# Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving

Pablo Abad*, Nazareno Aguirre†‖, Valeria Bengolea†‖, Daniel Ciolek‡, Marcelo F. Frias*‖,
Juan Galeotti‡, Tom Maibaum¶, Mariano Moscato§, Nicolás Rosner§, Ignacio Vissani§
*Instituto Tecnológico de Buenos Aires, Buenos Aires, Argentina. {pabad,mfrias}@itba.edu.ar.
†Universidad Nacional de Río Cuarto, Río Cuarto, Argentina. {naguirre,vbengolea}@dc.exa.unrc.edu.ar.
‡ Saarland University, Saarbrücken, Germany. galeotti@cs.uni-saarland.de.
§ Universidad de Buenos Aires, Buenos Aires, Argentina. {dciolek,mmoscato,nrosner,ivissani}@dc.uba.ar.
¶ McMaster University, Hamilton (ON), Canada. tom@maibaum.org.
‖ Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina.

***Abstract*—We present a novel and general technique for automated test generation that combines tight bounds with incremental SAT solving. The proposed technique uses incremental SAT to build test suites targeting a specific testing criterion, amongst various black-box and white-box criteria. As our experimental results show, the combination of tight bounds with incremental SAT, and the testing criterion driven approach implemented in our prototype tool FAJITA, enable us to effectively generate test suites for container classes with rich contracts, more efficiently than other state-of-the-art tools.**

| | Pex | Kiasan | Randoop | Autotest | EvoSuite | FAJITA |
|---|---|---|---|---|---|---|
| GC | 78% | 84% | 63% | 68% | 90% | 95% |
| BC | 75% | 92% | 69% | 71% | 90% | 98% |
| Time | 3h7m | 2h16m | 25h | 25h | 22h | 1h18m |

TABLE I
SUMMARY OF GOAL (GC) AND BRANCH COVERAGE (BC) PERCENTAGE, AND REQUIRED OVERALL INPUT GENERATION TIME.

## I. INTRODUCTION

Testing is a powerful and widely used technique for software quality assurance, which requires significant resources in most software development projects [20]. Finding appropriate data to test software is typically a manual activity. In the last few years an increasingly growing community that sees automated test data generation as an essential complement to manually built test suites has emerged. Many tools that automate test data generation have been developed based on a variety of generation techniques, targeting different kinds of programs and programming languages. Amongst these we have RANDOOP [23] and AutoTest [22] (based on random generation), Pex [27] (based on dynamic symbolic execution), Java PathFinder [29] and Kiasan [8] (based on model checking), Korat [3] (based on search), EvoSuite [12] (based on genetic algorithms) and UDITA [17] (based on model checking, over Java PathFinder). Some of these tools (e.g., RANDOOP, AutoTest and Korat) follow a *black-box approach* to test case generation, in the sense that the structure of the program under test is not considered during the test generation process. Other tools (e.g., Kiasan, Java PathFinder, EvoSuite and UDITA), take into account the structure of the program under test for the generation, and thus are driven by a *white-box approach*.

In this article we present a novel, flexible and push-button technique for test input generation, implemented in our prototype tool FAJITA. The experiments we present in this article show that FAJITA achieves high coverage in generating test cases involving heap allocated structures with rich contracts. Said coverage improves even with respect to the most effective tools targeting this domain. In Table I we present a summary of the experimental results we obtained on goal coverage and branch coverage, as well as the overall time required for test input generation. We used a benchmark consisting of 8 collection classes, 25 methods, 227 goals and 394 branching points. The FAJITA approach to test input generation presented in this paper makes the following contributions:

- It leverages the bug-finding mechanism introduced in TACO [16], our tool for bug-finding, to the testing domain. FAJITA is based on SAT-solving, and the TACO technique contributes a powerful mechanism based on symmetry reduction that allows us to remove a significant number of variables from the SAT problem. This is particularly effective in the context of rich contracts including structural data constraints.
- It makes a novel use of incremental SAT-solving for systematically and effectively building test suites that target a specific coverage criterion, amongst a variety of black-box and white-box criteria. This test generation process leads to test suites that are minimal and only contain terminating test inputs (the latter is essential for the automation of the execution process).
- We show that FAJITA produces better and faster coverage for container classes with rich contracts, compared to various state-of-the-art tools for test generation based on a wide range of techniques, including dynamic symbolic execution, random testing and genetic algorithms.

