Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software

Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar

FIT, Brno University of Technology, Czech Republic
{krena, iletko, vojnar}@fit.vutbr.cz

Abstract. Coverage metrics play a crucial role in testing. They allow one to estimate how well a program has been tested and/or to control the testing process. Several concurrency-related coverage metrics have been proposed, but most of them do not reflect concurrent behaviour accurately enough. In this paper, we propose several new metrics that are suitable primarily for saturation-based or search-based testing of concurrent software. Their distinguishing feature is that they are derived from various dynamic analyses designed for detecting synchronisation errors in concurrent software. In fact, the way these metrics are obtained is generic, and further metrics can be obtained in a similar way from other analyses. The underlying motivation is that, within such analyses, behavioural aspects crucial for occurrence of various bugs are identified, and hence it makes sense to track how well the occurrence of such phenomena is covered by testing. Next, coverage tasks of the proposed as well as some existing metrics are combined with an abstract identification of the threads participating in generation of the phenomena captured in the concerned tasks. This way, further, more precise metrics are obtained. Finally, an empirical evaluation of the proposed metrics, which confirms that several of them are indeed more suitable for saturation-based and search-based testing than the previously known metrics, is presented.

1 Introduction

Despite the constant development of various approaches to verification and bug finding based on formal roots, *software testing* still belongs among the most common ways of discovering errors in programs. However, it has to face new challenges related to the changes in programming paradigms commonly used in practice. In particular, in the past years, *concurrent programming* has become much more common than before. Testing concurrent software is much more difficult due to the non-determinism present in scheduling executions of concurrent threads. Various ways how to improve testing of concurrent software have been proposed, including, e.g., the use of noise injection or various dynamic analyses.

In testing, a crucial role is played by the so-called *coverage metrics*. A coverage metric is based on a *coverage domain* that is a set of *coverage tasks* representing different phenomena (such as reachability of a certain line, reachability of a situation in which a certain variable has a certain value, etc.) whose occurrence in the behaviour of a tested program is considered to be of interest. One can then measure how many of the phenomena corresponding to the coverage tasks have been seen in the witnessed behaviours of the tested program. Such a measurement can be used to asses how well the program has been tested. Moreover, in the so-called *saturation-based testing* [16], one looks for the moment when the obtained coverage stops growing, and hence the testing can be stopped. Further, in *search-based testing* [12], a fitness function driving an optimisation algorithm used to control the testing process can be based on the values of a coverage metric.

For metrics used in saturation-based or search-based testing, one can identify several specific properties that they should exhibit. First, within the testing process, the obtained coverage should as often as possible grow for a while and then stabilise. Hence, it should not immediately jump to some value and stabilise on it. On the other hand, it should not take too much time for the coverage to stabilise. Also, to enable a reliable detection of stabilisation, the coverage should grow as smoothly as possible, i.e., without growing through a series of distinctive shoulders. Next, in case of testing an erroneous program, the stabilisation should ideally not happen before an error is detected. Finally, the increase in coverage should be linked with witnessing more and more behaviours that differ in their potential of exhibiting a bug.

In this paper, we propose several *new coverage metrics* suitable for saturation-based or search-based testing of concurrent programs. These metrics are based on coverage tasks derived from the information about program behaviour that is gathered or computed by various *dynamic analyses* that have been proposed for *discovering synchronisation-related errors in concurrent programs*. In fact, the idea of inferring new metrics from these analyses is rather generic and can be applied to other dynamic as well as static analyses (even those that will appear in the future) too. The proposal is motivated by the idea that within the development of such analyses, behavioural aspects of concurrent programs that are highly relevant for the existence of synchronisation-related errors have been identified. Hence, it makes sense to measure how well the aspects of the behaviour tracked by such analyses have been covered during testing.

Further, we also combine coverage tasks of the newly proposed as well as some existing metrics with *abstract identifiers of the threads* involved in generating the phenomena reflected in the concerned tasks. The identifiers abstract away the concrete numerical identifiers of the threads, but preserve information on their type, the history of their creation, etc. This way, an increased number of coverage tasks is obtained, forming a new, more precise variant of the original metric.

We have performed an *empirical comparison* of the use of the newly proposed metrics against three common concurrency-related metrics. We show that several of the newly proposed metrics indeed meet the criteria of suitability for saturation-based and search-based testing in a significantly better way than the previously known metrics.

Plan of the paper. In Section 2, we discuss the related work. Section 3 details the proposed way of deriving new coverage metrics and presents several concrete new metrics. For comparison purposes, the section then also presents in a uniform way several existing metrics (one of these metrics is slightly extended too). Section 4 describes our experimental setting and the techniques we use for an abstract identification of objects and threads. Section 5 provides our experimental results. Finally, Section 6 concludes the paper and provides some notes on the possible future work.

2 Related Work

As said already in the previous section, testing is one of the most common approaches used for discovering concurrency bugs. The testing process is typically empowered in some way to cope with the fact that concurrency bugs often appear only under very special scheduling circumstances. To increase chances of spotting a concurrency bug, various ways of *influencing the scheduling* are often used. An example of this approach is random or heuristic noise injection used in the IBM Concurrency Testing Tool (ConTest) [4] or a systematic exploration of all schedules up to some number of context switches as used in the Microsoft CHESS tool [13].

