Combining Model-Based Testing and Runtime Monitoring for Program Testing in the Presence of Nondeterminism

Paolo Arcaini
Dipartimento di Informatica
Università degli Studi di Milano, Italy
Email: paolo.arcaini@unimi.it

Angelo Gargantini
DIIMM
Università di Bergamo, Italy
Email: angelo.gargantini@unibg.it

Elvinia Riccobene
Dipartimento di Informatica
Università degli Studi di Milano, Italy
Email: elvinia.riccobene@unimi.it

***Abstract*—In case of underspecified or not fully predictable systems, models specifying system behaviors are nondeterministic. Nondeterminism poses several challenges for the validation and verification activities, including the problem of inconclusive tests in model-based testing with model checker. It is a validation technique that use model checker counterexamples as test cases.**

In this paper, we tackle the problem of testing nondeterministic systems by combining *model-based testing* and *runtime conformance monitoring*: the input sequences of the tests are automatically generated from nondeterministic models; then their execution is runtime monitored to check conformance of the code w.r.t. its specification. This technique provides an oracle for the test data, it never bears inconclusive responses, and it allows measuring the requirement coverage.

The approach uses the Abstract State Machines as formal method for specification purposes and Java as implementation language. As a proof of concepts, the Tic-Tac-Toe game is taken as example of a system with nondeterministic behavior (both at specification and code levels).

I. Introduction

In the software system life cycle, models are used to represent system behavior in a high-level abstract way. Models are often *internally* nondeterministic, i.e., given the same input sequences at different times, different output sequences can be produced. This distinguishes from *external* nondeterminism due to the unknown behavior of the environment. Internal nondeterminism can be due to i) not fully predictable systems, ii) underspecification because some implementation choices are left abstract, iii) model abstraction used to reduce complexity (state space) or to remove constructs, which are difficult for simulation and verification (e.g., time aspects). For instance, in object oriented modeling, nondeterminism allows to better reflect inherent nondeterminism of the domain and reduce complexity [4].

The presence of internal nondeterminism makes all the common validation and verification model-based activities, e.g., model-based testing, more complex.

Model-based testing (MBT) is accepted as a fully automated, flexible, and efficient technique to generate test cases from models that can lead to more effective testing [16]. MBT overcomes some limitations of the white box software testing. It addresses the *test oracle problem*, which is still an open problem in the context of software testing: in MBT, models are used as oracles since expected outputs are generated together with the inputs. However, nondeterminism poses several challenges to MBT. For instance, it is well known that derivation of tests for nondeterministic models is more computationally difficult than for deterministic models, or even impossible [1].

Research on test generation for nondeterministic systems has resulted in numerous approaches differing in how test cases are generated from models and how they are represented. Two classical approaches are Labelled Transition Systems (LTS) (that are sometimes also called I/O automata) with the implementation relation **ioco** (i.e., Input-Output Conformance) [7], [23], where tests are represented as trees (generally as particular LTSs), and those for Finite State Machines (FSM) [21], where tests are represented as sequences (called *checking sequences*). Having tests as sequences avoids generating huge precomputed test cases in order to deal with all possible responses of the system under test. Indeed, test sequences try to sample a large state space rather than attempting to exhaustively represent it. However, for those approaches where test cases are linear sequences of execution states, the nondeterminism of the model poses a challenge for testing the implementation. Typically, the implementation passes the test (generated by the model-based approach) if, executed on the inputs provided by the test, it returns the expected outputs. In case of nondeterminism, the implementation could deviate from a test case, taking a different but valid execution path, and the test case would falsely fail. Therefore, in case of deviation, no conclusion can be achieved: neither that the implementation fails nor that it passes the test. These tests are usually called *inconclusive*.

The approach we here suggest tries to overcome this limitation. We combine *model-based testing*, used to automatically generate, from nondeterministic specifications, the inputs of the test cases (*test data*), with *runtime monitoring* which provides an oracle that never bears inconclusive responses. Our approach falls in the category of *online testing* [7], [27], which consists in combining into a single algorithm test derivation from models and test execution over the system under test. Although for nondeterministic systems it is possible to build all the desirable test cases before testing, structured as graphs in LTS or as checking sequences for the FSM, the online testing offers the advantages of generating test cases at run time, rather than pre-computing a finite transition system and its traversals [27].

Among the different techniques existing for MBT, we here exploit the model checker capability to generate counterexamples upon (trap) property violation, and to interpret counterexamples as tests [10]. Model checkers embrace various sophisticated optimization techniques to cope with state explosion problem, such as BDDs, partial order, and SAT. Nondeterminism is not a problem for a model checker itself either, but this technique suffers from the problem of inconclusive tests in case of nondeterministic models. This is here solved by monitoring at runtime the test execution and by checking, at each step, the conformance of the code w.r.t. its specification, even at the nondeterministic points. That avoids us to stop the test execution and discard the test, when the test deviates from the expected outputs obtained from the test sequence given by the model checker. A further advantage is that we are able to check, at each step, which testing requirements are achieved, so having a measure of adequacy and avoiding redundant tests. We still provide user-guided control over test execution by allowing the user to select particular behaviors (expressed in terms of test predicates) to test.

On the other hand, runtime monitoring can also benefit from our approach. Runtime monitoring does not suffer from the test oracle issue, but there is still the problem of selecting relevant inputs, and of measuring the confidence that the runtime monitoring covered all the possible system behaviors.

The approach is implemented using the Abstract State Machines (ASMs) [5] as formal method for specification purposes and Java as implementation language. With respect to other formalisms like the LTS and the FSM, the ASM framework provides a solid mathematical foundation to deal with arbitrarily complex states. In particular, ASMs can deal with state variables and environment inputs (external nondeterminism), and has a compact construct (the choose rule) to represent the internal nondeterminism. In [14] the model-based testing technique for deterministic ASM models has been presented, while runtime conformance verification of Java programs w.r.t. corresponding ASM specifications has been introduced in [2].

