Tapir: Language Support to Reduce the State Space in Model-Checking

Ronald Veldema Michael Philippsen

University of Erlangen-Nuremberg
Computer Science Department 2
Martensstr. 3 • 91058 Erlangen • Germany

veldema@cs.fau.de philippsen@cs.fau.de

Abstract:
Model-checking is a way of testing the correctness of concurrent programs. To do so, a model of the program is proven to match properties and constraints specified by the programmer. The model itself is created by disregarding irrelevant program details.

The biggest problem in model-checking is the number of program states that need to be tested. Tapir, a simple but familiar object-oriented language and accompanying tool chain, addresses this problem in two ways. First, the programmer can provide application specific program transformations that reduce the state space. Second, for selected classes and methods, the programmer can provide an alternative implementation: a slim black box version for use in model-checking that abstracts away many details of the full fledged implementation.

Tapir's aspect-oriented test case generation combined with black-boxing allows model-checking of low-level library code.

1 Introduction

System software is critical software that includes such things as operating systems, networking libraries, and middle-ware. In general system software provides some service to application code, and is hence essential for system stability. There is therefore a need for 'extreme testing' of system code which we implement by means of model-checking. Because system software is often used by multiple concurrent threads or processes, the software must be made safe with respect to concurrency. Model-checking provides this concurrent testing naturally.

Model-checking a large library or complex service provider is hard due to two reasons. First, model-checking requires complete, self-contained programs which by definition libraries or service providers are not. To model-check these, a service user (i.e. a test case) must be supplied as well. For exhaustiveness, however, *many* service users are required, better still, in combination with each other to increase test coverage. Second, model-checking itself is time-intensive as it performs a complete state enumeration over a huge

state space. This requires a large memory to hold all computed states (to be able to detect cyclic program states) and a lot of time to perform the state space enumeration itself.

Tapir has solutions to both of the above problems. First, Tapir supports Aspect Oriented Programming (AOP) to weave a service or library together with one or more service users creating many different complete (test-)programs. Second, Tapir allows the programmer to aid in state space reduction by both providing program transformations that reduce the model-checker's state space and by providing an additional slimmed version of a class that is used during model-checking.

The Tapir language is a subset of both Java and C++ with their worst and unneeded features removed and some new features added. As such, Tapir's language should be familiar to many. Tapir adds aspect orientation (covered in Sec. 3) for comfortable test case generation. Compile-time predicates, program transformations and analysis rules are added to enable state space reductions (see Sec. 4). The base Tapir language features:

- classes and templates, but no inheritance,
- no type casts, pointer arithmetic, nor function pointers,
- no exception handling support,
- no static/global variables,
- some functional programming support.

All these restrictions allow for easier program analysis and thus indirectly make state-space reducing optimizations easier: the first two items provide strong static typing and eliminate some types of errors. Removed exception handling simplifies control-flow. Removed support for global variables and added support for functional programming eliminate sources of non-local reasoning which eases program understanding.

Parallelization and distributed programming are explicit. In Tapir, a class can be marked with *process* or *thread*. A process class will have its *run* method invoked at virtual machine start-up. When a process is running it can create instances of thread classes and start them by invoking their *start* methods. Process classes communicate among each other via Remote Procedure Calls (RPC) in active-message fashion [vECGS92]. This level of communication abstraction allows systems programs to be written without knowledge of communication media. Communication between threads is via process memory.

The Tapir tool chain (see Fig. 1) cannot only generate a C/C++ implementation from the Tapir code. It can also generate program specific model-checkers (one generated per test case). When generating the (C++) implementation, the weaver removes all statements from the Tapir program that deal with AOP. What remains is a true library. That code is passed to the Tapir compiler which generates a C++ library from it. For model-checking, the aspect weaver generates all possible permutations of weavings to create the test case programs. Each of these new Tapir programs (now without the AOP statements) is passed to the Tapir compiler to generate a program (test case) specific model-checker. Each of these model-checker programs is generated in Java, in the same way that the Spin [Hol97] model-checker generates its C code for the Promela language. Such a Java program is then fed to a JVM for execution. Because of the model-checker's large memory requirements, a specialized JVM such as LVM (Large Virtual machine) [VP07] is preferable. LVM's language extensions for memory optimizations and support for distributed execution on a

