows:

$$
\begin{array}{ll}
\mathit{Domain} & ::= [\mathit{TypeDecl} \mid \mathit{ProcDecl}]^* \\
\mathit{TypeDecl} & ::= \textbf{type}\ \mathit{ident}\ \{\mathit{Decl}^*\} \\
\mathit{Decl} & ::= \mathit{ident} : \mathit{ident} \\
\mathit{ProcDecl} & ::= \mathit{ident}(\mathit{Decl}^*)[: \mathit{ident}]?\ \mathit{Pre}?\ \mathit{Post}? \\
\mathit{Pre} & ::= \textbf{require}\ \mathit{TaggedExpr}^* \\
\mathit{Post} & ::= \textbf{ensure}\ \mathit{TaggedExpr}^* \\
\mathit{TaggedExpr} & ::= \mathit{tag} : \mathit{Expr} \\
\mathit{Expr} & ::= \mathit{op}\ \mathit{Expr} \mid \mathit{Expr}\ \mathit{op}\ \mathit{Expr} \mid \mathit{Call} \\
\mathit{Call} & ::= \mathit{ident}(\mathit{Expr}^*)
\end{array}
$$

where *op* can be the common infix and prefix operators, with the addition of an `old` prefix operator.

To specify the desired initial and final states the following goal language is used, sharing the same expression and declaration syntax as the domain format.

$$
\begin{array}{ll}
\mathit{Goal} & ::= \mathit{Decl}^*\ \mathit{InitialState}\ \mathit{FinalState} \\
\mathit{InitialState} & ::= \textbf{initial}\ \mathit{Expr}^* \\
\mathit{FinalState} & ::= \textbf{final}\ \mathit{Expr}^*
\end{array}
$$

Example Figure 4 shows a program written in demonL, corresponding to the class `IDLE_COUNTER` in Figure 3.

The domain describes the state through data structures, functions on the state, as well as procedures that transform the state. Procedures and functions are described with pre- and postconditions. The goal describes the entities in the

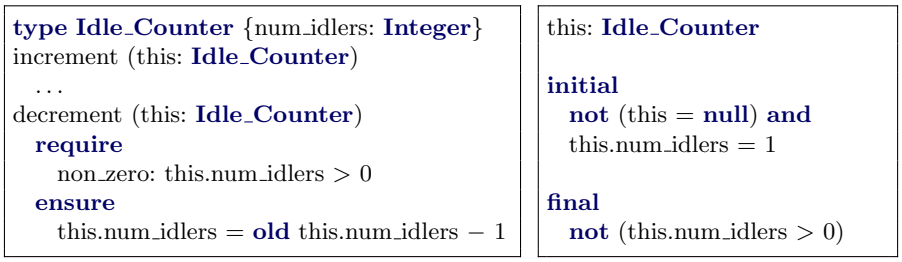

Fig. 4: The `IDLE_COUNTER` class in demonL

system and constraints on the initial state and final state. The final state relates the initial and goal states through the use of `old` operator, which references the values in the initial state.

DemonL constructs an initial state that satisfies the initial constraints, a series of actions, and a final state that is the result of the actions applied in order, and also satisfies the final-state constraints.

Example To find the possible interference that could be used to destabilize Figure 3, the goal specification found in Figure 4 is used. The goal specification contains the negation of the precondition of the `decrement` operation, here: `this.num_idlers <= 0`. However, if the goal includes the rely-condition restricting the interference to only non-decreasing effects on the number of idle workers, then the program is correct under the rely assumption.

```
final
  this.num_idlers >= old this.num_idlers and
  not (this.num_idlers > 0)
```

Again we can see the same shape stability criterion, $\exists\sigma'.\ R_\sigma(\sigma') \wedge \neg p(\sigma')$.

3.3 Class transformation

We assume an input (Eiffel) class C has three components: C_{name}, C_{attrs}, and C_{routines}. C_{name} denotes the name of the class. The attributes of the class, C_{attrs}, are denoted by $a : t$ to indicate an attribute a that has type t. Every routine s in C_{routines} has a name, denoted by s_{name}. Also, every routine can have a pre- and postcondition, denoted by s_{pre} and s_{post}.