The article is organized as follows. In Section II we show how FAJITA makes use of tight bounds, a technique introduced with the TACO tool for bug-finding. In Section III, we present the generic, coverage criteria driven approach to test generation that FAJITA implements, which uses incremental SAT solving. This generic technique is tailored in Sections IV and V to make FAJITA target various black-box and white-

box testing criteria. In Section VI we present the experimental results that, in particular, give origin to Table I. Finally, in Sections VII and VIII we discuss related work and present our conclusions and proposals for further work.

## II. Tight Bounds for Improved SAT-Solving

Given a propositional formula $\varphi$, the SAT problem consists of finding a satisfying valuation for $\varphi$, or reporting its unsatisfiability. The worst-case time complexity of SAT is exponential on the number of propositional variables of the formula being solved, and therefore reducing the number of variables most times contributes to reducing its solving time.

Our approach to test input data generation is based on SAT-solving. It takes the specifications of the input and other constraints on it (e.g., regarding the coverage of a particular equivalence class according to a testing criterion), and translates these into a propositional formula whose satisfying valuations correspond to test inputs complying with the specification and the additional constraints. In order to improve the analysis time of the SAT-solving process underlying our approach, we take advantage of a technique put forward in [16], that: (*i*) automatically synthesizes symmetry breaking predicates [5], making the SAT solver produce only *canonical* representations of valid inputs, removing other redundant instances (symmetric to the canonical ones), and (*ii*) uses the above mentioned symmetry breaking constraints in order to automatically remove from the propositional formulas some variables which, due to these constraints, are guaranteed to be *false* in the solving process. This technique was implemented in the sequential Java analysis tool TACO [16], and has proved to be extremely useful, allowing us to remove in several of our case studies over $60\%$ of the propositional variables used for characterizing the input states, and exponentially improving the analysis times.

FAJITA uses Alloy [18] as a high level language suitable for accessing off-the-shelf SAT-solvers. Furthermore, the use that Alloy makes of KodKod [28] along the translation of Alloy models to propositional formulas is essential to our approach. KodKod allows one to prescribe bounds for Alloy fields. Each field $f$ is equipped with a relation $U_f$ (its upper bound), which imposes restrictions on the interpretations of this field: in any Alloy model M, $f \subseteq U_f$. Therefore, tuples that are not in $U_f$ cannot belong to $f$. The translation of Java code models Java fields as Alloy relations, and each tuple in such a relation is modeled with a propositional variable (whose truth value indicates whether the corresponding tuple is in the relation or not). Therefore, propositional variables corresponding to tuples that do not belong to $U_f$ can be directly replaced in the translation process with the truth value *false*. We exploit this facility in order to, as mentioned above, reduce the number of propositional variables. To automatically generate these bounds, FAJITA preprocesses the description of valid objects (provided under the form of class invariants and method preconditions) as described in [16], using a cluster of computers, and automatically computes tight (i.e., with as few pairs as possible) bounds for the fields of these objects. These bounds are stored in a repository, since they are often reused when generating test data for different testing criteria, when different methods in the same Java class are considered, or for different equivalence classes within a testing criterion. Thus, the cost of computing the bounds is amortized. The computed bounds, together with the additional constraints corresponding to coverage according to a pre-selected testing criterion, are used to build a KodKod model that is finally translated into a propositional formula in conjunctive normal form, amenable to most SAT-solvers.

## III. Test Data Generation using Incremental SAT-Solving

We now present a generic algorithm for test data generation that, when instantiated with an adequate description of a testing criterion, allows FAJITA to generate test suites that satisfy the criterion. In Sections IV and V we will show different instantiations to black-box and white-box coverage criteria. In order to explain how the algorithm works, let us denote by $\mathbb{S}$ the state space of valid inputs of a program under test. Any coverage criterion, be it white-box or black-box, induces an equivalence relation on $\mathbb{S}$. For instance, *branch coverage* considers that two inputs are equivalent if, when executing the program under test for each of these inputs, each decision point returns the same truth values (none, if the decision point is never reached, only true, only false, or both true and false, since a decision point may be visited more than once). Thus, testing criteria give origin to a collection $p_1, \ldots, p_k$ of (mutually disjoint) unary predicates on $\mathbb{S}$ that induce a *partition* on $\mathbb{S}$ in which each $p_i$ characterizes a particular equivalence class on the set of inputs.