Note that test sequence generation from models in the context of FSM is a well studied theoretical problem, also in the presence of nondeterminism [20], [21], [1]. However, methods developed for classical FSM are rarely applied for real size specifications due to state explosion problem. The proposed combined use of ASM, model checking, and online testing tries to overcome this problem.

As a novel contribution of this paper, (a) we extend the model-based testing technique in [14] for nondeterministic ASMs, (b) we improve the runtime monitoring of Java code w.r.t. ASM models in case of nondeterminism, (c) we combine the two approaches to solve the problem of inconclusive tests, of generating relevant inputs, and of measuring the requirement coverage.

To experiment our approach, we select the Tic-Tac-Toe game as example of a system with nondeterministic behavior (both at specification and code levels). As most testing and runtime monitoring techniques, our method cannot guarantee completeness (i.e., every program bug is captured), so to assess its quality and to measure its fault detection capability, we apply mutation analysis obtaining satisfactory results. We perform also a comparison with two program-based testing techniques in order to evaluate if MBT can outperform white-box testing for nondeterministic programs.

Although we use ASMs as modeling language and Java as code language, the same idea can be applied to any model-based testing technique where test cases are given as sequences of execution states, and any runtime verification approach able to check conformance between implementation and specification.

The paper is organized as follows: Section II introduces the running case study, recalls basic definitions regarding the ASM formal method, and presents our previous work on model-based testing with model checker for ASMs and runtime conformance verification of Java programs w.r.t. ASM specifications. In Section III we extend both techniques of ASM-based testing and runtime conformance checking in the presence of nondeterminism. Section IV explains our approach to combine model-based testing with runtime monitoring, while Section V reports our experiments on the Tic-Tac-Toe example. Section VI relates our work with similar contributions, and Section VII concludes the paper.

II. Background

A. A simple nondeterministic model

As motivating example and running case study, we consider a Tic-Tac-Toe game where a human player challenges a computer program. The requirements include that only valid moves are accepted, i.e., each player can put her symbol (nought or cross) only in an empty cell and when it is her turn, until one wins. The system must be able to identify valid moves and ignore invalid moves, check if one player wins and if the game is tie. The user moves are monitored by the system, while the program decides its moves according to some strategies. At specification level, the designer does not want to detail how the computer will play, since the strategy may be complex, change in order to improve performance, and include some random choices. The computer decisions will be left unspecified as nondeterministic choices. However, the designer wants to be sure that the implementation satisfies the requirements of correctness listed above.

B. Abstract State Machines

Abstract State Machines (ASMs) [5] are an extension of FSMs, where unstructured control states are replaced by states with arbitrary complex data. The *states* of an ASM are multi-sorted first-order structures, i.e., domains of objects with functions and predicates defined on them. *Static* functions never change during any run of the machine. *Dynamic* functions are distinguished between *monitored* (only read by the machine

```
asm ticTacToe
import StandardLibrary
signature:
    domain Coord subsetof Integer
    enum domain Sign = {CROSS | NOUGHT | EMPTY}
    enum domain Status = {TURN_USER | TURN_COMP}
    enum domain ActionDomain = {U_MOVE | C_MOVE}
    enum domain ResDom = {PLAYING | U_WON | C_WON | TIE}
    //first argument is the row, second argument is the column
    controlled board: Prod(Coord, Coord) −> Sign
    controlled status: Status
    monitored uSelCol: Coord
    monitored uSelRow: Coord
    monitored action: ActionDomain
    controlled res: ResDom
    controlled numOfMoves: Integer
    derived winOnRow: Prod(Coord, Coord, Sign) −> Boolean
    derived winOnCol: Prod(Coord, Coord, Sign) −> Boolean
    derived winOnDiag: Prod(Coord, Coord, Sign) −> Boolean
definitions:
    domain Coord = {0..2}
    //derived functions definition

    rule r_makeMove($r in Coord, $c in Coord, $s in Sign) =
        par
            board($r, $c) := $s
            numOfMoves := numOfMoves + 1
            if(winOnRow($r, $c, $s) or winOnCol($r, $c, $s) or winOnDiag($r, $c, $s)) then
                if($s = CROSS) then
                    res := U_WON
                else if($s = NOUGHT) then
                    res := C_WON
                endif endif
            else if (numOfMoves = 8) then
                res := TIE
            endif endif
        endpar

    rule r_moveUser =
        if (status = TURN_USER and
                board(uSelRow, uSelCol) = EMPTY) then
            par
                r_makeMove[uSelRow, uSelCol, CROSS]
                status := TURN_COMP
            endpar
        endif

    rule r_moveComp =
        if(status = TURN_COMP) then
            par
                choose $r in Coord, $c in Coord with
                                        board($r, $c) = EMPTY do
                    r_makeMove[$r, $c, NOUGHT]
                status := TURN_USER
            endpar
        endif

    main rule r_Main =
        if(res = PLAYING) then
            if(action = U_MOVE) then
                r_moveUser[]
            else
                r_moveComp[]
            endif
        endif

default init s0:
    function status = TURN_USER
    function board($r in Coord, $c in Coord) = EMPTY
    function res = PLAYING
    function numOfMoves = 0
```

Fig. 1: ASM specification of Tic-Tac-Toe

and modified by the environment), and *controlled* (read and written by the machine).

ASM states are modified by *transition relations* specified by "rules" describing the modification of the functions interpretation from one state to the next one. There is a limited but powerful set of *rule constructors* including guarded actions (`if-then`) and simultaneous parallel actions (`par`). The constructor `choose` expresses nondeterminism concisely.

It is also possible to specify state *invariants*.

A *computation* of an ASM is a finite or infinite sequence $s_0, s_1, \ldots, s_n, \ldots$ of states of the machine, where s_0 is an initial state and each s_{i+1} is obtained from s_i by executing the machine (unique) *main rule*. An ASM can have more than one *initial state*. Because of the nondeterminism of the choose rule and of the environment moves, an ASM can have several different runs starting in the same initial state.

The Asmeta framework[1] is used for the development and simulation of ASM models. Fig. 1 shows the ASM specification of the Tic-Tac-Toe.