The translation function to convert class files into demonL domains (see Figure 2) is shown in Table 1. Note that the presentation of this translation function uses a pattern-matching style, with the function matching arguments in a top-down fashion.

– Attributes, along with the class name, are transformed into a datatype in demonL.

$$
\begin{array}{rl}
trans(C) & = \{feat(C, f) \mid f \in C_{\mathsf{features}}\} \cup \{data(C)\} \\
\hline
data(C) & = \textbf{type}\ C_{\mathsf{name}}\ \{\ C_{\mathsf{attrs}}\ \} \\
feat(C, f) & = f_{\mathsf{name}}(args(C, f_{\mathsf{args}})) \\
 & \quad \textbf{require}\ expr(f_{\mathsf{args}}, f_{\mathsf{pre}}) \\
 & \quad \textbf{ensure}\ expr(f_{\mathsf{args}}, f_{\mathsf{post}}) \\
\hline
args(C, \boldsymbol{as}) & = (this : C_{\mathsf{name}}) :: \boldsymbol{as} \\
\hline
expr(\mathsf{args}, x.f(\boldsymbol{as})) & = f(expr(\mathsf{args}, x), expr(\mathsf{args}, \boldsymbol{as})) \\
expr(\mathsf{args}, e_1\ \mathsf{op}\ e_2) & = expr(\mathsf{args}, e_1)\ \mathsf{op}\ expr(\mathsf{args}, e_2) \\
expr(\mathsf{args}, \mathsf{op}\ e) & = \mathsf{op}\ expr(\mathsf{args}, e) \\
expr(\mathsf{args}, v) & = \begin{cases} v & \text{if } v \in \mathsf{args} \\ this.v & \text{otherwise} \end{cases}
\end{array}
$$

Table 1: Translation function

– Routines are transformed using *feat* directly into demonL procedures with pre- and postconditions.

The result of a function is denoted by having equality on the `Result` value, for example `Result = 2 * x`. Argument-list transformation of routines and functions explicitly includes the normally implicit self-reference in object-oriented programs. The translation of expressions is largely straightforward, with the target of a call moving to the first argument of the call, to coincide with the argument-list transformation.

Example An example of this translation process can be seen by examining how Figure 3 is translated to Figure 4.

3.4 Routine instrumentation

As part of the technique, the routine under test must be instrumented (see Figure 2). The instrumentation augments the program execution so it is able to encode the dynamic state of the program for demonL. This procedure is straight-forward.

3.5 The demonL tool

The output of the tool is the sequence of actions, and their arguments, that bring the program from the initial to the final state. Given the specifications in Figure 4, this would be a call to `decrement`. If the underlying SMT solver reports that the constraints are unsatisfiable, this indicates that no sequence of actions could be found. To avoid long synthesis times, the tool constructs sequences bounded by number of instructions and number of unique references for each user-constructed type.

However, because of the constraint-based nature of the encoding, first the tool solves the interference problem in a single step with *no* actions to constrain the

transformation. This is equivalent to a *proof* of instability. If the tool determines that interference is possible then it tries to obtain the sequence of actions. If, even without constraints, it cannot find interference then interference is impossible no matter the actions or bounds given to the tool. This means that demonL's determination of the absence of interference is not limited by the inability of the tool to construct a sequence of appropriate instructions.

Having an intermediate language and tool offers substantial advantages to the application of the demonic technique: separating the complexity of encoding the verification conditions from the task of routine instrumentation, and the possibility to target more than one source language and and more than a single solver in the back-end. The current technology choices for demonL are Eiffel as a source language to be translated to demonL, and Yices [9] as the SMT solver.