Since our approach is based on propositional satisfiability, a *scope* must be imposed for the analysis. A scope basically establishes a bound on the number of objects for each class, and a limited range of values for numerical domains. Moreover, for white box coverage criteria, the scope also includes a limit for the code under analysis, in the form of a maximum number of loop iterations (this limit is used to unroll loops and make the code under analysis loop-free). Our algorithm for test generation guarantees that, if an equivalence class of the test criterion is coverable within a provided scope (bounds on the number of objects and the number of iterations), then an input corresponding to this class is eventually generated. Furthermore, the algorithm produces an *optimal* coverage, in the sense that exactly one test is produced for each equivalence class of the test criterion coverable within the provided scope. If a class is not covered, this does not necessarily mean that it is infeasible, since by using a larger scope a test input covering the class might be generated. However, one can employ a incremental approach with respect to the scope: start with scope 1 and cover as many equivalence classes as possible, and then start incrementing the scope to attempt to cover the remaining uncovered classes, until all classes are covered, or the analysis becomes infeasible. This is the approach that FAJITA follows, which has a significant impact in reducing path explosion in the analysis.

Within a particular scope, FAJITA makes use of another interesting form of incrementality, via *incremental SAT solving*. Incremental SAT-solvers have two distinguishing features.

```
Algorithm IncPartitionCoverage
Suite = empty;
rm_val = true;
while (I-SAT(alpha, rm_val))
    val = getValuation();
    Suite = Suite + getTestInput(val);
    rm_val = getPredicateFrom(val);
endwhile
```

Fig. 1. Partition coverage using incremental SAT Solving.

When a satisfying valuation for a formula $\varphi$ is found: (*i*) a new formula $\alpha$ can be added in order to prune the search space even further while searching for a new valuation satisfying (now) $\varphi \wedge \alpha$, and (*ii*) the internal state of the solver keeps track of the already traversed portion of the search space. A new search for a satisfying valuation for $\varphi \wedge \alpha$ will not revisit those states that have already been visited.

By `I-SAT(`$\varphi, \alpha$`)` we denote the invocation of the incremental SAT-solver on a formula $\varphi$, with the added information provided by formula $\alpha$. Notice that a sequence of calls `I-SAT(`$\varphi, \alpha_1$`)`,...,`I-SAT(`$\varphi, \alpha_m$`)` generates valuations that, respectively, satisfy formulas $\varphi \wedge \alpha_1$, $\varphi \wedge \alpha_1 \wedge \alpha_2$, ..., $\varphi \wedge \alpha_1 \wedge \cdots \wedge \alpha_m$.

Given the above mentioned predicates $p_1, \ldots, p_k$, our goal is to generate test input data covering each of the equivalence classes they determine. Let `alpha` be a formula characterizing the state space $\mathbb{S}$ (for instance, a class invariant). Assume that each of the above described predicates, corresponding to test data equivalence classes, is captured by a corresponding propositional variable `p_i`. Then, when a satisfying valuation `val` is retrieved by the incremental SAT-solver, it determines a test input, and exactly one `p_i` is true in this valuation (recall that predicates $p_1, \ldots, p_k$ are mutually disjoint). Then, simply by adding to `alpha` the extra clause $\neg$`p_i`, we guarantee that new satisfying valuations will correspond to equivalence classes different from the one captured by `p_i`. Since already covered classes are disregarded by this process, algorithm `IncPartitionCoverage` (Fig. 1) yields exactly one test input per equivalence class. In the algorithm, `rm_val` corresponds to the above described clause, which removes the equivalence class of valuation `val` (i.e., $\neg$`p_i`, where `p_i` characterizes the equivalence class of `val`). The algorithm takes advantage of incremental SAT-solving in order to generate a coverage with respect to the testing criterion characterized by predicates $p_1, \ldots, p_k$.