C. Model-based testing for ASMs

One of the main applications of MBT for the ASMs, consists in automatically generating tests from ASM models [13].

Testing requirements are represented by *test predicates*, which are formulas over the state determining if a particular testing goal is reached. A coverage criterion C is a function that, given an ASM, produces a set of test predicates. A *test sequence* or *test* is a finite computation. A set of tests, called *test suite*, satisfies a coverage criterion C if each test predicate generated by C is satisfied in at least one state of a test sequence in the test suite.

Several coverage criteria have been defined for ASMs [13]. For instance, one of the basic criteria for ASMs is the *decision coverage*. A test suite satisfies the *decision coverage* criterion if, for every decision d_i of a rule r_i (e.g., the guard of a conditional rule), there exists at least one state in a test sequence in which r_i fires and d_i evaluates to *true*, and there exists at least a state in a test sequence in which r_i fires and d_i evaluates to *false*. For example, the test predicate for the coverage at *false* of the guard of the conditional rule in *r_moveComp* (see Fig. 1), is the following predicate:

```
DC_r_moveComp_CRf: res = PLAYING and action != U_MOVE and
                   status != TURN_COMP
```

In order to build test suites satisfying some coverage criteria, we use a technique based on the capability of the model checkers to produce counterexamples. The method consists of the following steps:

1) The test predicates set $\{tp_i\}$ is derived from the ASM according to some desired coverage criteria.

2) The ASM specification is translated into the language of the model checker.

[1]http://asmeta.sourceforge.net/

```
-------------
State 1
-------------
res = PLAYING
status = TURN_USER
action = U_MOVE
uSelRow = 1
uSelCol = 2
numOfMoves = 0
board(0, 0) = UNDEF
....
-------------
State 2
-------------
res = PLAYING
status = TURN_COMP
action = C_MOVE
uSelRow = 2
uSelCol = 2
numOfMoves = 1
board(0, 0) = UNDEF
...
board(1, 2) = CROSS
board(2, 0) = UNDEF
...
-------------
State 3
-------------
res = PLAYING
status = TURN_USER
action = C_MOVE
uSelRow = 2
uSelCol = 2
numOfMoves = 2
board(0, 0) = NOUGHT
board(0, 1) = UNDEF
...
board(1, 2) = CROSS
board(2, 0) = UNDEF
...
```

Fig. 2: ASM test for the test predicate DC_r_moveComp_CRf

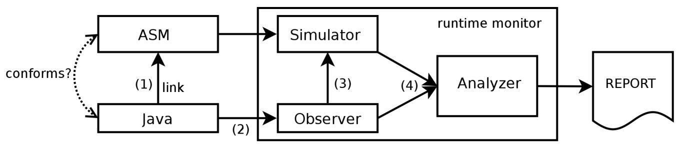

Fig. 3: CoMA: Conformance monitoring through ASM

3) For each test predicate tp_i the *trap property* `never`(tp_i) is proved. If the model checker finds a state s where tp_i is true, it stops and returns as counterexample a state sequence leading to s: from such sequence it is possible to build a test that covers tp_i. If the model checker explores the whole state space without finding any violation of the trap property, then the test predicate is said *unfeasible* and ignored. In the worst case, the model checker terminates without exploring the whole state space and without finding a violation of the trap property (i.e., without producing any counterexample), usually because of the state explosion problem. In this case of model checker *inconclusive* result, the user does not know if the test predicate is unfeasible or if a test exists but it is too difficult to find.

In this paper, to derive test sequences from ASM models, we use the ATGT tool [14], based on the model checker Spin [17]. Fig. 2 shows the ASM trace obtained from the counterexample produced by Spin under the violation of the trap property of the test predicate DC_r_moveComp_CRf.

D. CoMA: Conformance Monitoring through ASMs

CoMA is a technique for runtime monitoring of Java programs through ASMs [2]. The runtime monitor observes the behavior of a Java code and determines its correctness w.r.t. the ASM specification working as an oracle of the expected behavior: while the software system is executed, the monitor checks conformance between the observed state and the expected state.

Fig. 3 shows the structure of the proposed framework:

- A *link* between a Java class and an ASM must be provided (1) in order to describe the conformance relation; a set of annotations is used to this purpose.
- The *Observer* evaluates when the Java object *observed* state is changed (2), and leads the corresponding ASM to perform a machine step (3).
- The *Analyzer* evaluates the conformance between the Java execution and the ASM behavior (4). If a conformance violation is detected, a trace in form of counterexample can be recorded for debugging.

Fig. 4 shows a Java implementation of the Tic-Tac-Toe. It has been decorated with a set of annotations to link it to its formal ASM specification in Fig.1.

A complete description of CoMA can be found in [2]. We here report only some basic definitions.

Let C be a Java class, O_C an object of C, and ASM_C an ASM model of the expected behavior of any object of C.

We define *observed state*, $OS(C)$, the set of all public fields, and *pure*[2] public methods of the class C the user wants to observe. We define *changing methods*, $CM(C)$, the non pure methods whose execution can change the element values of $OS(C)$ and that the user wants to monitor.

A *Java step* is defined by the triple (s, m, s'), being m a method of C, and s and s' the states of an object O_C before and after the method execution. A *change step* is a Java step (s, m, s') where $m \in CM(C)$.

Definition 1: **State Conformance** We say that *a state s of O_C conforms to a state S* of ASM_C if all observed elements of C have values in O_C conforming to the values of the functions in ASM_C linked to them; i.e.,

$$conf(s, S) \equiv$$
$$\forall e \in OS(C) : val_{Java}(e, s) \overset{conf}{=} val_{ASM}(link(e), S)$$

Definition 2: **Step Conformance** We say that *a change step* (s, m, s') of an instance O_C, with m a method of C, *conforms with a step* (S, S') of ASM_C if $conf(s, S) \wedge conf(s', S')$.

$$\begin{tikzcd} ASM_C & S \arrow[r, "simulation\ step"] & S' \\ O_C & s \arrow[u, dotted, "conf"] \arrow[r, "invocation\ of\ method\ m"] & s' \arrow[u, dotted, "conf"'] \end{tikzcd}$$

Definition 3: **Runtime Conformance** Given an *observed computation* of a Java instance O_C, we say that C is *runtime conforming* to its specification ASM_C if the following conditions hold:

1) the initial state s_0 of the computation of O_C *conforms* to the initial state S_0 of the computation of ASM_C, i.e., it yields $conf(s_0, S_0)$;
2) every observed change step (s, m, s') with s the current state of O_C, *conforms* with the step (S, S') of ASM_C with S the current state of ASM_C;
3) no specification invariant of ASM_C is ever violated.