Another way to improve traditional concurrency testing is to try to extrapolate the behaviour seen within a testing run and to warn about a possible error even if such an error was not in fact seen in the test execution. Such approaches are called *dynamic analyses*. Many dynamic analyses have been proposed for detecting special classes of bugs, such as data races [2, 5, 14, 15], atomicity violations [10], or deadlocks [1, 7]. These techniques may find more bugs than classical testing, but on the other hand, their computational complexity is usually higher, and they can also produce false alarms.

An alternative to testing and dynamic analyses is the use of *static analyses*. They avoid execution of the given program or execute it on a highly abstract level only. Various static analyses of concurrent software exist, including light-weight analyses that look for specific patterns in the code that might lead to a bug [6] or, e.g., various dataflow-based analyses that try to identify bugs like data races [8] or deadlocks [20]. *Model checking* [3] (sometimes viewed as a heavy-weight static analysis too) tries to systematically analyse all possible interleavings of threads in a given program (the CHESS approach can, in fact, be seen as a form of bounded model checking). Light-weight static analyses may produce many false alarms and heavy-weight approaches may have troubles with scalability. There also exist approaches that combine static and dynamic analyses in an attempt to suppress their deficiencies.

We build our new coverage metrics on the information that is gathered or computed by several different dynamic analyses mentioned above, namely, Eraser [15], GoldiLocks [5], AVIO [10], and GoodLock [1]. In our experiments with these metrics, we use ConTest and its noise injection mechanisms to generate different legal interleaving scenarios in repeated executions of the considered test cases. Although not explored in this paper, new coverage metrics could be derived from various static analyses too.

Many different coverage metrics have been proposed targeting probably all areas of testing in the past decades. Testing of concurrent software is not an exception. Out of the existing *concurrency-related metrics*, among the ones that we find as the probably most promising from the point of view of their practical applicability there is the metric based on du-pairs proposed in [21], the metric based on concurrent pairs of events from [2], and the synchronisation coverage [18]. We discuss these metrics in more detail in Section 3.3, and we experimentally compare our metrics with them in Section 5.

The idea of extending coverage tasks of metrics by further information has also been presented in [16] where saturation-based testing of concurrent programs is introduced. The authors propose three types of context information which can be used to refine existing metrics. The *pair context* handles situations where two events in the concurrent programs interact and makes this information explicit for the metric. The *group context* makes explicit the type of thread that performed an event (this is a special case of our abstract thread identifiers). Finally, the *thread context* explicitly identifies the thread which performed the event.

3 Concurrency Coverage Metrics

Our goal is to create metrics that are suitable for saturation-based and search-based testing of concurrent software. As we have already said in the introduction, metrics to be used in this context should have some special properties. In particular, during testing, the coverage should as often as possible first increase for some reasonable amount of time and then stabilise. The stabilisation should not happen too early nor too late. This typically implies that the number of coverage tasks should not be too small nor too large. The growth should not generate distinctive shoulders so that saturation can be reasonably detected. In case of testing an erroneous program, the stabilisation should as often as possible happen after the error is detected. Finally, a growth of coverage should be in some relation to witnessing more and more behaviours distinct from the point of view of their potential for generating some concurrency error. In addition, one should also consider a generic requirement for the metrics not to be too costly to use

We now first discuss a methodology how metrics satisfying the above can be obtained, and then propose several new concrete metrics. Finally, for comparison purposes, we describe (and in one case also extend) some existing metrics too.

3.1 Methodology of Deriving Suitable Coverage Metrics

To derive metrics satisfying the criteria set up above, we propose to get inspired by various existing *dynamic (and possibly even static) concurrency error detection techniques*. This is motivated by two observations: (1) These detection techniques focus on those events occurring in runs of the analysed programs that appear relevant for detection of various concurrency-related errors. (2) The techniques build and maintain a representation of the context of such events that is important for detection of possible bugs in the program. Hence, trying to measure how many of such events have been seen, and possibly in how many different contexts, seems promising from the point of view of relating the growth of a metric to an increasing likelihood of spotting an error.

The described idea is very generic, and we can speak about a new class of concurrency coverage metrics that can be obtained in the described manner. A crucial step in the creation of a new coverage metric based on some error detection algorithm is to choose suitable pieces of information available to or computed by the detection algorithm, which are then used to construct the domain of the new coverage metric such that the other, above mentioned criteria are met. This leads to a trade off among the precision of the metric and the amount of information tracked, the associated computational complexity, and speed of saturation. One extreme is to build a coverage metric directly on warnings about concurrency errors issued by the detection algorithm. In this case, we need to implement the detection algorithm entirely. Another extreme is to build a coverage metric counting just the events tracked by the detection algorithm, without their context. In such a case, we often obtain very similar metrics to already existing metrics. Within this process—which can hardly be made algorithmic and which requires certain ingenuity and also experimental evidence, it can also of course turn out that some detection algorithms are not suitable as a basis of a coverage metric at all.

Let us demonstrate the described problem on an example of two dynamic data race detection algorithms. The *vector-clock-based algorithms*, e.g., [14], maintain for each thread an internal clock which is an integer value representing the number of synchronisation events that the thread executed so far. The algorithm then also maintains for each thread, each lock, and each variable vectors of clocks representing synchronisation bindings among events performed on these program elements. The goal is to obtain the so-called *happens-before relation* that says which events are *guaranteed* to happen before other events, meaning that such events cannot participate on a data race (where the order of the events must not be fixed). Nevertheless, vectors of clocks are not suitable for our purposes because they encode the history context using a too large number of values. This would lead to a huge number of coverage tasks, a slow progress towards saturation, and also a high cost of measuring the obtained coverage.