As we mentioned, the coverage obtained by the algorithm in Fig. 1 is *optimal*: each class coverable within the provided scope is covered by exactly one test input. We will make sure that specializations of the algorithm to a (selected) coverage criterion will preserve this optimality.

### A. Alloy Intermediate Models for Test Input Generation

The inputs to the test generation process are a method to be tested with its corresponding Java class, and a user-selected coverage criterion. From them, FAJITA automatically synthesizes input predicates $IP_1(\text{params}_1), \ldots, IP_n(\text{params}_n)$. For black-box test criteria, these predicates are synthesized from a specification, e.g., from the JML precondition of the method under analysis. For white-box criteria, these predicates are synthesized from the source code, e.g., from branching conditions in the method's source code. Appropriately combined, they give origin to the above mentioned predicates $p_1, \ldots, p_k$ describing the equivalence classes to be covered. For the sake of simplicity in the description, we allow predicates $IP_i$ to be parameterized, although for the definition of the equivalence classes these parameters must be instantiated with constant values (i.e., $\text{params}_1, \ldots, \text{params}_k$ are constants). In Sections IV and V we will describe how these predicates are constructed in order to enforce selected testing criteria. The translation from annotated code to a CNF formula includes, as an intermediate step, the construction of an Alloy model. We only describe here how the Alloy model for test input generation is constructed provided one has already defined the testing criterion's predicates.

```
one sig PredsAndParams {
   this : C,
   ip_1,...,ip_n : boolean,
   params_1 : Types1,..., params_n : Typesn }
fact defPreds {
  PredsAndParams.ip_1=true <=>
    AIP1[PredsAndParams.this,
        PredsAndParams.params_1]
   and ... and
  PredsAndParams.ip_n=true <=>
    AIPn[PredsAndParams.this,
        PredsAndParams.params_n] }
fact InvAndPrecondition {
  I(PredsAndParams.this) and
    mp(PredsAndParams.this) }
```

Fig. 2. Instrumentation of the Alloy model for characterizing a testing criterion's equivalence classes.

FAJITA uses the same translation of JML-annotated Java code presented in [16] in order to arrive to an Alloy model. We instrument this model in order to appropriately capture predicates $IP_1(\text{params}_1)$, ..., $IP_n(\text{params}_n)$, by corresponding Alloy predicates $AIP_1(this, \text{params}_1)$, ..., $AIP_n(this, \text{params}_n)$. The latter are automatically obtained from the JML or Java predicates using the translation described in [16]. From the class invariant and the method precondition, we also obtain Alloy formulas $I$ and $mp$, respectively.

We now describe the fragment of Alloy used in Fig. 2. For a thorough description of Alloy we refer the reader to [18]. Signatures denote sets of objects. The modifier "`one`" constrains signature `PredsAndParams` to have a single object $P$. Then, object $P.this$, which belongs to signature $C$, corresponds to the input datum being generated, and the Boolean values $P.ip\_1, \ldots, P.ip\_n$ will allow us to synthesize the formula `rm_val` we will use in algorithm `IncPartitionCoverage`. In order to relate these boolean variables to the corresponding parameterized predicates, fields $P.params_1, \ldots, P.params_n$ are incorporated in the model to represent the predicate's parameters, and axioms are introduced to relate each Boolean variable $P.ip\_i$ to its corresponding predicate $AIP_i$. Axioms are called *facts* in Alloy. Fact `defPreds` relates, as we described, predicates with boolean variables storing their truth values. Finally, fact `InvAndPrecondition` requires test input candidates

retrieved from this Alloy specification to satisfy the class invariant and the precondition of the method under analysis.

In Sections IV and V we describe how predicates $IP_1(\text{params}_1), \ldots, IP_n(\text{params}_n)$ are defined, and how they relate to predicates $p_1, \ldots, p_k$ in order to capture several black-box and white-box testing criteria. In this way, we show how testing criteria fit the generic scenario proposed in this section, and how algorithm `IncPartitionCoverage` enables us to obtain optimal coverage (again, within the provided scope) according to these testing criteria.