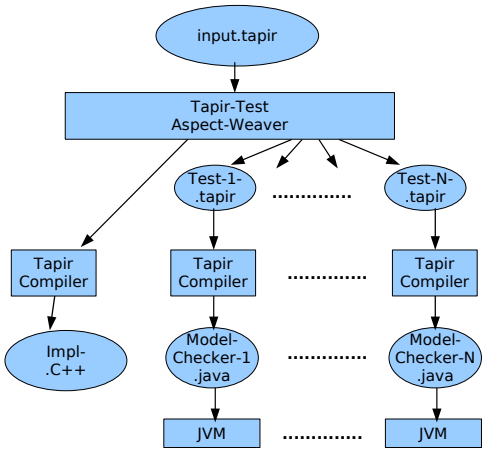

Figure 1: Tapir tool chain.

cluster meet the model-checker's large memory requirements.

The main contributions made in this paper are: (1) a technique to allow the programmer to specify, at language level, multiple implementations of a class, method, or statement (a version used during model-checking and a version used during code generation), (2) a way to allow the programmer to aid the compiler in creating application specific optimization rules to reduce the search space.

2 Self-consistency

Typical model-checking approaches express properties external to the code or use a formulaic approach. We take a code-centric approach.

We call an application self-consistent if all incorporated checks succeed for all possible runs. In Java/C/C++, such checks are often implemented by assert statements: `assert(X)` where X evaluates to a boolean condition. If X evaluates to false the program aborts, if true, nothing happens. These assert statements are then placed all over the program to test the self-consistency of its data structures.

For many applications, it is also important to know that eventually an event will occur, that a given program state will be reached, or that after some event some other condition should hold. Although many model-checking environments provide specialized formalisms to express such properties, they can also be explicitly programmed by means of asserts with the help of counters. E.g., to test that after event X, eventually Y should hold, one could write:

```
boolean X_happened = false;
boolean Y_happened = false;
```

```
int Y_wait_counter = 0;
...
if (X)
   X_happened = true;
...
if (X_happened)
{
  assert {
    Y_wait_counter++;
    if (Y)
    {
      Y_wait_counter = 0;
    }
    assert(Y_wait_counter < MAX_Y_WAIT_COUNTER);
  }
}
```

Linear Temporal Logic (LTL) [DAC99] is a formalism that is meant to describe such event sequences (until/after X, Y should hold, etc.). CTL (Computation Tree Logic) [DAC99] extends LTL by adding that a condition holds on all possible paths following a program state. By using model-checking we can guarantee CTL properties as it tests all possible states.

Some problem domains have explicit time requirements as well. Again, instead of a special formalism to express real-time properties, we support a minimalist approach by providing a 'system.currentClockTick' method so that programs can check their to-be-checked properties with asserts as well. When model-checking, the currentClockTick intrinsic returns the number of executed instructions of the non-deterministic machine. When used for model-checking, it will return the number of instructions executed by the simulated machine. When used for code-generation, it works as expected. Using the currentClockTick() method inside assert statements allows us to test real-time properties of programs. For example:

```
long start = system.currentClockTick();
work();
long end = system.currentClockTick();
assert((end-start) < N);
```

when executed under the model-checker, the aspect will test if `work()` will always execute in less than N instructions.

Note that it is the responsibility of the programmer to write sufficient asserts into his code to allow the model-checker to find all errors. While for simple tests automatic assert generation seems feasible, tests of program semantics is definitely the programmer's job.

Unfortunately, the above solutions have the potential to increase the state space as each timer and counter variable becomes part of the state of a process. That can be partially fixed by means of the techniques discussed below.

3 Aspect oriented testing

As Tapir is meant for programming system software, a Tapir program will in general not be complete in the sense that it can link to an executable. What is needed is some start-up environment that invokes the functions of the service provider to allow its functionality to be tested. Moreover, because a library can often be used in different ways (by calling its

methods in different orders, for example), many different individual programs need to be generated where each generated program uses the library in a different way.

To create a test, a set of classes and methods is woven into the library under test. We call such testing code (that consists of ordinary Tapir classes and methods) a 'scenario'. Thus, the Tapir weaver works as follows; for a given joint-point, a number of scenarios may apply. For example, for a join-point X(), multiple scenario's may contain a matching X() method. For each permutation of scenarios that apply to a given join-point, the weaver generates an output file in which each join point is replaced by a call to the method with that name from inside the scenario. When there are multiple scenarios in the permutation, the methods from that scenario are called in the order of the scenario permutation.

```
class FileStream join_point<FS_Mixin>  {
  boolean is_open;
  StorageDevice device;
  Lock l;

  int open(string name) {
    ..
    is_open = true;
    join_point<test_open(ret, name)>;
    return ret;
  }

  byte read_byte(int fd) {
    ..
    Block b = device.get_block();
    ..
    join_point<test_read(ret)>;
    return ret;
  }

  void close(int fd) {
    ..
    is_open = false;
    join_point<test_close(fd)>;
  }
}
```

Figure 2: A file stream.