DemonL is similar to planning tools. In particular it allows the movement from an initial state to a final state by a series of actions. However, the specification of the initial state and the actions are permitted to be weaker than generally allowed by planning tools that use languages such as the Planning Domain Definition Language (PDDL) [16]. Where PDDL only allows the effect of an action to be expressed using certain atomic-terms our tool has no such restriction: any expression can be used to describe the effect of an action. For example, where a PDDL domain would require a post-condition such as `attribute = 5`, demonL is able to deal with with post-conditions such as `attribute > 3`. DemonL also does not assume determinism of the actions. These qualities are important when representing program specification, which are typically incomplete.

DemonL is available for download from [7].

3.6 Handling synchronization primitives

The use of threads to construct concurrent programs inherently exhibits two types of effects:

- the *necessary*, where a thread contributes a result to another thread, and
- the *incidental*, which are side-effects of necessary actions, and are also modifications to shared state.

When we consider concurrent applications as a combination of necessary and incidental effects, the necessary aspect of concurrency can be seen as a dependency, and the incidental aspect can be seen as interference. One thread depends on another to provide a computational result in a shared memory location. In threaded programs, these dependencies are made explicit by a mutex's `lock`, or a condition variable's `wait` routine.

When unit-testing a class or method, it is common to provide stub methods or objects in the place of dependencies. For example, a full database connection may be replaced with one containing only a small fixed selection of data.

Although mutexes, semaphores, and condition variables carry no explicit invariants, their usage in programs is almost always accompanied by an implicit invariant related to a resource. Consequently, they can have meaningful post-conditions that we can use to create stubs to test concurrent programs without

requiring threads. They merely need to be replaced with normal function calls that ensure the same postcondition.

Example Assume a simple producer/consumer-style program, such as that given in Figure 5. The call to `cond_var.signal` in the `produce` routine has the precondition that the number of products is greater than zero. The counterpart in the `consume` routine, the call to `cond_var.wait`, has the same post-condition: `product > 0`.

```
produce
  do
    product := product + 1
    if product = 1 then
      cond_var.signal
    end
  end
```

```
consume
  do
    if product = 0 then
      cond_var.wait
    end
    product := product - 1
  end
```

Fig. 5: Producer/consumer coordination

To create a stub for the call to `cond_var.wait`, replace the implementation of `wait` on the condition variable with

```
wait do product := product + 1 end
```

The new `wait` satisfies the invariant for the condition variable, and requires no other thread to work. The corresponding stub for `signal` would similarly have `product > 0` as a precondition and an empty body.

4 Experimental evaluation

It is essential for a testing technique to be judged by its reaction to bugs that occur in real software. For this purpose, we use a selection of bugs from a concurrency bug database [26, 6] to determine if demonic testing can detect and help form fixes for the faults. No particular criteria was used to select bugs from the database, besides striving for an overall diversity of faults. All experiments were carried out on an Intel Q6600 2.4GHz with 4GB of RAM.

4.1 Conversion from source programs

All of our test cases are extracted from real projects and translated into Eiffel. Since well-known concurrent applications with specifications are rare, we slice the non-essential elements from well-known code then convert it to Eiffel and add contracts. This is also done to enable the analysis of bugs from many languages, while minimizing the differences due to language features. To see an example of this process, the original Apache C-code for the running example is given in Figure 6. The main differences are the removal of the recycled pool functionality,

and the removal of the explicit return-value checking of concurrency primitive (locks, condition variable) operations that is typically handled by exceptions in languages that support them.

```
apr_status_t ap_queue_info_wait_for_idler
  (fd_queue_info_t *queue_info,
   apr_pool_t **recycled_pool)
{
  apr_status_t rv;
  *recycled_pool = NULL;
  if (queue_info->idlers == 0) {
    rv = apr_thread_mutex_lock(
      queue_info->idlers_mutex);
    if (rv != APR_SUCCESS) {
      return rv;
    }
    if (queue_info->idlers == 0) {
      rv = apr_thread_cond_wait(
        queue_info->wait_for_idler,
      queue_info->idlers_mutex);
      if (rv != APR_SUCCESS) {
        apr_status_t rv2;
        rv2 = apr_thread_mutex_unlock(
          queue_info->idlers_mutex);
        if (rv2 != APR_SUCCESS) {
          return rv2;
        }
        return rv;
      }
    }
    rv = apr_thread_mutex_unlock(
      queue_info->idlers_mutex);
    if (rv != APR_SUCCESS) {
      return rv;
    }
  }
  apr_atomic_dec32(&(queue_info->idlers));

  ... recycling of data structures
}
```