On the other hand, the Eraser algorithm [15] computes the so-called *locksets*. For each thread, the algorithm computes a set of locks currently held by the thread, and for each variable access, the algorithm uses these sets to derive the set of locks that were held by each thread that had so far accessed the variable. These so-called locksets are maintained according to a *state* assigned to each variable which represents how the variable has been operated so far (e.g., exclusively within one thread, shared among threads, for reading only, etc.). This algorithm is more suitable for our purposes because the history context used by it gives rise to a reasonable number of coverage tasks (as we show below).

Finally, we note that, according to our experimental evidence mentioned later on, the precision of the constructed metrics can further be suitably adjusted by combining their coverage tasks with some *abstract identification of the threads* involved in generating the phenomena reflected in the concerned tasks. The identification should of course not be based on the unique thread identifiers, but it can preserve information on their type, the history of their creation, etc. A similar identification can then also be used whenever the coverage tasks contain some dynamically instantiated objects (e.g., locks).

3.2 New Coverage Metrics

We are now going to derive several new concrete coverage metrics. As we have already said, they are all based on some dynamic analyses used for detecting errors in synchronisation of concurrent programs. In order to allow for a quick comparison among the metrics, Table 1 presents an overview of all the proposed metrics, together with some other metrics that we will consider in our experiments. For each metric, the second column shows a tuple defining coverage tasks of the metric, and the third column contains information whether the metric is new (N), already existing (E), or whether it is our modification of some already known metric (M). The first item of each of the tuples representing a coverage task (denoted as pl_1) gives a primary program location which generates the given task when reached by some thread. The rest of the tuples can then be viewed as a context under which the location is reached. For most of the metrics, we provide two versions: a basic version

Table 1. The considered coverage metrics

metric	coverage task	note
Avio	(pl_1, pl_2, pl_3)	N
Avio*	$(pl_1, pl_2, pl_3, var, t_1, t_2)$	N
Eraser	$(pl_1, state, lockset)$	N
Eraser*	$(pl_1, var, state, lockset, t_1)$	N
GoldiLock	$(pl_1, goldiLockSetSC)$	N
GoldiLock*	$(pl_1, var, goldiLockSetSC, t_1)$	N
GoodLock	(pl_1, pl_2, l_1, l_2)	N
GoodLock*	$(pl_1, pl_2, l_1, l_2, t_1)$	N
HBPair	$(pl_1, pl_2, syncObj)$	N
HBPair*	$(pl_1, pl_2, syncObj, t_1, t_2)$	N
ConcurPairs	$(pl_1, pl_2, switch)$	E
DUPairs	(pl_1, pl_2, var)	E
DUPairs*	$(pl_1, pl_2, var, t_1, t_2)$	M
Sync	$(pl_1, mode)$	E

and a version with an extended context, denoted by *. In the following paragraphs, the versions with the extended context are described only. The basic versions can easily be derived from them by dropping some elements of the context.

In order to make the description more concrete, in the rest of the paper, we assume the *Java memory model* [11]. In the text below, we use the following notation. V is a set of identifiers of instances of non-volatile variables (i.e., non-volatile fields of objects) that may be used in the tested program at hand, O is a set of identifiers of instances of volatile variables used in the program, L is a set of identifiers of locks used in the program, T is a set of identifiers of all threads that may be created by the program, and P is a set of all program locations in the program. We discuss one possible concrete way how the needed identifiers may be obtained in Section 4.1.

A coverage metric based on Eraser. The coverage metric Eraser* is based on the Eraser algorithm [15] whose basics have been sketched above. Its coverage tasks have the form of a tuple $(pl_1, var, state, lockset, t_1)$ where $pl_1 \in P$ identifies the program location of an instruction accessing a shared variable $var \in V$, $state \in \{virgin, exclusive, exclusive', shared, modified, race\}$ gives the state in which the Eraser's finite control automaton is when the given location is reached (we consider the extended version of Eraser using the $exclusive'$ state as introduced in [19], which is more suitable for the Java memory model), and $lockset \subseteq L$ denotes a set of locks currently guarding the variable var. Finally, $t_1 \in T$ represents the thread performing the access operation.

A coverage metric based on GoldiLocks. GoldiLocks [5] is one of the most advanced lockset-based algorithms. The main idea of this algorithm is that it combines the use of locksets with computing the happens-before relation. In GoldiLocks, locksets are allowed to contain not only locks but also volatile variables and threads. If a thread t appears in the lockset of a shared variable when the variable is accessed, it means that t is properly synchronised for using the given variable because all other accesses that might cause a data race are guaranteed to happen before the current access. The algorithm uses a limited number of elements placed in the lockset to represent an important part of the synchronisation history preceding an access to a shared variable. This is in contrast with the vector-clocks-based algorithms mentioned above. The basic GoldiLocks algorithm is still relatively expensive but can be optimised by the so-called *short circuit checks* (SC) which are three cheap checks that are sufficient for deciding race freedom between the two last accesses to a variable. The original algorithm is then used only when SC cannot prove race freedom. Our GoldiLock-based metric GoldiLock* is based on coverage tasks having the form of tuples $(pl_1, var, goldiLockSet, t_1)$ where $pl_1 \in P$ gives the location of an instruction accessing a variable $var \in V$ within a thread $t_1 \in T$, and $goldiLockSet \subseteq O \cup L \cup T$ represents the lockset computed by GoldiLocks.