III. Dealing with nondeterminism

A. Test generation in the presence of nondeterminism

We extend the approach presented in Sect. II-C in order to deal with nondeterministic ASM specifications containing a `choose` rule of the form:

[2]A method is *pure* when its execution does not affect the program state.

```
@Asm(asmFile = "models/ticTacToe.asm")
public class TicTacToe {
    @FieldToFunction(func = "board")
    public Sign [][]  board;
    private Random rnd;
    @FieldToFunction(func = "numOfMoves")
    public int  movesExecuted = 0;
    @FieldToFunction(func = "status")
    public Status status;
    private Sign winner;

    @StartMonitoring
    public TicTacToe() {...}

    @RunStep(setFunction = "action", toValue = "U_MOVE")
    public void execUserMove(@Param(func="uSelRow") int r, @Param(func="uSelCol") int c) {
        if (winner == null && status == Status.TURN_USER && board[r][c] == Sign.UNDEF) {
            board[r][ c] = Sign.CROSS;
            movesExecuted++;
            status = Status.TURN_COMP;
            if (checkWinner(r, c, Sign.CROSS))
                winner = Sign.CROSS;
        }
    }
    @MethodToFunction(func = "res")
    public String getWinner() { ...  }

    @RunStep(setFunction="action", toValue = "C_MOVE")
    public void execComputerMove() {
        if (winner == null && movesExecuted < 9 &&
                status == Status.TURN_COMP) {
            int r = −1;
            int c = −1;
            do {
                r = rnd. nextInt (3);
                c = rnd. nextInt (3);
            }
            while(board[r][c]!=Sign.UNDEF);
            board[r][ c] = Sign.NOUGHT;
            movesExecuted++;
            status = Status.TURN_USER;
            if (checkWinner(r,c,Sign.NOUGHT))
                winner = Sign.NOUGHT;
        }
    }

    private boolean checkWinner(int r,int c,Sign sign) { ...  }
}
```

Fig. 4: Java code of Tic-Tac-Toe

choose \$x **in** $\{a_1, a_2, ... , a_n\}$ **with** cond(\$x) **do** R[\$x]

where $\$x$ is a variable ranging in the set $\{a_1, a_2, ..., a_n\}$ and satisfying *cond(\$x)* in order to fire rule R.

To deal with nondeterminism we have to extend the coverage criteria and to provide a translation of the `choose` rule in Promela, the language of Spin.

The coverage criteria in [13] are extended in the presence of the `choose` rule. For instance, the basic rule coverage must require both that the `choose` rule is executed and that the rule R inside the `choose` rule is executed as well. This means that variable $\$x$ must take at least a value in the set $\{a_1, a_2, \ldots, a_n\}$ with *cond*($\$x$) true. Moreover, all the test predicates obtained from the rule R inside the `choose` rule must be enriched by taking into account the guard in the `choose` rule and the value of $\$x$.

In order to translate the constructor `choose` to Promela, we use the nondeterministic guarded case selection shown below, where $\$x \leftarrow a_{\text{i}}$ denotes $\$x$ substituted by a_{i}.

```
if
:: cond[$x← a1]−>R[$x← a1]
...
:: cond[$x← an]−>R[$x← an]
fi
```

If more than one guard is true, Spin nondeterministically chooses only one corresponding rule to be executed[3].

B. Runtime monitoring in the presence of nondeterminism

Definition 3 assumes that, in any computation, the next state of a Java class instance O_C and of its specification ASM_C is unique. Thus, the definition is adequate for deterministic systems in which the nondeterminism is limited to monitored (external) quantities (e.g., which method has been called or what values have been used as actual parameters). Once these quantities are fixed by the environment, the evolution of the system is, however, deterministic.

For dealing with internal nondeterminism, our conceptual framework must be extended – from now on we refer to internal nondeterminism only. The following scenarios can be identified:

- Nondeterministic Java class and nondeterministic ASM specification. A class method has nondeterministic behavior (for instance it contains a call to a method in the class `java.util.Random`), as well as the abstract specification.
- Deterministic Java class and nondeterministic ASM specification. This situation arises when the ASM model is more abstract (with less implementation details) than the corresponding Java code. Bekaert and Steegmans have shown that nondeterminism in the behavioral specifications of object oriented conceptual models can simplify the representation of complex functionalities and achieve a better separation of concerns [4].

In case a class C or its model ASM_C are nondeterministic, the next computational state of O_C or ASM_C is not always uniquely determined, and, therefore, their conformance, according to Def. 3, may fail not because of a non conformant behavior of the implementation, but because O_C and ASM_C may choose two next states which are not conformant. We here refine points 1 and 2 of Def. 3 of runtime conformance in case of nondeterminism, distinguishing between weak and strong conformance. For the *weak* conformance, we require that the next step of O_C is state-conforming with *at least one* of the next states of the specification ASM_C. For the *strong* conformance, we require that the next step of O_C is state-conforming with *one and only one* of the next states of the specification.

Definition 4: **Weak [Strong] runtime conformance** We

[3]Nondeterministic choices could be translated in Promela also by means of monitored variables, but the proposed translation is better because it does not increase the size of the state, since Spin does not have to retain information about the chosen $\$x$.