To illustrate this, let us examine the simple Tapir program from Fig. 2. It depicts a simple file I/O class. Here, for example, `join_point<test_open()>` can be replaced by a corresponding call to method `test_open` from one or more test scenarios.

From the two test scenarios from Fig. 3, four new tapir programs (four permutations) are created by the weaver. One that looks only at scenario S1 which causes the join-point statements to be replaced by calls to S1's `test_open, test_read,` etc. The second generated Tapir program is similar and considers only scenario S2. The other two permutations replace each join-point with two calls, one per scenario. For example, `join_point<test_open()>` will be replaced by {`S1's test_open(); S2's test_open(); `} in the third permutation. In the last permutation, scenario S2 is considered first followed by scenario S1. This causes a replacement snippet:
{`S2's test_open(); S1's test_open(); `}.

Whenever a join-point causes a call to a scenario's method, that scenario's method is also added to the class being woven into. This allows a weaved-in method to use the members of the class where the join-point was performed. For example, this allows the use of

```
scenario S1 {
  void test_open(int ret, string name) {..}
  void test_read(byte ret)  {assert(is_open);..}
  void test_close(int fd) {..}

  class FS_Mixin {
    void main() {
      int fd = open("S1");
      byte val = read_byte(fd);
      assert(val == 42);
      close(fd);
    }
  }
}

scenario S2 {
  void test_open(int ret, string name) {..}
  void test_read(byte ret) {..}
  void test_close(int fd) {..}

  class FS_Mixin {
    void main() {..}
  }
}
```

Figure 3: Two test scenarios for Fig. 2.

`is_open` in the assert statement in the `test_read` method of scenario S1.

The clause `join_point<FS_Mixin>` in the first line of Fig. 3 causes the methods and fields of class `FS_Mixin` from a test scenario to be added to the class FileStream. For the first two permutations, the `main` method of either S1 or S2 is copied in. For the other two permutations, we cannot bring in two equally typed `main` methods of course. Instead, we concatenate the two bodies to create a single new `main` method.

As not all permutations of scenario weavings are valid, the programmer can restrict the set of generated permutations by means of additional syntax:

- `excludes X` says that this scenario excludes scenario 'X'.
- `repeat Y` says that this scenario can be repeated up to Y times inside a permutation. By default a scenario is allowed to occur only once inside a permutation.
- `exclusive` says that this scenario cannot be combined with any other scenario.
- `after X` says that this scenario can only be put into a permutation if scenario X has already been added.

4 Application specific code transformations to reduce the state space

To support threaded execution in the model-checker, Tapir supports a `context_switch` instruction. Execution of this instruction allows the model-checker to non-deterministically execute another (or the same) thread or process. This instruction spans the state space in the model-checker to create all possible interleavings of thread and process executions. Removing even one single context-switch thus has the potential to greatly reduce the state space.

While some context-switch instructions can potentially be removed by generic (optimization) rules, many can only be removed if application specific knowledge is applied. For example, many context-switches can be removed if the programmer would provide the compiler with knowledge that a piece of code only accesses thread-private data (and thus cannot be influenced by other threads). It is hard (and most times impossible) for compiler analysis alone to determine that data is actually thread-private.

This is why Tapir supports programmer supplied program transformation rules. These allow the programmer to code such programmer level information that can't be extracted easily (or at all) by static compiler analysis alone. These transformation rules and propagation rules are coded by the programmer in his source code alongside his Tapir classes so both are inside the same file(s). The rules themselves are interpreted by the compiler so no compiler internals are exposed to the application programmer.

A program transformation rule needs to express two things: (1) when to allow the transform and (2) what to transform into what. The 'when' question is almost always non-local: information from somewhere allows a transformation elsewhere. To allow programmer supplied transformations, we must therefore (application specifically) propagate information using (predicate) propagation rules and provide a mechanism for describing program transformations. Both mechanisms are supplied in the language (as opposed to hard-coded inside the compiler).