A coverage metric based on Avio. The Avio algorithm that detects atomicity violation over one variable is presented in [10]. We choose this algorithm because it does not require any additional information from the user about instructions that should be executed atomically. The algorithm considers any two consecutive accesses a_1 and a_2 from one thread to a shared variable var to form an atomic block B. Serialisability is then defined based on an analysis of what can happen when B is interleaved with some read or write access a_3 from another thread to the variable var. Out of the eight total cases arising in this way, four (namely, r/w/r, w/w/r, w/r/w, r/w/w) are considered to lead to an unserialisable execution. Tracking of all accesses that occur concurrently to a block B can be very expensive. Therefore, we define our criterion to consider only the last interleaving access to the concerned variable from a different thread. Our Avio* metric uses coverage tasks in the form of tuples $(pl_1, pl_2, pl_3, var, t_1, t_2)$ where $var \in V$, $pl_1, pl_2, pl_3 \in P$, and $t_1, t_2 \in T$. The considered atomic block B spans between pl_1 and pl_2, and it is executed by a thread t_1. Finally, pl_3 gives a location of an instruction executed in a thread t_2 that interferes with the block B.

A coverage metric based on GoodLock. GoodLock is a popular deadlock detection algorithm that exists in several modifications—we, in particular, build on its modification published in [1]. The algorithm builds the so-called *guarded lock graph* which is a labelled oriented graph where nodes represent locks, and edges represent nested locking within which a thread that already has some lock asks for another one. Labels over edges provide additional information about the thread that creates the edge. The algorithm searches for cycles in the graph wrt. the edge labels in order to detect deadlocks. Our metric focuses on occurrence of nested locking that is considered interesting by GoodLock. We omit collection of the locksets of the threads which the original algorithm uses as one element of the edge label because this information is used in the algorithm to suppress certain false alarms only. Our GoodLock* metric is therefore based on coverage tasks in the form of tuples $(pl_1, pl_2, l_1, l_2, t_1)$ where $pl_1, pl_2 \in P$, $l_1, l_2 \in L$, and $t_1 \in T$. Such a task is covered when the thread t_1 has obtained the lock l_1 at pl_1, and now the same thread is obtaining the lock l_2 at pl_2.

A coverage metric based on happens-before pairs. This coverage metric is motivated by observations we get from the GoldiLocks algorithm and the vector-clock algorithms, both of them depending on computation of the happens-before relation. In order to get rid of the possibly huge number of coverage tasks produced by the vector-clock algorithms and trying to decrease the computational complexity needed when the full GoldiLocks algorithm is used, we focus on pieces of information the algorithms use for creating their representations of the analysed program behaviours (without actually computing and using these representations). All of these algorithms rely on synchronisation events observed along the execution path. Inspired by this, we propose the HBPair* metric that tracks successful synchronisation events based on locks, volatile variables, wait-notify operations, and thread start and join operations used in Java. A coverage task is defined as a tuple $(pl_1, pl_2, syncObj, t_1, t_2)$ where $pl_1 \in P$ is a program location in a thread $t_1 \in T$ that was synchronised with the location $pl_2 \in P$ of the thread $t_2 \in T$ using the synchronisation objects $syncObj \in L \cup O \cup \{\bot\}$. Here, $\bot$ represents a thread start or a successful join synchronisation where no synchronisation object is needed.

3.3 Existing Metrics

In order to compare our metrics with already existing metrics, we further consider—and in one case also extend—the following metrics.

Coverage based on concurrently executing instructions (ConcurPairs). The coverage of concurrent pairs of events proposed in [2] is a metric in which each coverage task is composed of a pair of program locations that are assumed to be encountered consecutively in a run and a third item that is $true$ or $false$. It is $false$ iff the two locations are visited by the same thread and $true$ otherwise—that is, $true$ means that there occurred a context switch between the two program locations. This metric provides statement coverage information (using the $false$ flag) and interleaving information (using the $true$ flag) at once. In our notation, each task of the metric is a tuple $(pl_1, pl_2, switch)$ where $pl_1, pl_2 \in P$ represent the consecutive program locations (only concurrency primitives and variable accesses are monitored), and $switch \in \{true, false\}$ denotes whether the context switch occurs in between of them. Since this metric produces a large number of coverage tasks even for small programs, we decided not to enrich it with any further context information.

Definition-use coverage. This coverage metric is based on the *all-du-path* coverage metric for parallel programs described in [21]. This metric considers coverage tasks in the form of triples (var, d, u) where var is a shared variable, d is a node in the parallel program flow graph (PPFG) where the value of var is defined, and u is a node in the PPFG where the value is read. The du-pair therefore denotes an existing path in the PPFG from a node d to a node u where the value of var from d is still available, i.e., there is no node redefining the value of var on the path between d and u. We consider the original all-du-pair coverage metric (denoted as DUPairs), and we also extend it to a metric which adds more context information to the coverage tasks. Our metric DUPairs* is based on coverage tasks in the form of tuples $(pl_1, pl_2, var, t_1, t_2)$ where $pl_1, pl_2 \in P$ represent program locations where the value of the variable $var \in V$ is defined and used, respectively, $t1 \in T$ denotes the thread that performed the definition of var at pl_1, and $t_2 \in T$ denotes the thread that subsequently uses the value at pl_2.