## IV. Incremental Test Input Generation for Black-Box Criteria

In this section we show how the generic process described in Section III can be instantiated for generating test inputs, while at the same time achieving optimal coverage with respect to black-box testing criteria. As mentioned in Section III, any coverage criterion establishes an equivalence between inputs. For instance, for the black-box testing criterion *partition coverage*, the relationship between partitions and the equivalence between inputs we refer to is straightforward. As we will show in this section, this is also the case for other testing criteria.

Let us consider, for instance, *Boolean Query Coverage* [22]. If one needs to test a method whose receiver is an object of class `C`, then one can take into consideration the parameterless Boolean queries of class `C` in order to define a partition on the state space of this class. As an example, consider a class, say, `Stack`, with two parameterless Boolean queries `isEmpty()` and `isFull()`. These two parameterless Boolean queries define four classes, corresponding to the four different combinations of outputs of these queries, and constitute a partition of the state space of any object of type `Stack`. More generally, let us consider a class $C$ with $n$ different parameterless Boolean queries $q_1(X), \ldots, q_n(X)$ (here, $X$ represents the implicit `this` receiver object of these queries). Tuples of the form $\langle q_1(o), \ldots, q_n(o) \rangle$, for $o$ an object of class $C$, induce a partition on the set of possible states of $o$, i.e., on the state space of class $C$. The predicates defined as $p_{l_1,\ldots,l_n}(X) = l_1(X) \wedge \ldots \wedge l_n(X)$, for every possible combination of literals $l_1, \ldots, l_n$ (where each $l_i$ is associated to propositions $\{q_i, \neg q_i\}$), are mutually disjoint. This family of mutually disjoint predicates characterizes the equivalence classes to be covered according to Boolean query coverage. We will use predicates $q_1(X), \ldots, q_n(X)$ as the predicates $IP_1, \ldots, IP_n$ required in the instrumentation of the Alloy model described in Section III-A. In order to instantiate algorithm `IncPartitionCoverage` (Fig. 1), it only remains to show how to define clause `rm_val`, required by this algorithm. Formula `rm_val` is expressed by a single clause $c$ defined as follows:

$$c := \bigvee_{1 \leq i \leq k,\ q_i \ \texttt{s.t.}\ \texttt{val}(q_i) = true} \neg\, q_i \quad \vee \bigvee_{1 \leq j \leq k,\ q_j \ \texttt{s.t.}\ \texttt{val}(q_j) = false} q_j \ . \quad (1)$$

Formula (1) requires that the validity of query $q_i(X)$ be captured by a corresponding Boolean variable $q_i$. This requirement is already fulfilled by fact `defPreds` in the Alloy instrumentation (Fig. 2). In Fig. 3 we show an example of the instantiation of the Alloy model for Boolean query coverage.

```
one sig PredsAndParams {
   this : Stack,
   q_1,q_2 : boolean }
fact defPreds {
  PredsAndParams.q_1=true <=>
    AIP1[PredsAndParams.this]
   and
  PredsAndParams.q_2=true <=>
    AIP2[PredsAndParams.this] }
pred AIP1[s:Stack]{isEmpty[s]}
pred AIP2[s:Stack]{isFull[s]}
fact InvAndPrecondition {
    StackInv(PredsAndParams.this) and
    mp(PredsAndParams.this) }
```

Fig. 3. Boolean query coverage: Alloy model for class `Stack`.

By using formula (1) in the algorithm of Fig. 1, and instantiating the Alloy model (Fig. 2) with the Boolean queries $q_1(X), \ldots, q_n(X)$, we obtain an optimal Boolean query coverage. To prove that we indeed obtain optimal coverage for this criterion, the most critical part is to show that adding clause (1) as the `rm_val` formula in the algorithm exactly removes the equivalence class covered by the test input value `getTestInput(val)`. Let us argue about this fact. Let `val` be a valuation retrieved by `getValuation` during the $j$-th iteration of the loop in algorithm `IncPartitionCoverage`. Due to the Alloy instrumentation, this valuation satisfies exactly one predicate $p_{l_1,\ldots,l_n}$. We will show first that by adding the appropriate instance of clause (1), the class induced by predicate $p_{l_1,\ldots,l_n}$ is indeed removed. Let us call this instance of clause (1), $Rem$. It is straightforward to notice that `val` does not satisfy clause $Rem$. Let us now consider any valuation $\texttt{val}'$ retrieved during an iteration $j'$ ($j' > j$) of the loop. Due to the use of the incremental SAT-solver, $\texttt{val}'$ must satisfy $Rem$:

$$\bigvee_{1 \leq i \leq k,\ q_i \ \texttt{s.t.}\ \texttt{val}(q_i) = true} \neg\, \texttt{val}'(q_i) \quad \vee \bigvee_{1 \leq j \leq k,\ q_j \ \texttt{s.t.}\ \texttt{val}(q_j) = false} \texttt{val}'(q_j) \equiv true.$$