The implications of bad (and good) applications of block-boxing are discussed in Sec. 5.2.

4.1 Transformation rules

```
int open(string name) {
  l.lock();
  ..
  work_open();
  ..
  is_open = true;
  l.unlock();
  return ret;
}
private void work_open() {
  ..
  context_switch;
  ..
}
```

Figure 4: A propagation rule example.

Consider the `open` method of the FileSystem example of Fig. 4. Because `open` might be concurrently invoked, is its state is protected by lock `l`. For some reason there is a `context_switch` statement in `work_open` which is called from `open`. However, because there is a lock/unlock pair surrounding it, the context-switch statement might be superfluous when no other thread can interfere. This knowledge is application specific as, for example, locks are programmed using ordinary Tapir statements and so provide no language level 'hints' to atomicity. To be more precise, the lock class is a meta class (from Sec. 5).

To get rid of the context-switch in this example, the program can provide a transformation rule like:

transformation_rule {
    guard: LockedToken[x]
    pattern: { context_switch; }
    target: { }
}

The semantics is that when the pattern's statement (the context-switch statement here) is found in the code while the guard holds, the pattern's statement is replaced by the target's (empty) statement. The guard is a compile-time predicate. How that works is described below.

4.2 Compile-time predicates

With the `predicate` statement, the programmer can tell the compiler that something holds for a program variable. Compile-time predicates have no code generated for them at all. For example, the `Lock.lock()` method can have:

predicate `LockedToken[this]`

at its end. The compiler propagates this predicate through the code and can then prove that `LockedToken[l]` holds in `FileSystem.open`. The propagation itself is facilitated using programmer supplied propagation rules, see below in Sec. 4.3.

Note that transformation rules and predicates encode *application level* knowledge. The compiler does not need to know that the `Lock.lock()` and `Lock.unlock()` methods are special in any way.

In addition to using these compile-time predicates as guards in program transformations, they can also be used in `must_hold` annotations on fields. For example:

class FileStream **join_point** <FS_Mixin> {
    boolean `is_open` **must_hold:** `LockedToken[l];`
    ...
}

The compiler will flag an error if the predicate has not been propagated to any statement that uses `is_open`. Extended syntax allows to distinguish read and write accesses. The `must_hold` annotations over class fields are similar to type-states, see [DF01, NGC05].

4.3 Propagation rules

Consider the following propagation rule that states that if the predicate `LockedToken[x]` holds for variable `x` on a control-flow path that leads to the assignment `'y = x'`, then

the predicate `LockedToken[y]` holds afterwards:

propagation_rule {
  if_holds_before LockedToken[x]
  with y = x;
  then_holds_after LockedToken[y]
}

The statement for a propagation rule (here `'y = x'`) can include field accesses and calls to allow heap and inter-procedural predicate propagation. In case of multiple control-flow-path predecessors, "`if_holds_before`<0> P[x] `and if_holds_before`<1> P[x]" is used, where '0' and '1' denote predecessor paths.

Note that transformation guards *must* be compile-time entities and not run-time entities as they are used to enable compile-time transformations. The compiler must *prove* that a guard always holds. We do this by propagating information.

Instead of compile-time predicates, run-time predicates could be used also. However, due to the time-intensive nature of model-checking, making the model-checked code as small and fast as possible is imperative. Adding run-time tests is therefore counter productive.

5 Application specific black boxing to reduce the state space

```
meta class StorageDevice {
  Block[] blocks;

  Block get_block(int id) {
    return blocks[id]
  }

  void put_block(int id, Block b) {
    blocks[id] = b;
  }
}
```

Figure 5: Meta class.

Abstract-data types hide implementation from specification. We re-introduce the concept here by allowing the programmer to provide two implementations of a class: a simplified implementation used during model-checking and a complex version used during code generation.

The specification of a type can be modified in three ways: by marking a field, a method, or a whole class with the *meta* keyword. In the FileStream class of Fig. 2, a StorageDevice class is used to perform actual device I/O. The full code of this class is rather lengthy and its implementation is transparent to the FileStream class under test. It would be both impractical and computationally intensive to use it in model-checking since the state space would explode. To alleviate this problem, a simplified version that (loosely) implements the same interface contract can be provided, see Fig. 5. In the example, when Tapir's compiler generates the library, calls to `get_block` and `put_block` are generated (whose

implementation is to be externally supplied). When Tapir generates the model-checkers it uses the code of the meta class.