Synchronisation coverage (Sync). The synchronisation coverage [18] focuses on the use of synchronisation primitives and does not directly consider thread interleavings. Coverage tasks of the metric are defined based on various distinctive situations that can occur when using each specific type of synchronisation primitives. For instance, in the case of a synchronised block (defined using the Java keyword `synchronised`), the obtained tasks are: *synchronisation visited*, *synchronisation blocking*, and *synchronisation blocked*. The synchronisation visited task is basically just a code coverage task. The other two are reported when there is an actual contention between synchronised blocks—when a thread t_1 reaches a synchronised block A and stops because another thread t_2 is inside a block B synchronised on the same lock. In this case, A is reported as blocked, and B as blocking (both, in addition, as visited). In our notation, the metric is defined using tuples of the form $(pl_1, mode)$ where $pl_1 \in P$ represents the program location of a synchronisation primitive, and $mode$ represents an element from the set of the distinctive situations relevant for the given type of synchronisation.

4 The Infrastructure Used for Experiments

Our architecture for collecting concurrency related coverage is built upon the IBM Java Concurrency Testing Tool (ConTest) [4]—an advanced tool for testing, debugging, and measuring test coverage for concurrent Java programs. The tool provides a facility for bytecode instrumentation and a listeners infrastructure allowing one to create *plug-ins* for collecting various pieces of information about the multi-threaded Java programs being executed as well as to easily implement various algorithms for dynamic analyses. The tool is itself able to collect structural coverage metrics (basic blocks, methods) and some concurrency-related metrics (ConcurPairs, Sync) too. ConTest further provides a noise injection facility which injects the so-called noise into the execution of a tested application and so allows us to observe different legal interleavings if the test is executed repeatedly. We use our platform called SearchBestie [9] to set up and execute tests with ConTest, and to collect, maintain, and export results produced by ConTest and its plug-ins from multiple executions of a test.

4.1 Abstract Object and Thread Identification

Our coverage metrics introduced in Section 3 are based on tasks that include identification of threads and instances of variables and locks. The Java virtual machine (JVM) generates identifiers of objects and threads dynamically. Such identifiers are, however, not suitable for our purposes: (1) In long runs, too many of them may be generated. (2) We would like to be able to match semantically equivalent tasks generated in different runs (may be not precisely, but at least with a reasonable precision), and the identifiers generated by JVM for the same threads (from the semantical point of view) in different runs will quite likely be different.

Previous works, such as [16], used Java types to identify threads. We consider this type-based identification of elements as too rough. Our goal is to create identifiers which distinguish behaviour of objects and threads within the program more accurately, but still keeping a reasonable level of abstraction so the set of such abstract identifiers remains of a moderate size.

Our abstract *object identification* (used to identify locks as well as instances of variables[1]) is based on the observation that, usually, objects created in the same place in the program are used in a similar way. For instance, there are usually many instances of the class `String` in an average Java program, but all strings that are created within invocations of the same method will probably be manipulated similarly. Therefore, we define an object identifier as a tuple $(type, loc)$ where $type$ refers to the type of the object, and loc refers to the top of the stack (excluding calls to constructors) when the object is created. The record at top of the stack contains a method, source file, and line of code.

Our abstract *thread identification* is based on an observation that the type and place of creation are not sufficient to build a thread identifier. Several threads created at the same program location (e.g., in a loop) can subsequently process different data and therefore behave differently. We need more information concerning the thread execution trace to better capture the behaviour of threads. Therefore, we use as the identifier a tuple $(type, hash)$ where $type$ denotes the type of the object implementing the thread, and $hash$ contains a hash value computed over a sequence of n first method identifiers that the thread executed after its creation (if the thread terminates sooner, then all methods it executed are taken into account). The value of n influences precision of the abstraction. Of course, when a pool of threads (a set of threads started once and used for several tasks) is used, the computation of the hash value must be restarted immediately after picking the thread up from the pool.

[1] Instances of variables are identified by an object identifier and the appropriate field of the object.

5 Experiments

We have evaluated our metrics on four small test cases (Dining philosophers, Airlines, Crawler, and FtpServer) and one big test case (TIDOrbj).

The *Dining philosophers* test case is an implementation of the well-known synchronisation problem of dining philosophers. Our implementation is taken from the distribution of the Java PathFinder model checker. The program generates a set of 6 philosophers (each represented by a thread) and the same number of shared objects representing forks. A deadlock can occur when executing the test case.

The *Airlines* test case is a simple artificial program simulating an air ticket reservation system. It generates a database of air tickets and then allows 2 resellers (each represented by a separate thread) to sell tickets to 4 sets of 10 customers (each set is represented by a separate thread). Finally, a check whether the number of customers with tickets is equal to the number of sold tickets is done. The program contains a high-level atomicity violation whose occurrence makes the final check fail.

The three other considered programs are real-life case studies. *WebCrawler* is a part of an older version of a major IBM production software. It demonstrates a tricky concurrency bug detected in this software. The crawler creates a set of threads waiting for a connection. If a connection simulated by a testing environment is established, a worker thread serves it. There is a bug in a method that is called when the crawler shuts down. The bug causes an exception sometimes leading to a deadlock. The trickiness of the bug can be seen from its very low error probability shown in Table 2.