Fig. 6: Original `wait_for_idle` routine from Apache

4.2 Results

Table 2 lists the collection of concurrency bugs that we use to perform our evaluation; the first seven are from the bug database, with the last being a well-known Java standard library bug. All bugs have been replicated using the demonic testing technique, with the exception of MySQL #169, as explained in the discussion at the end of the section. Inspired by the AutoTest approach, work initially began using Eiffel as the source language; to broaden the scope of the evaluation we translated bugs from multiple other languages. These examples are available for download [7].

The time taken to generate interference, or determine that none exists, was measured for the bugs that were successfully tested. The average time taken was 100ms for each request to demonL. This time is different from the times in Table 2, as each time in the table may include many requests to demonL.

The rest of this section analyzes the effort required to write the necessary program contracts, the conversion process, and concludes with a discussion of the notable properties of the technique.

4.3 Annotation complexity

In any approach which requires the addition of specification via program annotation, the burden that this annotation places on the programmer is highly relevant. Although difficult to measure objectively, we place the annotations into three categories:

	Program/Bug	Bug type	LOC	Annotation			Time (s)
				Lock	Simple	Complex	
1	Apache #21285	Atomicity violation	125	0	4	0	0.982
2	Apache #25520	Data-race	101	0	2	0	0.124
3	Apache #45605	Data-race	227	1	4	0	0.217
4	MySQL #169	Atomicity violation	69	–	–	–	–
5	MySQL #644	Data-race	124	0	3	1	0.939
6	MySQL #791	Data-race	113	0	1	0	0.139
7	pBZip2	Order violation	168	1	1	0	2.289
8	Java Vector	Data-race	70	0	2	0	0.032

Table 2: Bug collection

- Lock – a rely-annotation denoting that a lock protects some shared data from change by another thread.
- Simple – a non-concurrency-related program annotation stating a property of the program that is either a non-null check for a reference, or a linear equation.
- Complex – a non-linear expression, or a frame condition that is necessary to limit the scope of an operation.

Table 2 collects the types of annotations required in the test cases. These are the types of annotations required for demonic testing to give the correct cause of the bug in the full program in the cases of Apache, MySQL, and pBZip2. In the Java vector implementation one of many possible causes is given, as it is part of a library.

4.4 Discussion

The Apache bug #45605 example is notable due to the the double-check present in the `wait_for_idle` routine. Separate tools exist to classify some data-races as "potentially benign" [20]; the double-check pattern is benign and difficult for pure data-race checkers to deal with. Demonic testing does not require any secondary approaches to accomplish this: the determination of benign vs. malignant data-races is based on the program contracts.

The only bug from our test set which could not be discovered using demonic testing came from MySQL bug #169. The reason is that the invariant of the program could not be expressed without either ghost variables or artificially adding more data.

Incorrectly stated rely conditions will lead to both false-positives and false-negatives, as these essentially form an axiom of the routine to which they belong. However, as in Table 2, all rely conditions required were of a very plain type, merely indicating that a certain lock protects some shared state.

The bounded synthesis done by demonL may affect the results by not considering interference from sequences of instructions that exceed the bound. However, this bound concerns the search for instruction sequences; there is also an

initial unbounded-verification that demonL performs to determine the stability before trying to synthesize interference. The worst case is that the tool is unable to find the sequence of actions but still reports whether interference is possible. All bugs our evaluation examined required only single-action interference to become evident. This suggests that many concurrent bugs manifest themselves with little prompting; and that causing errors to present themselves in threaded executions is difficult due to scheduling rather than maintaining very complicated invariants.