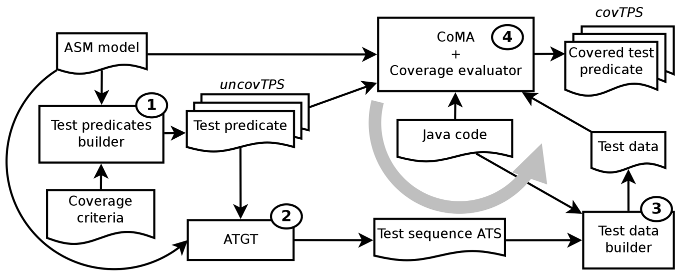

Fig. 6: Process for testing Java programs by combining ATGT and CoMA

say that C is *weakly [strongly]* runtime conforming to its specification ASM_C if the following conditions hold:

1) the initial state s_0 of the computation of O_C conforms to *at least one [one and only one]* initial state S_0 of the computation of ASM_C, i.e., $\exists$ [$\exists!$] S_0 initial state of ASM_C such that $conf(s_0, S_0)$;
2) for every change step (s, m, s') with s the current state of O_C, $\exists$ [$\exists!$] (S, S') step of ASM_C with S the current state of ASM_C, such that (s, m, s') is *step conforming* with (S, S').

Currently, our monitoring system can only deal with strong conformance. A similar assumption is made in nondeterministic FSMs testing [15], where the FSM is required to be *observable*, i.e., in each state the transition taken can be deduced from the applied input and the produced output.

Therefore, in case of nondeterminism, during the runtime monitoring our system chooses, among the next states of the ASM, the unique state that conforms to the Java state. Fig. 5a depicts this situation: given the Java state s' produced by the execution of the method m, only one of the next states S'_j of the ASM is state conformant with s'. In the Tic-Tac-Toe example, the method m could be `execComputerMove` that, given a board configuration, chooses to place a nought in (0, 2): the obtained Java state is conformant with only one of the next states of the corresponding ASM model (see Fig. 5b).

If there is more than one next state conformant (weak conformance), instead, the system does not know which one to choose and currently rises an exception. Weak conformance will be considered for future work.

IV. Combining MBT and Runtime Monitoring

In this section we explain how our approach combines model-based testing and runtime monitoring. Fig. 6 depicts the process we propose.

1) A set of test predicates $uncovTPS$ is built from an ASM and a set of coverage criteria.

2) An uncovered test predicate tp is randomly chosen from $uncovTPS$. ATGT produces, if possible, an abstract test sequence *ATS* that covers tp (see Sect. II-C). If tp is unfeasible, it is removed from the collection, while in case of model checker inconclusive result (maybe because of the state explosion problem), the process continues with another test predicate.

3) The test data builder translates *ATS* to a concrete input sequence (*test data*) for the Java code (see Sect. IV-A).

4) The test data are executed and runtime monitored through CoMA, which provides the oracles for the test data and evaluates the test predicate coverage (see Sect. IV-B). During the test data execution, the test predicates that are covered are removed from $uncovTPS$ and added to $covTPS$, the set of covered test predicates. The process restarts from point 2 until a desired coverage is reached (see Sect. IV-C).

In our process the test data generation and the test execution are combined together: a single test is executed right after it has been constructed. Such approach permits to build only the necessary test data. Sometimes this approach is called *online* testing [27], that distinguishes from traditional *offline* testing where a complete test suite is built before the test execution.

A. Test data construction

In Sect. II-C we have seen a procedure to derive, from a specification, test sequences that cover some test predicates. From each abstract test sequence *ATS*, the tool derives concrete Java test data consisting of a sequence of method calls. The expected outputs in the *ATS* are discarded and the concrete tests do not contain any oracle. The procedure that identifies the inputs in a test sequence and maps them in method invocations with values for their parameters exploits the Java annotations[4] used to implement the linking function of CoMA:

- The value of the monitored function in the `@RunStep` annotation (e.g., `action` in the Tic-Tac-Toe example) identifies what method must be called.
- The values of the monitored functions linked in the `@Param` annotations of the (possible) method formal parameters are used as actual parameters in the method invocation (e.g., the formal parameters `r` and `c` of method `execComputerMove` are connected to the monitored functions `uSelRow` and `uSelCol`).

```
public void testDCrmoveCompCRf {
    TicTacToe t = new TicTacToe();
    t.execUserMove(1, 2);
    t.execComputerMove();
    t.execComputerMove();
}
```

Fig. 7: Test data derived from the counterexample of the trap property of the test predicate DC_r_moveComp_CRf (Fig. 2)

Fig. 7 shows the test data produced starting from the counterexample shown in Fig. 2. In each state, the value of the monitored function `action` (that is linked in the `@RunStep` annotations) is used to identify what method must be executed: if its value is `U_MOVE`, the method `execUserMove` is executed; otherwise, if its value is `C_MOVE`,

[4]`@RunSteps` identify the changing methods (set *CM*). The annotation has two attributes: *setFunction* specifying the name of a monitored function of the ASM model, and *toValue* specifying the value that the function must be set to, when the corresponding method is executed.
`@Param` annotates each parameter of the changing methods. It has an attribute *func* specifying the name of a monitored function of the ASM model: when the corresponding method is executed, the function is set to the value of the actual parameter.

$$\begin{tikzcd}
(O_C) & & (ASM_C) & \\
s \arrow[rr, dotted, "conf"] \arrow[d, "m"] & & S \arrow[dl, "step"'] \arrow[d, "step"] \arrow[dr, "step"] & \\
s' \arrow[rr, bend right, "conf(s',S'_j)"'] & S'_1 & \dots \mathbf{S'_j} \dots & S'_n
\end{tikzcd}$$

(a) Strong conformant step

	Instance of TicTac-Toe.java	Simulation of ticTacToe.asm	
State i	O | X | X | X | | O |	O | X | X | X | | O |	
State $i+1$	O | X | X | X | O | O |	O | X | X O | X | | O |	O | X | X | X | O | O |
		O | X | X | X | O | O |	O | X | X | X | | O | O

(b) Strong conformant step: Tic-Tac-Toe

Fig. 5: Strong conformance

the method `execComputerMove` is executed[5]. When the method `execUserMove` must be executed, the values of the monitored functions `uSelRow` and `uSelCol`, that are linked in the `@Param` annotations of its formal parameters `r` and `c`, are used as actual parameters for `r` and `c`.