Our second real-life case study is an early development version of an open-source *FTPServer* produced by Apache. This case study has 120 classes. The server creates a new worker thread for each new incoming connection to serve it. The version of the server we used contains several data races that can cause exceptions during the shut down process when there is still an active connection. The probability of spotting an error when noise injection is enabled is quite high in this example because there are multiple places in the test where an exception can be generated.

Our biggest test case is *TIDorbJ*—a CORBA-compliant ORB (Object Request Broker) product that is a part of the MORFEO Community Middleware Platform [17]. The instrumented part of the middleware has 1399 classes. We have used the *Echo concurrent* test which checks how the infrastructure handles multiple concurrent simple requests. The test starts an instrumented server and then 10 clients, each sending 5 requests to the server. There was originally no error in this test, and therefore we introduced one by commenting one `synchronised` statement in the part of code that is executed by the test. This way we introduced a high-level atomicity violation that leads to a null pointer exception.

We used our infrastructure introduced in Section 4 to collect relevant data from 10,000 executions of the small test cases and 4,000 executions of TIDOrbj. In order to see as many different legal interleaving scenarios as possible, we set up ConTest to randomly inject noise into the executions. We have implemented ConTest plug-ins to collect coverage information and set up SearchBestie to detect occurrences of errors (deadlocks were detected using a timeout, other errors by detection of unhandled exceptions). All further studies of the metrics were done using the collection of executions obtained this way. For instance, we often needed to evaluate the behaviour of the metrics on series of executions. To generate the needed series of executions, we used

Table 2. Test cases and abstract identifiers

		Error	ObjectAbstraction			ThreadAbstraction			
	Classes	Ratio	Real	Type	Abs	Real	Type	Abs_{10}	Abs_{20}
Dining phil.	2	0.4151	130	3	3	7	2	2	2
Airlines	8	0.0333	15210	6	6	60	3	3	4
Crawler	19	0.0006	1828	13	14	180	4	9	12
FtpServer	120	0.4032	26110	27	29	1641	5	5	6
TIDOrbJ echo	1399	0.017	180320	98	129	79	5	9	11

SearchBestie to randomly select a needed number of executions out of the recorded collection and to compute accumulated values of the chosen metrics on such series. All tests were executed on a computer with an Intel 6600 processor and 2 GB of memory, running Sun Java version 1.6 under GNU Linux.

Table 2 gives some statistics about our test cases. The second column of the table shows the number of instrumented classes for each test case. The following column shows the probability of spotting an error during a test execution when random noise injection is used (computed as the number of executions where an error occurs divided by the total number of executions). The rest of the columns provide information about the size of the case studies in terms of the numbers of threads and objects created in them. These columns also illustrate how our abstract identifiers of objects and threads work. The *Real* column contains the total number of distinct objects (or threads) we encountered in 10 performed executions of the tests. The *Type* column shows the total number of distinct object (or thread) types we have spot, and the *Abs* columns show the total number of distinct abstract objects (or threads) we distinguish using our abstract identifiers introduced in Section 4.1. For the thread abstraction, two values are given showing the influence of the length n of the considered sequence of methods called by the threads.

5.1 Results of Experiments

A typical behaviour of the considered coverage metrics can be seen in Figure 5.1. All four sub-figures show the cumulative number of coverage tasks of the metrics covered during one randomly chosen series of the Crawler test case executions (with the thread abstraction variable n set to 20).

Figure 1(a) shows the behaviour of the metrics that, according to our opinion, do not capture the concurrent behaviour accurately enough. One coverage metric for non-concurrent code measuring the number of *basic blocks* covered during tests is added to demonstrate the difference between classical and concurrency-related coverage metrics. The coverage obtained under the metric based on basic blocks is nearly constant all the time because we are repeatedly executing the same code with the same inputs. For the rest of the metrics shown in Figure 1(a), the cumulative number of tasks covered during test executions increases only within approximately the 200 first executions, and then a saturation is reached. The only metrics which slightly differ from the others in this group are Eraser and DUPairs. The Eraser metric has a similar behaviour to the $Avio$ metric (and the metrics close to it) but approximately four times higher numbers of covered tasks. This is caused by the fact that the tracked shared variables usually get to