Since $\texttt{val}'$ satisfies $Rem$ and `val` does not, $\texttt{val} \neq \texttt{val}'$. Moreover, since clause $Rem$ only depends on variables $q_1, \ldots, q_n$, there must exist an index $i_0$, with $1 \leq i_0 \leq n$, such that $\texttt{val}(q_{i_0}) \neq \texttt{val}'(q_{i_0})$. Suppose that $\texttt{val}(q_{i_0}) = true$ and $\texttt{val}'(q_{i_0}) = false$. Then, $\texttt{val}'$ cannot satisfy $p_{l_1,\ldots,l_n}$. A similar reasoning can be applied if $i_0$ is such that $\texttt{val}(q_{i_0}) = false$ and $\texttt{val}'(q_{i_0}) = true$.

Finally, let us prove that the addition of clause $Rem$ *only* removes the class induced by predicate $p_{l_1,\ldots,l_n}$. Let $\phi$ be the CNF formula considered by the incremental SAT-solver previous to the addition of clause $Rem$. Let $\varphi = \phi \wedge Rem$. We must show that any valuation $\texttt{val}'$ that satisfies $\phi$ and differs from `val` on variables $q_1, \ldots, q_n$, satisfies $\varphi$.

Let $i_0$ be such that $1 \leq i_0 \leq n$ and $\texttt{val}'(q_{i_0}) \neq \texttt{val}(q_{i_0})$. If $\texttt{val}(q_{i_0}) = true$, then $\neg\texttt{val}'(q_{i_0}) = true$ and $\texttt{val}'$ satisfies $Rem$. Similarly, if $\texttt{val}(q_{i_0}) = false$, then $\texttt{val}'(q_{i_0}) = true$

```
if (Cond1) then {
   {/*$goal 1*/}
   if_body;
} else {
   {/*$goal 2*/}
   else_body; }
assert (Cond2);
{/*$goal 3*/}
```

$\Longrightarrow$

```
PG1 = false;
PG2 = false;
PG3 = false;
if (Cond1) then {
   PG1 = true;
   if_body;
} else {
   PG2 = true;
   else_body; }
assert (Cond2);
PG3 = true;
```

Fig. 4. Code instrumentation for goal coverage.

and `val`′ satisfies $Rem$. Since by hypothesis `val`′ satisfies $\phi$, it satisfies $\varphi$.

## V. Incremental Test Input Generation for White-Box Criteria

We now show how to instantiate the generic methodology of section III in order to obtain optimal coverage according to various white-box testing criteria. As in the case of black-box criteria, the approach consists of appropriately defining equivalence class predicates so that the algorithm presented in Fig. 1 achieves optimal coverage for the criterion under consideration. We show how to define equivalence class predicates corresponding to *goal coverage*, *statement coverage*, *branch coverage*, and *path coverage*.

### A. Handling Goal Coverage

*Goal coverage* is a white-box testing criterion that consists of defining particular points in the code, which should be traversed by tests exercising the code. Goal coverage is supported by the ROOPS (Reachability in Object-Oriented ProgramS) language initiative [24], a common ground for comparing testing tools targeting Java and C# programs.

In order to deal with goal coverage, we instrument the source code by adding a predicate (boolean variable) $PG_i$ for each goal $i$. These predicates are initialized as `false`, and set to `true` in the place in the code where the original goal was placed. Figure 4 shows an example.