To summarize, when marking a field as *meta*, the field itself is used during model-checking, but in the generated implementation code, calls are emitted to getter and setter methods. When marking a method *meta*, the method's code is used during model-checking but when generating the implementation code, the code of the function needs to be supplied externally. Marking a class as *meta* causes all its methods and fields to be marked *meta*.

5.1 Locks, condition variables, and compile-time asserts

Note that in Tapir, lock objects and condition variable objects, are themselves regular Tapir objects. To ensure atomic execution of statements, Tapir provides the `atomic` statement-block which is only allowed to occur in meta statements as it only has meaning for the model-checker:

```
atomic { statements }
```

The actual implementation of the class must ensure its own atomic execution if needed. Atomic blocks are guaranteed to be atomic inside the model-checker by not allowing either the programmer or the compiler to insert context-switch instructions in them.

To allow waiting for a specific condition to become true, Tapir supplies the `wait_until` statement:

```
wait_until <condition> statement
```

which blocks the current Tapir thread until the condition holds. Once the condition holds, the statement is executed. Each `wait_until` implicitly adds a context-switch statement so that if the condition does not hold, another thread can run. We can use atomic blocks and the `wait_until` statements to implement the lock object as follows:

```
class Lock {
  ...
  meta private void blocking_lock()
  {
    predicate(UnlockedToken[this]);
    Thread t = Thread.current();

    wait_until value == 0 atomic {
      owner = t;
      value = 1;
    }

    predicate(UnlockedToken[this] => LockedToken[this]);
  }

  meta void unlock()
  {
    predicate(LockedToken[this]);
    atomic {
      owner = null;
      value = 0;
    }
    predicate(LockedToken[this] => UnlockedToken[this]);
  }
}
```

Let us look at the `blocking_lock` method. First, we assert that the compile-time predicate LockedToken holds over 'this'. Afterwards we store the reference to the current

thread for later use. As soon as the value becomes zero, the current thread becomes the lock owner and the value is set to one to indicate that the object is currently locked.

If `UnlockedToken[this]` holds, the transition predicate changes the predicate to `UnlockedToken[this]`. We need an explicit transition statement here as simply asserting `predicate(LockedToken[this])` at the end of `blocking_lock` would cause to have both `UnlockedToken[this]` and `LockedToken[this]` valid on the same reference (`this`) at the end of the method. There would then be no way to disambiguate between a correct predicate use here and a detected problematic usage.

The unlock method expects the LockedToken property to be set on it and so has the predicate asserted. It then atomically resets the owner and the value before compile-time predicate translation.

The code of the Condition variable is similar. A `wait_until` statement blocks on a thread's `signal` field. A list of blocked threads is used for waking them up once signalled. Condition.Signal, takes one thread from the list, and sets it `signal` field.

Because locks and condition variables use `wait_until`, the more they are used, the larger gets the search space for the model-checker to explore.

Deadlock detection is implemented by the Lock and Condition objects themselves as they count the number of blocked threads and compare it against the number of running threads by an assert statement. Live-lock detection is currently not implemented as it is harder. For example, it could happen by message exchanges, back-off algorithms, etc.

5.2 Discussion

While black-boxing helps to reduce the state-space, it also allows for divergence of the model-checked system and the final system. If the black-box exactly implements the semantics of the actual implementation, then no problems will occur. If the black-box provides semantic changes in allowing less than the actual implementation, the system could show errors not present when the black-box (meta) types are replaced by their actual implementations. If the black-box allows more in some way (in inputs, outputs, etc.) compared to the actual implementations they replace, the final system will not show the errors during model-checking.

It is the responsibility of the programmer to provide accurate replacement meta types. This problem is endemic, however, to all systems doing black-box replacement. At least in the solution proposed here, the places where abstraction from 'reality' occurs are clearly marked. Also, this guarantees that the *structure* of the whole program is the same as in the actual implementation while this cannot be guaranteed if the specification or model are completely separate from the implementation.

Table 1: File System Benchmark

	Average Time	Heap Size
No Optimizations	342 s	11 GByte
Propagation	163 s	7 GByte
Propagation + Meta FileSystem	91 s	4 GByte

**Dual Opteron 246, 2 GHz, 12 GByte of memory.*

6 Performance

To measure model-checking performance, we use one application generated by applying one test-scenario to our file system example. We first measure the effectiveness of the propagation rules and user-level optimization rules and next the effectiveness of using meta classes to remove code complexity.