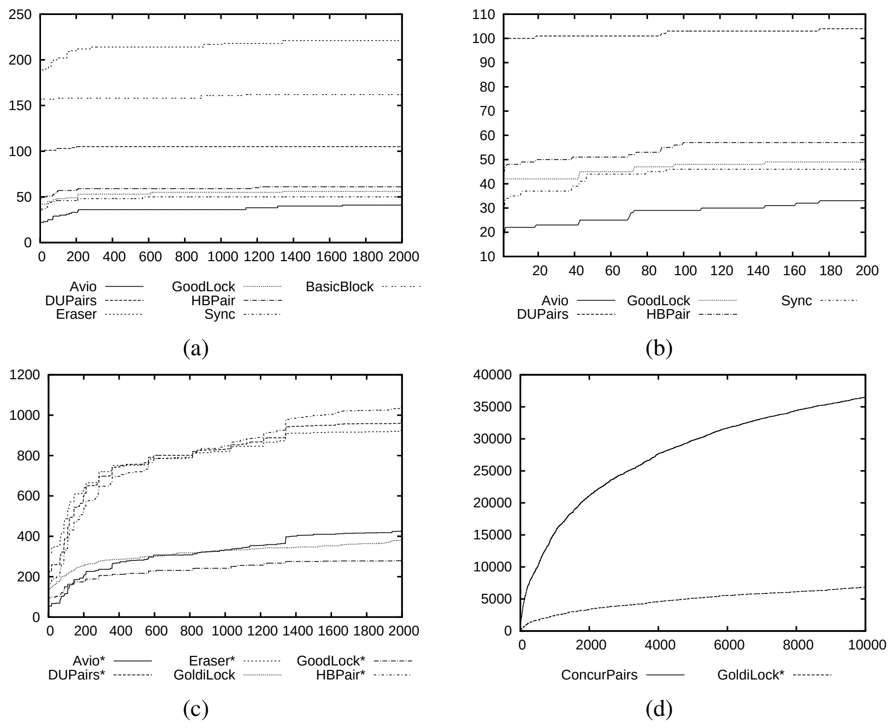

Fig. 1. Saturation of different metrics on the Crawler test case (the horizontal axis gives the number of executions, the vertical axis gives the cumulative number of covered tasks)

four Eraser states. The DUPairs metric has also higher numbers of covered tasks but it is almost all the time stabilised.

The most interesting part of Figure 1(a) between 0 and 200 executions is zoomed in Figure 1(b). One can see that the saturation effect occurs earlier (at about 100 executions) for the HBPair and Sync metrics which both focus on synchronisation events only. The Avio metric (and also the Eraser metric which is not shown) that focus on accesses to shared variables saturate a bit later. The depicted curves demonstrate one further disadvantage of the concerned metrics—a presence of distinctive shoulders. A repeated execution of the test case does examine different concurrent behaviours (which is indicated by the later discussed metrics) but the metrics concerned in the figure are not able to distinguish differences in these behaviours, and therefore we can see clear shoulders in the curves (i.e., sequences of constant values). The presence of such shoulders makes automatic saturation detection harder.

Figure 1(c) demonstrates a positive effect of considering an extended context of the tracked events as proposed in Section 3. The metrics concerned in this sub-figure (i.e., Avio*, Eraser*, DUPairs*, HBPair*, GoodLock*, and GoldiLock) are able to distinguish differences in the behaviour of the executed tests more accurately, leading to shorter shoulders, bigger differences in the cumulated values, and a later occurrence of the saturation effect—indicating that the concerned metrics behave in a way much better for saturation-based testing. As can be seen from a similar jump in the obtained coverage of the HBPair*, Eraser*, and Avio* metrics at around 1300 executions, the extended context can sometimes have a dramatic influence. The jump is caused by the abstract thread identifiers. At the given point, a thread with a new abstract identifier appears, and all tasks involving this thread are different to those already known. This leads to a much more significant increase in the cumulative coverage. On the other hand, a special attention should be paid to the GoldiLock metric. This metric does not suffer from shoulders nor sudden, dramatic increases of the obtained coverage, and it reaches saturation near the saturation points of the other metrics. This is a very positive behaviour, and the GoldiLock metric is clearly winning here.

Figure 1(d) shows problems of metrics that are too accurate, namely, ConcurPairs and GoldiLock*. These metrics work fine for small test cases but when used on a bigger test case they tend to saturate late and produce enormous numbers of covered tasks.

Quantitative properties of the considered metrics in all our test cases can be seen in Table 3. The table shows for each metric and each test case three values computed from a set of 100 different random series consisting of 2,000 test executions. The columns labelled as *Total* show the average total number of distinct tasks produced by the metric. This number demonstrates a big disadvantage of the ConcurPairs coverage metric, namely, its problem with scalability. The metric produced nearly 5 million of distinct tasks for 2,000 executions of the TIDOrbJ test case which makes further analyses quite time demanding.

The columns of Table 3 labelled as *Average percentage* represent the ratio between the Total and average number of tasks covered within one execution. A high number in this column means that most of the total number of covered tasks were covered within one execution. The cumulative coverage under such metrics (for DUPairs, Eraser, and Sync) usually stabilises early or grows very slowly. In both of these cases, the detection of saturation is problematic. Contrary, if the average percentage is too low (for ConcurPairs and GoldiLock*), the cumulative coverage grows for a very long time.

Finally, the columns of Table 3 labelled *Smooth percentage* give an insight in how smooth the growth of the accumulated coverage is. The column contains the ratio between the average number of the distinct cumulative coverage values reached under a metric when going through the considered executions and the number of test executions (2,000). High values (for ConcurPairs and GoldiLock*) mean that the cumulated coverage under the metric changed many times, and therefore there was contiguously growing. Low values (for Avio, DUPairs, Eraser, GoodLock, and Sync) mean that the cumulated coverage changed only a few times, and therefore there either occurred a fast saturation or there appeared long shoulders. Both of these phenomena are problematic for a good metric to be used in saturation-based testing.

The table also shows a disadvantage of the GoodLock* metric. The metric focuses on nested locking as was described in Section 3.2. If such a phenomenon does not occur in the tested program, the metric provides no information as can be seen in the Airlines and FtpServer test cases. On the other hand, the metric can provide additional information which cannot be directly inferred by other metrics in programs which contains this phenomenon. In total, the evaluation in Table 3 gives similar champions for a good metric to be used in saturation-based testing as what we saw in Figure 1(c). Namely, this is the case of the Avio*, Eraser*, DUPairs*, HBPair*, and GoldiLock metrics.