In order to achieve goal coverage by employing the generic process described in section III, we need to determine the input predicates and the clauses to be removed. Following the terminology from Section III, we will use the Alloy predicates $AIP_1, \ldots, AIP_n$ defined (using Alloy notation) by

$$\texttt{pred AIP}_i\texttt{[]}\{\texttt{PredsAndParams.PG}_i\texttt{=true}\}$$

The Alloy instrumentation for achieving goal coverage is based on that shown in Fig. 2. We use predicates `PG1`, ..., `PGn` as the predicates $IP_1, \ldots, IP_n$ required in the instrumentation of the Alloy model described in Section III-A. Unlike previous cases, besides the class invariant $I$ and the precondition $mp$, we conjoin in fact `InvAndPreconditionAndCode` the Alloy predicate `Code` obtained as a result of the translation of the method's source code to Alloy [16]. The corresponding Alloy model deviates slightly from that in Fig. 2, becoming the model in Fig. 5. According to this characterization, in the satisfying valuations of the model, the values of `this`, `PG1`, ..., `PGn` are dependent on the executions of the method under test. In particular, `this` corresponds to the method's receiver object, and variables $PG_1, \ldots, PG_n$ store goal reachability information. Notice also that we include fact `SomeGoalMustBeCovered`, which guarantees that the initial call to the SAT-solver will cover at least one goal. In Fig. 6 we show an example where method `push` from `Stack` is assumed to be annotated with three goals. Notice the use of boolean variables `PG1`, `PG2`, and `PG3`, whose value in each valuation is determined by the Alloy encoding of the `push` method code. In fact, Boolean variables $p_1, \ldots, p_n$ are maintained only for presentation purposes, since we can remove them and directly use variables $PG_1, \ldots, PG_n$ instead.

```
one sig PredsAndParams {
   this : C,
   PG1,...,PGn : boolean }
fact defPreds {
    PredsAndParams.PG1=true <=> AIP1[]
    and ... and
    PredsAndParams.PGn=true <=> AIPn[] }
pred AIP1[]{ PredsAndParams.PG1 = true }
...
pred AIPn[]{ PredsAndParams.PGn = true }
fact InvAndPreconditionAndCode {
    I and mp and Code }
fact SomeGoalMustBeCovered {
    PredsAndParams.PG1=true
    or ... or
    PredsAndParams.PGn=true }
```

Fig. 5. Instrumentation of the Alloy model for goal coverage.

```
one sig PredsAndParams {
   this : Stack,
   PG1,PG2,PG3 : boolean }
fact defPreds {
    PredsAndParams.PG1=true <=> AIP1[] and
    PredsAndParams.PG2=true <=> AIP2[] and
    PredsAndParams.PG3=true <=> AIP3[] }
pred AIP1[]{PredsAndParams.PG1 = true}
pred AIP2[]{PredsAndParams.PG2 = true}
pred AIP3[]{PredsAndParams.PG3 = true}
fact InvAndPreconditionAndCode {
    StackInv and Push_Requires and Push_Code }
fact SomeGoalMustBeCovered {
    PredsAndParams.PG1=true or
    PredsAndParams.PG2=true or
    PredsAndParams.PG3=true }
```

Fig. 6. Alloy model for goal coverage: method `push` from class `Stack`.

Now let us place our attention on the `rm_val` clause required in algorithm `IncPartitionCoverage`. On invocation to the incremental SAT-solver, it produces a valuation `val` that is retrieved by subroutine `getValuation`. Clause `rm_val` should be such that, when added to the propositional CNF formula resulting from the translation of the model, removes all further valuations covering the same goals already covered by `val`. Clause `rm_val` is built as follows:

$$\bigvee_{1\leq i\leq n,\ val(\texttt{PG}_i)=false,\ \texttt{Gi not yet covered}} \texttt{PredsAndParams.PG}_\texttt{i}\ . \tag{2}$$

Notice that formula (2) is indeed a clause that can directly be fed to the incremental SAT-solver. Also, according to Alloy's semantics, `pi` has type `PredsAndParams` $\rightarrow$ `boolean`. Therefore, expresion `PredsAndParams.pi` has type `boolean`.

Since objects of type `boolean` are modeled inside the SAT-problem with a single propositional variable, the implementation of FAJITA will use the corresponding propositional variable resulting from the translation of the Alloy model to a propositional formula, rather than using the expression `PredsAndParams.pi`.