Note that we assume that the implementation is *input enabled*, i.e., it never refuses any input. *Input enabledness* is a common assumption in MBT; in the ioco theory, for example, the implementation is represented by an IOTS, i.e., an LTS in which *outputs are initiated by the system and never refused by the environment, and inputs are initiated by the environment and never refused by the system* [24]. Input enabledness is also assumed in [6]. In our setting input enabledness means that any method can always be called with any (type-correct) values for its parameters; for example, the test data in Fig. 7 calling the method `execComputerMove` twice in a row, is correct, although only the first one actually changes the state.

B. Using CoMA as Test Oracle and Coverage Evaluator

In our approach we do not derive the oracle from the test sequence, as done in classical MBT for deterministic systems, but we use runtime monitoring to provide the oracle. If CoMA detects a not conforming behavior during monitoring, it signals a failure.

At each Java step, CoMA also checks which test predicates are covered; note that a test predicate may be not covered by *its* test: if the implementation, due to some internal nondeterminism, chooses a different behavior, the observed behavior may not cover the test predicate which the test sequence is generated for. In this case, the test predicate is kept in the collection of uncovered predicates $uncovTPS$. Note, however, that a test predicate can be removed from the collection even during the execution of test cases generated for other test predicates.

[5]Note that the repetition of method call `execComputerMove` is intentional and permitted as explained later.

C. Using test predicates as a measure of conformance

The aim of runtime verification techniques is to observe a system while it is running and determine if it assures some properties. Empirically, the more the system is executed and monitored, the higher is the confidence that the system is correct. But, how to measure such degree of confidence? To do this we can use coverage criteria. The idea is using CoMA not only to verify that the implementation is conformant with the specification, but also to identify what test predicates generated by MBT have been covered.

We introduce a *conformance index*

$$CI = \frac{\#tpsCovered}{\#tpsFeasible} \tag{1}$$

that provides an indication of *how deeply* a system has been monitored. CI could be used to decide when to interrupt the runtime monitoring: when CI becomes greater than a threshold K, we are confident enough that the system is correct, and we stop monitoring.

V. Experiments

To evaluate our approach we use the Tic-Tac-Toe as case study. The specification of the system is reported in Fig. 1, while its implementation is given in Fig. 4. The Java implementation randomly chooses the next computer move by taking an empty cell.

We have run all the experiments on a Linux machine, Intel(R) Core(TM) i7, 4 GB Ram. For obtaining short counterexamples, we use the breadth-first-search option in Spin.

We consider only the structural coverage criteria: rule coverage, decision coverage, update coverage, and MCDC. Totally, we generate 258 test predicates, 27 of which are unfeasible. For each test predicate, we have always been able either to produce a counterexample or to prove its unfeasibility, since Spin has always terminated with a conclusive result.

As first experiment, we want to assess the viability of our method by applying the process described in Section IV and requesting that a given percentage P of (feasible) test predicates is covered. We have applied our technique for 20 times and computed the average of the data, including the process execution time. We have not been able to apply our technique with P greater or equal to 80% in a reasonable time (we put a time limit of one hour for each experiment). Table I reports all the data about the following indicators:

1) *Conformance Index (CI)*: the percentage of feasible test predicates actually covered. *CI* may be greater than P because a test may cover more test predicates than requested.

2) *Unfeasible*: the number of test predicates found unfeasible (also as % over all the unfeasible tps).

3) *Selected*: the number of test predicates selected, that is equal to the number of iterations in the process in Fig. 6. It may depends on

4) *Checked and not covered*: the number of test predicates for which a test has been generated but, due to some different choices in the implementation, they were not covered by their tests and neither by other tests.

5) *Java test length*: the number of Java statements executed in the tests.

6) *Mutation score*: to evaluate the capability of our approach to detect faults, we have applied mutation analysis by using the Javalanche tool [22]. The mutation score is the ratio between the faults detected over all the faults injected.

As shown in the table, one test is enough to cover 10% of test predicates, and in this case only around 20 Java instructions are executed with an already acceptable mutation score. By increasing P, all the quantities increase. In particular, the time and the number of *checked and not covered* test predicates increases more than linearly, while the mutation score reaches a maximum around 86%. We found that some test predicates represent behaviors which have a very low probability of being executed by the implementation and for this reason they are not covered. The fact that the mutation score reaches a maximum, is because some injected faults are not related to the behavior of the implementation, and special test cases should be developed for them. The CI is a good index of the fault detection capability of the monitoring activity.

Since we were already confident that the implementation actually conformed to its specification, we expected no fault in the implementation, and this was the case.

A. Comparison with code-based approaches

Although MBT is a very promising technique, in practice code-based tools are more used. For this reason, we consider of great interest a comparison of our approach with two state-of-the-art techniques that generate test cases (with oracles) from Java code, namely Evosuite and Randoop.

Evosuite [9] is a tool that automatically generates test cases by applying a search-based approach that generates and optimizes whole test suites, rather than generating distinct test cases directed towards distinct coverage goals. It is able to suggest oracles by adding small sets of assertions that summarize the current behavior. *Randoop* [19] generates unit tests using *feedback-directed random testing*, a technique inspired by random testing that uses execution feedback gathered from executing test inputs as they are created, to avoid generating redundant and illegal inputs. It allows annotation of the source code to identify methods to be omitted and observer methods to be used for assertion generation.

Both methods are able to generate test suites together with oracles, by capturing the current behavior of the system. This works well in order to protect against future defects breaking this behavior, but tends to generate *falsely failing* tests in correspondence of nondeterminism. We say that a test falsely fails, if it fails in one execution but there exists another execution in which it does not fail. For this reason, we had either to modify the generated tests in order to make them pass (auto+mod. in Table II) or generating them without assertions (where possible). In details, for Randoop, we have applied a mixed approach between those suggested in [19] (auto+remapping in Table II): some nondeterministic methods are ignored (like the `getWinner` method) because their return value cannot be used in an assertion, while we have preferred to keep other methods (like the `execUserMove` method) and remapping them in order to avoid nondeterministic behavior. Note that even without assertions a test has a residual fault detection capability due to some implicit oracles (e.g., no `NullPointer` exception).