Table 3. A quantitative comparison of the metrics

	Dining phil.			Airlines			Crawler			FtpServer			TIDOrbJ echo		
	Total	Average %	Smooth %	Total	Average %	Smooth %	Total	Average %	Smooth %	Total	Average %	Smooth %	Total	Average %	Smooth %
Avio	6	47	0	17	60	1	40	22	1	529	45	10	822	50	8
Avio*	30	10	0	490	2	10	418	3	9	1023	33	16	3280	29	22
ConcurP.	4059	6	38	16730	6	85	20866	3	83	526280	6	100	4908100	2	100
DUPairs	18	76	0	43	97	0	105	81	1	330	92	2	1933	98	2
DUPair*	72	19	0	1401	3	9	921	11	8	646	82	3	3092	90	4
Eraser	29	76	0	73	96	0	217	64	2	684	88	4	2978	90	4
Eraser*	89	25	0	1429	5	8	861	19	5	1086	79	4	4886	83	6
GoldiLock	26	73	0	102	64	2	384	20	12	1091	61	9	6265	51	29
GoldiLock*	119	16	0	4217	1	20	3335	3	26	2210	47	12	10434	41	46
GoodLock	9	56	0	0	-	0	57	52	1	0	-	0	321	63	3
GoodLock*	22	23	0	0	-	0	258	17	4	0	-	0	915	34	6
HBPair	6	62	0	25	79	0	61	39	1	13	73	0	131	70	2
HBPair*	29	13	0	1013	2	13	984	4	12	28	49	0	420	46	5
Sync	8	56	0	27	78	0	49	46	1	22	66	0	172	79	2

6 Conclusions and Future Work

We have proposed a methodology of deriving new coverage metrics to be used in testing of concurrent software from dynamic (and possibly also static) analyses designed for discovering bugs in concurrent programs. Using this idea, we have derived several new concrete metrics. We have performed an empirical evaluation of these metrics, which has shown that several of them are indeed better for use in saturation-based and search-based testing than various previously known metrics.

As an additional advantage of the metrics that we have proposed, we can mention their better applicability in debugging. For debugging, understandability of each coverage task is important. We believe that tasks generated by our metrics provide much more problem-related information to the tester than existing metrics such as ConcurPairs or DUPairs. The tester can track the threads and objects that appear in the covered tasks to their place of creation or use some additional information (e.g., a lockset) present in the tasks to better understand what happened during the witnessed executions.

In the future, more experimental evidence about the proposed metrics should be obtained to further explore their properties. Metrics based on other dynamic as well as static analyses could be considered too. Finally, an evaluation of the metrics within the entire framework of search-based testing should be done.

Acknowledgement. This work was supported by the Czech Science Foundation (projects no. P103/10/0306 and 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), and the internal BUT project FIT-11-1.

References

1. S. Bensalem and K. Havelund. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Proc. of HVC'05*, LNCS 3875, Springer-Verlag, 2005.
2. A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of Synchronization Coverage. In *Proc. of PPoPP'05*, ACM Press, 2005.
3. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
4. 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.
5. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*, ACM Press, 2007.
6. D. Hovemeyer and W. Pugh. Finding Concurrency Bugs in Java. In *Proc. of PODC'04*, ACM Press, 2004.
7. P. Joshi, C.-S. Park, K. Sen, and M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proc. of PLDI'09*, ACM Press, 2009.
8. V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and Accurate Static Data-race Detection for Concurrent Programs. In *Proc. of CAV'07*, LNCS 4590, Springer-Verlag, 2007.
9. B. Křena, Z. Letko, T. Vojnar, and S. Ur. A Platform for Search-based Testing of Concurrent Software. In *Proc. of PADTAD'10*, ACM Press, 2010.
10. S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proc. of ASPLOS'06*, ACM Press, 2006.
11. J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proc. of POPL'05*, ACM Press, 2005.
12. P. McMinn. Search-based Software Test Data Generation: A Survey: Research Articles. *Software Testing, Verification, and Reliability*, 14(2):105–156, 2004.
13. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proc. of OSDI'08*, USENIX Association, 2008.
14. E. Pozniansky and A. Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proc. of PPoPP'03*, ACM Press, 2003.
15. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. of SOSP'97*, ACM Press, 1997.
16. E. Sherman, M. B. Dwyer, and S. Elbaum. Saturation-based Testing of Concurrent Programs. In *Proc. of ESEC/FSE'09*, ACM Press, 2009.
17. J. Soriano, M. Jimenez, J. M. Cantera, and J. J. Hierro. Delivering Mobile Enterprise Services on Morfeo's MC Open Source Platform. In *Proc. of MDM'06*, IEEE CS, 2006.
18. E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi. Forcing Small Models of Conditions on Program Interleaving for Detection of Concurrent Bugs. In *Proc. of PADTAD'09*, ACM Press, 2009.
19. C. von Praun and T. R. Gross. Object Race Detection. In *Proc. of OOPSLA'01*, ACM Press, 2001.
20. A. Williams, W. Thies, and M. D. Ernst. Static Deadlock Detection for Java Libraries. In *Proc. of ECOOP'05*, LNCS 3586, Springer-Verlag, 2005.
21. C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path Coverage for Parallel Programs. In *Proc. of ISSTA'98*, ACM Press, 1998.