This experimental method is limited by the number and selection of examples, it is possible that drawing on a larger pool of examples would offer greater insight into the properties of demonic testing. However, the small sample size is mitigated by the wide variety of types of concurrency bugs. Although every effort was made to make a faithful reproduction of the programs in the target language, there is the possibility transcription errors while moving between different programming languages.

5 Related Work

The idea of using routine specifications to discover concurrency errors is not unique to demonic testing. The Colt tool [24] for Java also uses this approach. However, their approach is less general, relying hard-coded specification of the existing Java concurrent collection classes. Demonic testing is more generic as it works with user-supplied classes and specification, and as well allows finer-grained control of what constitutes an error through the usage of rely-conditions.

A common practice for testing of concurrent programs is load or stress testing. This frequently proves to be ineffective as in typical testing environments interleavings might only change marginally from one test run to the other. To force different interleavings, Edelstein et al. [10] present the ConTest tool, which combines a technique for deterministic replay of concurrent programs [4] with a heuristic for varying thread schedules by seeding sleep calls at synchronization points in the program.

Dynamic model checking [12, 19, 25] provides a more systematic approach by systematically exploring all possible thread interleavings. The search is stateless in that it provides a specialized scheduler that runs the program in its real execution environment, and hence can avoid storing concrete program states. The main problem is to overcome state explosion, which makes brute-force exhaustive search infeasible for large applications. Techniques such as partial order reduction as employed in the VeriSoft tool (Godefroid [12]) or preemption bounding (giving priority to schedules with fewer preemptions) in the CHESS tool (Musuvathi et al. [19]) can mitigate the effects of state space explosion only to a small degree. Wang et al. [25] propose a heuristic where ordering constraints learned from successful runs are used to guide the selection of interleavings for future runs. All of the above works focus on varying thread interleavings to produce undesired behaviour. Demonic testing differs from this approach by considering

the routines in a program and finding sequences which lead to the violation of a program invariant, avoiding an exhaustive search of interleavings.

A number of works use combinations of dynamic and symbolic analyses to improve testing of concurrent programs. Sen and Agha [23] use a combination of concrete and symbolic execution, termed concolic execution, to test multithreaded Java programs with the tool jCUTE. Symbolic execution produces input values that guide the concrete execution to alternate paths; concrete execution guides the symbolic computation along a concrete path to concretize any values that cannot be handled by a constraint solver. Besides producing alternate input values, their technique also systematically generates thread schedule variations such that potentially all causal structures of a concurrent program can be explored. Sen [22] introduces RaceFuzzer, an algorithm which uses race warnings from race detection tools to create problematic interleavings during testing in order to eliminate false positives automatically. Park et al. [21] propose CTrigger, a testing tool to expose atomicity violation bugs. The tool analyzes traces to find unserializable interleavings then these interleavings are explored during testing to expose bugs. Kundu, Ganai, and Wang [14] present a framework that combines conventional testing with symbolic analysis. A test harness invokes the program with random test values. Concrete traces are relaxed into concurrent trace programs, which capture all linearizations of events that respect the control flow of the program. The concurrent trace programs are then symbolically verified. All these techniques combine in some way the dynamic execution of programs with symbolic computation and verification, and most closely resemble the work presented in this paper. However, they, like all other related work shown, are not able to achieve truly modular testing of concurrent software; they all depend on multithreaded executions or traces.

Contracts have been used successfully in unit testing of sequential software [18], where they can provide test oracles and filter inputs for random testing. Araujo et al. [1] evaluate the use of contracts in a concurrent setting, based on an extension of the JML [15] contract semantics. They found contracts as test oracles effective in finding and diagnosing concurrency-related faults on an industrial case study in Java/JML. In contrast to this work, demonic testing emphasizes the use of contracts also for symbolic analyses, in addition to test oracles.