No test suite generated by Evosuite or by Randoop is able to reach the mutation score obtained by our approach. Even the best Randoop test suite has a mutation score between the two worst test suites of ours, but it requires a much greater number of Java instructions and a comparable time to generate it. Evosuite produces smaller tests, but with a reduced mutation score. Although the comparison is not completely fair since nondeterminism significantly reduces the effectiveness of these code-based testing tools, our experiments show that MBT has still several advantages over program-based testing techniques in the presence of nondeterminism.

VI. Related work

A way to combine model-based testing and runtime verification is presented in [3]. A test case generator, starting from a model of the input domain given as a nondeterministic Java program, produces inputs for the application using the *Java PathFinder* model checker that has been extended in order to perform symbolic execution. Together with the inputs also temporal properties, that must be guaranteed during the execution, are produced. Then, the execution of the application over the inputs is monitored by the runtime verification framework *Eagle* that checks that the properties are satisfied. The main conceptual difference w.r.t. our approach is that the properties for runtime verification depend on the particular input, while in our case the specifications must be independently provided and are general for every input.

In [12], [11] the test case generation process using model checkers is extended in order to deal with nondeterminism. The authors present a technique that permits to discover if a deviation exists from the expected output during a test case

P(%)	10%	20%	30%	40%	50%	60%	70%
Conformance Index (CI) (%)	19%	26%	37%	46%	57%	66%	74%
# Unfeasible	0.31	0.25	0.5	1	1.67	6	15.67
Unfeasible (%)	0.31	0.25	0.5	1	1.67	6	15.67
# Selected	1	1.5	3.67	6.75	12.33	28	71
# Checked and not covered	0.86	1.25	3.33	5	10.33	19.5	51
Java test length	19.43	42.25	162.33	183.5	379.33	781	2045.67
Mutation score (%)	74.58	84.60	84.99	85.39	85.56	85.70	86.21
Time (seconds)	30.49	58.49	139.14	248.79	453.66	1036.12	2412.28

TABLE I: Experimental results

Tool	EvoSuite				Randoop		
Options	branch coverage		mutation		1k	10k	100k
	assert	no assert	assert	no assert		-	
Generation time (sec.)	1004	154	943	605	13	35	209
Oracles	auto+mod.	No	auto+mod.	No	auto+remapping		
N. of tests	12	12	63	86	914	8880	88666
Java test length	205	152	1070	1353	5126	62258	940686
N. of asserts	27	N/A	125	N/A	2	2	6
Execution time (sec.)	0.05	0.05	0.13	0.14	0.16	0.7	2.7
Mutation score avg (%)	29.84	30.65	54.04	26.82	11.7	60.9	81.4
Mutation score var	0.66	0.2	1.95	0.22	.25	5.9	0.1

TABLE II: Other approaches results

execution due to a nondeterministic choice: such test cases are classified as *inconclusive*. Starting from an inconclusive test, the proposed process can iteratively build a *tree-like* test case in which the alternative valid branches of a computation are considered. They also extend common coverage criteria for deterministic systems to nondeterministic systems. This approach differs from ours since we do not need to stop if the output deviates from the expected one during test execution. Indeed, the runtime monitor we use as oracle can discover if the nondeterministic choice of the implementation is valid.

The problem of test generation for nondeterministic systems is also dealt with in [6]. They compare the traditional technique based on *model* checking, with a *module* checking approach. Module checking is useful to verify open systems, i.e, systems interacting with the environment and whose behavior is influenced by this interaction. They propose techniques to derive tests from mutants of the original specification, distinguishing between *weak* and *strong* tests. A weak test executed on the mutant and on the original specification can produce different outputs; a strong test, instead, always produce different outputs. The problem of test execution, in particular of weak tests, is not tackled.

Two classical approaches for dealing with nondeterminism are based on FSMs [21], [20], [1] and LTS with its conformance relation ioco [7], [23]. Methods developed for FSMs are rarely applied for real size specifications due to state explosion problem. For instance, the Tic-Tac-Toe example would roughly require $3^9 \times 3 \simeq 59k$ states. Extended FSMs enriching FSMs with variables, events, and guards, to concisely represent complex systems, promise to overcome these limitations. EFSM can be seen as a particular class of ASMs.

In LTS, a test is a particular Input Output LTS (IOLTS). Tests have normally a tree-like structure, although in some approaches they are extended with verdicts and some additional properties. For instance, in TGV [18], a IOLTS has a complex behavior whose structure is a graph with possible loops. A IOLTS naturally deals with nondeterminism. The test provides some particular inputs to the implementation under test (IUT) and can accept any output provided by the IUT (i.e., it is input enabled), until it reaches a *pass* or *fail* state. For this reason, this approach is suitable for online testing. This approach, however, suffer from the problem that there is no symbolic representation of data, that so are encoded in action names representing concrete values, making the representation of complex systems very cumbersome and curbing the usability of test generation tools. In order to deal with this problem, Symbolic Transition Systems (STS) [8] have been proposed for symbolic representation of LTS, where the notion of data and data flow are founded on first order logic.

Moreover, the problem of traversing the IOLTS (also called *synthesis*) remains open and several problems in test synthesis can be understood as reachability problems. Synthesis is completely random in TorX [25], whereas in TGV [18] it is driven by some *test purposes* (similar to our test predicates), i.e., descriptions of behaviors to be tested.

In [28], a technique for testing concurrent Java components is presented. In such a scenario, if multiple threads are waiting for a resource, the order in which they are woken up is not predictable, so making the system nondeterministic. The proposed solution is an extension of the *ConAn* (Concurrency Analyser) tool that permits one to write oracles handling all the possible configurations that can result because of nondeterminism. The approach seems to be not very scalable, since all the possible configurations must be considered in the definition of the oracle. In our approach, instead, the high power expression of the `choose` rule permits to concisely describe any nondeterministic choice of any degree.