The code of the benchmark is too big to include verbatim, so we sketch it here. The whole program starts by creating two process objects. Each process then creates a thread to run `main()`. Each `main()` then creates a `FS_tester` thread to run the FileSystem tests and waits for completion (using a condition variable object). Each `FS_tester` thread accesses the file system object to perform a series of writes and reads and uses (runtime) asserts to check self-consistency. Because there could be multiple `FS_tester` threads, the file system object is protected by a lock object. Outside of these threads, the Tapir runtime library creates two additional Tapir threads per Tapir process for providing services (a thread for managing incoming messages from other Tapir processes and another for waiting for global system termination). In total, there are thus two processes to model-check, each with four threads running and interacting and a number of objects allocated by them.

We then create two versions of the application. One version (A) uses the file system class described above and a companion Device meta class. A second version (B) of the application replaces the FileSystem class with a meta class (making the Device meta class redundant). Version (B)'s FileSystem methods have just enough implementation to satisfy their interfaces. Furthermore, the benchmarks include Tapir's runtime so that measurements include the time needed to non-deterministically start-up the service threads and model their interactions (which already contain a number of locks, etc. internally).

We then run the model-checker twice on version (A): once with and once without propagation rules. For version (B) we time the model-checker with propagation rules enabled.

Table 1 gives performance results. Without optimizations (version (A), no propagation), memory usage and run time are high. With optimization (version A with propagation), two context switch instructions are removed (over a total of 12). However, one of those is in a loop and hence critical. This causes the big performance gains seen. Version (B) makes the code simpler (fewer instructions to simulate) and removes some object allocations (smaller process state).

Memory requirements are high because every new state discovered in the search space is placed into a hash-table. We use a 64 bit hash key computed over the simulated processes.

The hash table ensures that no state that has already been visited is expanded again. Memory usage is kept low by maintaining diffs of the simulated processes's memory whenever possible instead of creating a full-blown copy of the process each time it is stored into the model-checker's hash table.

7 Related Work

Except for the language restrictions, the programmer is free to code anything in Tapir. This includes recursion which is not allowed by some other software model-checkers (SPIN/Promela [Hol97], NuSMV [CCG+02]). Tapir also supports concurrency (processes and threads) which other software checkers cannot handle, e.g. Moped [mop03]. In general, model-checkers that use state-machines (like Petri-networks [CCM97]), cannot handle recursion but can handle concurrency. Stack-machine based model-checkers cannot handle recursion but can handle concurrency efficiently. Approaches that use virtual (Von-Neumann) machines can handle both (such as we do here). They can handle both but require more details in their models to operate and so generate a larger search space compared to the restricted techniques.

Most model-checking approaches use separate implementation and model-checking languages. This can cause both implementations to diverge such that a bug-free, model-checked application can still show logic bugs in the implementation, see [PFHV07]. Tapir uses one single language with only abstracting away low-level code in meta classes. This approach should scale better than model-checking the complete application directly such as CMC [MPC+02] does.

The use of Aspect-orientation has been proposed before for various tasks from implementing the parallelization itself [Sob06] to adding unit-tests [Ben08]. But to our knowledge, AOP has not been used in a tool chain to generate permutations of test programs.

Condate[Vol06], extends GCC with external rules for semantic analysis. For example, one might write a rule that states that between a call to *malloc()* and a call to *free()* there should be a test against NULL. We achieve the same effect by a combination of predicates, propagation rules, and transformation rules at language level but for optimization purposes. Cobalt [LMC03] is similar to Condate, but uses its rules to generate a static optimizer in a compiler while in Tapir rules are taken from the program and interpreted in the compiler for extra flexibility.

Unlike our language-level approach, some approaches use compiler based static state space reduction, e.g. [YG04] which does not allow application-level knowledge to reduce the search space. Tools such as F-Soft [GGI+08], Bandera [CDH+00] and Java-PathFinder [LV01] take a program and extract a specification for an existing program verifier by removing program details automatically. In contrast, Tapir leaves it to the programmer to decide which program details are superfluous or essential. Other techniques to reduce the search space, such as program slicing [HRB88], and predicate reduction or predicate abstraction [DDP99] are orthogonal to Tapir's approach and could be added to our tool chain.