Rely-guarantee reasoning has been applied in testing of concurrent programs. Dingel [8] uses the state exploration tool VeriSoft [12] for rely-guarantee verification of C/C++ components. The component code is executed in parallel with an environment which generates initial states, monitors the component execution, and generates responses. If a program step is found to violate one of the guarantees, a flaw is found. Blundell et al. [3] use labelled transition systems to model the behaviour of components, whereas demonic testing works directly on source code. Assumptions on the model-level are used as environments in which individual components are executed. The execution results in traces which are in turn checked against the guarantees of the model. Failure of a check suggests an incompatibility between a model and its implementation.

6 Conclusion

Until recently, the testing of concurrent systems has generally been regarded as inferior to static approaches. The realization that purely static reasoning also faces problems of scalability or precision when applied to concurrent systems has led to a more pragmatic assessment, leaving testing its due place, as evidenced by the approaches reviewed in the previous section.

Unlike many of these approaches, which are only suitable for testing entire systems, demonic testing can be applied to the important problem of unit testing for concurrent programs. Through its combination of dynamic and symbolic techniques, demonic testing provides two significant benefits over other proposals. First, it leverages available testing tools for sequential programs, which it uses as an essential part of its architecture. Second, instead of searching the state space of thread interleavings, demonic testing uses program synthesis as a constructive means to find problematic thread interference. If a test fails, a test case and a problematic sequence of interactions is available for analysis.

Acknowledgments

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, and ETH (ETHIIRA). Earlier work has also benefited from grants from the Swiss National Foundation and Microsoft (Multicore award).

References

1. W. Araujo, L. Briand, and Y. Labiche. On the effectiveness of contracts as test oracles in the detection and diagnosis of race conditions and deadlocks in concurrent object-oriented software. In *Proc. ESEM'11*. IEEE Computer Society, 2011.
2. M. Barnett, B. E. Chang, R. Deline, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. FMCO'05*, number 4111 in Lecture Notes in Computer Science, pages 364–387. Springer, 2006.
3. C. Blundell, D. Giannakopoulou, and C. S. Păsăreanu. Assume-guarantee testing. In *Proc. SAVCBS'05*. ACM, 2005.
4. J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proc. SPDT'98*, pages 48–59. ACM, 1998.
5. Code contracts. `http://research.microsoft.com/en-us/projects/contracts/`, 2011.
6. Collection of Concurrency Bugs. `http://www.eecs.umich.edu/~jieyu/bugs.html`, 2011.
7. Demonic test case downloads. `http://se.inf.ethz.ch/people/west/demonic-cases/`, 2011.
8. J. Dingel. Computer-assisted assume/guarantee reasoning with VeriSoft. In *Proc. ICSE'03*, pages 138–148. IEEE Computer Society, 2003.

9. B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. CAV'08*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.
10. O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.
11. EVE project. `https://svn.eiffel.com/eiffelstudio/branches/eth/eve/`, 2011.
12. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. POPL'97*, pages 174–186. ACM, 1997.
13. C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981.
14. S. Kundu, M. K. Ganai, and C. Wang. Contessa: Concurrency testing augmented with symbolic analysis. In *Proc. CAV'10*, volume 6174 of *LNCS*, pages 127–131. Springer, 2010.
15. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31:1–38, 2006.
16. D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL: The planning domain definition language. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control, 1998.
17. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.
18. B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *IEEE Computer*, 42:46–55, 2009.
19. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proc. OSDI'08*, pages 267–280. USENIX Association, 2008.
20. S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data racesallusing replay analysis. *ACM SIGPLAN Notices*, 42(6):22–31, 2007.
21. S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *Proc. ASPLOS'09*, pages 25–36. ACM, 2009.
22. K. Sen. Race directed random testing of concurrent programs. In *Proc. PLDI'08*, pages 11–21. ACM, 2008.
23. K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proc. CAV'06*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.
24. O. Shacham, N. G. Bronson, A. Aiken, M. Sagiv, M. T. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *Proc. OOPSLA'11*, pages 51–64, 2011.
25. C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *Proc. ICSE'11*, pages 221–230. ACM, 2011.
26. J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proc. ICSA'09*, pages 325–336. ACM, 2009.