Our approach shares several features with that presented in [27], using the testing framework Spec Explorer [26]. In [27], the testing of reactive systems is seen as a game between the tester and the IUT. The conformance between an IUT and its specification is given in terms of *alternating simulation*, similar to our notion of runtime conformance. They share with us the use of the ASMs as specification language (they use the ASM syntax AsmL). However, we are able to address some open problems identified by the authors in [27]. A first one is the *scenario control* problem, i.e., that of generating strategies that obtain particular behaviors, since it is not unusual that a model program may correspond to an automaton with a large or even infinite number of transitions, and in such cases selecting the scenarios is of great importance. This problem is tackled in [26], where several different techniques for scenario control are presented. In our setting this problem is addressed in an uniform way by using testing criteria that provide goals for the testing. The *failure analysis* problem, i.e., understanding the cause of a failure after a long run, in our setting can be mitigated by requiring the model checker to provide the shortest counterexample for a particular goal. Our conformance index also tackles the *achieving and measuring coverage* problem of [27].

VII. Conclusions and Future Work

We present a technique that combines model-based testing and runtime monitoring in an effective manner in order to deal with nondeterminism. We extend our model-based testing technique, we improve the runtime monitoring of Java code w.r.t. ASM models, and we combine the two approaches to solve the problem of inconclusive tests, of generating relevant inputs, and of measuring the requirement coverage.

Regarding the runtime monitoring, we plan to extend the runtime framework for supporting weak conformance.

The initial experiments suggest that our method is viable. We plan to experiment it by using complex systems, and to evaluate whether the tests generated by MBT lose efficiency by increasing the size and the nondeterminism of the implementation. Indeed, we experienced that the higher is the degree of nondeterminism, the higher is the probability that a test predicate is not covered during its test case execution.

Finally, we plan to use a more advanced conformance index taking into account the kinds of involved coverage criteria by assigning a weight specifying the *importance* of each criterion.

References

[1] R. Alur, C. Courcoubetis, and M. Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC '95, pages 363–372, New York, NY, USA, 1995. ACM.

[2] P. Arcaini, A. Gargantini, and E. Riccobene. CoMA: Conformance Monitoring of Java programs by Abstract State Machines. In K. Sen and S. Khurshid, editors, *Runtime Verification. Second international Conference, RV 2011*, volume 7186 of *LNCS*. Springer, 2012.

[3] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3):209–234, May 2005.

[4] P. Bekaert and E. Steegmans. Non-determinism in conceptual models. In K. Baclawski and H. Kilov, editors, *Proceedings of the Tenth OOPSLA Workshop on Behavioral Semantics*, pages 24 – 34, 2001.

[5] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[6] S. Boroday, A. Petrenko, and R. Groz. Can a model checker generate tests for non-deterministic systems? *Electronic Notes in Theoretical Computer Science*, 190(2):3 – 19, 2007.

[7] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France*, volume 2067, pages 187–195. Springer-Verlag, 2001.

[8] L. Frantzen, J. Tretmans, and T. A. C. Willemse. A symbolic framework for model-based testing. In *Formal Approaches to Software Testing and Runtime Verification, FATES 2006 and RV 2006*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2006.

[9] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proc. of ACM SIGSOFT ESEC/FSE*, pages 416–419, 2011.

[10] G. Fraser and A. Gargantini. An evaluation of model checkers for specification based test case generation. In *ICST 2009, 1-4 April 2009, Denver, Colorado, USA*, pages 41–50. IEEE Computer Society, 2009.

[11] G. Fraser and F. Wotawa. Nondeterministic testing with linear model-checker counterexamples. In *Proc. of the 7th International Conference on Quality Software*, QSIC '07, pages 107–116, Washington, DC, USA, 2007. IEEE Computer Society.

[12] G. Fraser and F. Wotawa. Test-case generation and coverage analysis for nondeterministic systems using model-checkers. In *Proceedings of the International Conference on Software Engineering Advances*, ICSEA '07, pages 45–, Washington, DC, USA, 2007. IEEE Computer Society.

[13] A. Gargantini and E. Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation. *J.UCS*, 7:262–265, 2001.

[14] A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In *ASM 2003 - Taormina, Italy, March 2003. Proceedings, LNCS 2589*, 2003.

[15] R. M. Hierons. Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Trans. Comput.*, 53(10):1330–1342, Oct. 2004.

[16] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, Feb. 2009.

[17] G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[18] C. Jard and T. Jéron. Tgv: theory, principles and algorithms. *Int. J. Softw. Tools Technol. Transf. (STT)*, 7(4):297–315, Aug. 2005.

[19] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[20] A. Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 196–205. Springer, 2001.

[21] A. Petrenko, A. Simao, and N. Yevtushenko. Generating checking sequences for nondeterministic finite state machines. In *Proc. of the 2012 IEEE 5th International Conference on Software Testing, Verification and Validation*, ICST '12, pages 310–319, Washington, DC, USA, 2012. IEEE Computer Society.

[22] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Proc. of ESEC/ ACM FSE '09*, pages 297–298, August 2009.

[23] J. Tretmans. Test generation with inputs, outputs, and quiescence. In *Proc. of Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *LNCS*, pages 127–146. Springer, 1996.

[24] J. Tretmans. Model based testing with labelled transition systems. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 1–38. Springer, 2008.

[25] J. Tretmans and E. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003.

[26] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems

with Spec Explorer. volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer-Verlag, Berlin, Heidelberg, 2008.

[27] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *Proc. of the 10th ESEC/13th ACM SIGSOFT FSE*, ESEC/FSE-13, pages 273–282, New York, NY, USA, 2005. ACM.

[28] L. Wildman, B. Long, and P. Strooper. Dealing with Non-Determinism in Testing Concurrent Java Components. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, APSEC '05, pages 393–400, Washington, DC, USA, 2005. IEEE Computer Society.