8 conclusion

Using AOP to generate test programs for libraries is a useful first step towards automatic library testing. What is further required are things such as coverage analysis (how much of the library is covered by a non-deterministic run of a library test), input value boundary tests, etc. The use of meta classes to separate code into model-checking specific and implementation specific parts is key to allow model-checking for some types of software, especially if it interacts with hardware. Even if such problems are absent, black boxing code inside simplified meta classes can severely increase model-checking performance (at the cost of false negatives/positives).

Even though programmer written optimization rules can be hard to write and get right, they can help performance enormously. Still, by providing them in libraries the programmer may even never see their use.

References

[Ben08] Sebastian Benz. AspectT: aspect-oriented test case instantiation. In *AOSD '08: Proc. Intl. Conf. on Aspect-Oriented Software Development*, pages 1–12, Brussels, Belgium, 2008.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV 2002: Proc. Intl. Conf. on Computer-Aided Verification*, volume 2404, Copenhagen, Denmark, 2002.

[CCM97] Allan Cheng, Sren Christensen, and Kjeld H. Mortensen. Model Checking Coloured Petri Nets Exploiting Strongly Connected Components. In *WODES96: Proc. of the Intl. Workshop on Discrete Event Systems*, pages 169–177, 1997.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE '00: Proc. Intl. Conf. on Software Engineering*, pages 439–448, Limerick, Ireland, 2000.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proc. of the 21st Intl. Conf. on Software Engineering*, pages 411–420, New York, NY, USA, 1999. ACM.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with Predicate Abstraction. In *CAV '99: Proc. of the 11th Intl. Conf. on Computer Aided Verification*, pages 160–171, 1999.

[DF01] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *PLDI '01: Proc. Conf. on Programming Language Design and Implementation*, pages 59–69, Snowbird, UT, USA, 2001.

[GGI+08] Malay K. Ganai, Aarti Gupta, Franjo Ivani, Vineet Kahlon, Weihong Li, Nadia Papakonstantinou, Sriram Sankaranarayanan, and Chao Wang. Towards Precise and Scalable Verification of Embedded Software. In *DVCon: Proc. of 2008 Design and Verification Conference*, San Jose, CA, 2008.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proc. Conf. on Programming Language Design and Implementation*, pages 35–46, Atlanta, Georgia, 1988.

[LMC03] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI '03: Proc. Conf. on Programming Language Design and Implementation*, pages 220–231, San Diego, CA, 2003.

[LV01] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *SPIN '01: Proc. Intl. SPIN workshop on Model checking of software*, pages 80–102, Toronto, Ontario, Canada, 2001.

[mop03] MOPED: A Multi-objective Parzen-Based Estimation of Distribution Algorithm for Continuous Problems. In *EMO 2003: Proc. Intl. Conf. on Evolutionary Multi-Criterion Optimization*, pages 71–81, Faro, Portugal, Apr. 2003.

[MPC+02] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[NGC05] Mangala Gowri Nanda, Christian Grothoff, and Satish Chandra. Deriving object typestates in the presence of inter-object references. In *OOPSLA '05: Proc. Conf. on Object Oriented Programming, Systems, Languages, and Applications*, pages 77–96, San Diego, CA, 2005.

[PFHV07] Jun Pang, Wan Fokkink, Rutger Hofman, and Ronald Veldema. Model checking a cache coherence protocol of a Java DSM implementation. *Journal of Logic and Algebraic Programming*, 71(1):1–43, 2007.

[Sob06] J. Sobral. Incrementally Developing Parallel Applications with AspectJ. In *Proc. Intl. Parallel and Distributed Processing Symp.*, pages 1–10, Rhodes, Greece, April 2006.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA '92: Proc. Intl. Symp. on Computer Architecture*, pages 256–266, Queensland, Australia, 1992.

[Vol06] Nic Volanschi. Condate: a proto-language at the confluence between checking and compiling. In *PPDP '06: Proc. of the Intl. Conf. on Principles and Practice of Declarative Programming*, pages 225–236, Venice, Italy, 2006.

[VP07] Ronald Veldema and Michael Philippsen. Supporting Huge Address Spaces in a Virtual Machine for Java on a Cluster. In *LCPC: Languages and Compilers for Parallel Computing*, Urbana, IL, Oct. 2007.

[YG04] Karen Yorav and Orna Grumberg. Static Analysis for State-Space Reductions Preserving Temporal Logics. *Formal Methods in System Design*, 25(1):67–96, 2004.