Let us argue about the correctness of our characterization of goal coverage, i.e., that under the aforementioned conditions algorithm `IncPartitionCoverage` indeed achieves goal coverage. Let $m(o)$ be a goal annotated routine, whose input is an object $o$ of class $C$. Suppose that we translate $m(o)$ by first instrumenting its code according to Fig. 4, and then instrumenting the Alloy intermediate translation according to Fig. 5. Each time algorithm `incPartitionCoverage` produces a valuation (as a result of calling subroutine `getValuation`), this valuation is such that variable `PredsAndParams.this` covers the class induced by tuple $\langle$`PredsAndParams.PG1`$,\ldots,$`PredsAndParams.PGn`$\rangle$; that is, the value of `PredsAndParams.this` corresponds to an input that reaches exactly those goals whose predicates `PredsAndParams.PG1`, ..., `PredsAndParams.PGn` were assigned the value *true* in the valuation.

Let us consider clauses $c_j$ ($j = 1, \ldots, p-1$), introduced by subroutine `getPredicateFrom` in iterations $1, \ldots, p-1$ of algorithm `incPartitionCoverage`. The valuation $v$ obtained during the $p$-th iteration must satisfy $\bigwedge_{1 \leq j \leq p-1} c_j$. Moreover, by (2), $v$ must cover a class in which at least one variable `PGj` that has not been set to true before, is now set to true. Thus, each iteration covers at least one previously uncovered goal.

It remains to show that each goal reachable within the provided scope (recall that our analysis is limited by a scope, i.e., a user provided limit in the number of objects, and a maximum number of loop iterations, or loop unrolls) is indeed covered, and that no unreachable goal is covered. The latter is easy to guarantee resorting to the correctness of the characterization of Java code given in [14], since valuations must be coherent with method executions, and therefore no execution exists that reaches an unreachable goal.

Let us suppose that the algorithm terminated without covering a goal $G$, reachable within the provided scope. Then, the formula $\alpha$ obtained is unsatisfiable, yet there is a method execution that, within the provided scope, reaches goal $G$. We take this execution and build a valuation, as follows:

- `PredsAndParams.this` = input value that traverses the path.
- `PGi` = *true* if goal $G_i$ is covered, and *false* otherwise.

The resulting propositional valuation $v'$ satisfies the original formula $\varphi$. Also, since goal $G$ has not been so far covered, $v'$ satisfies clauses $c_j$ ($j = 1, \ldots, p-1$), contradicting the unsatisfiability of formula $\alpha$. Thus, every "bounded-reachable" goal is indeed covered by the algorithm.

Finally, notice that each call to the incremental SAT-solver in the goal coverage instantiation of algorithm `incPartitionCoverage` covers at least one goal that was previously not covered. Thus, given a method including $n$ goals, the algorithm terminates after at most $n$ invocations to the SAT-solver.

Goal coverage is a quite versatile testing criterion; other well known white-box testing criteria can be obtained by appropriately introducing goals in the code. For instance, *statement coverage*, which requires the test suite to exercise every executable statement at least once, can be achieved as a particular case of goal coverage. This is obtained by introducing a new goal immediately after each decision occurring in the code.

### B. Handling Branch Coverage

Branch coverage requires the test suite to contain tests that evaluate each executable branching condition in the program under test to *true* and *false*. Each decision point (predicate) *cond* in the program generates two branches, the branch in which *cond* is true (denoted by $B_{+cond}$), and the branch in which *cond* is false (denoted by $B_{-cond}$).

As discussed earlier in this section, in order to make algorithm `incPartitionCoverage` to produce a suite that achieves a test criterion of interest (in this case, branch coverage), we need to appropriately choose equivalence class predicates capturing the criterion. We will obtain branch coverage in our setting by resorting to goals. In effect, adding a goal after each branch implies that any suite that achieves goal coverage also achieves branch coverage of the program under test. Notice that this approach will require twice as many propositional variables (goals) as decision points in the program under test. If we take into account that modern SAT-solvers can handle formulas involving millions of variables, this is a very economical representation.

Since our approach to obtaining branch coverage resorts to the above described goal coverage approach, we only instrument the source code in a different way compared to the instrumentation presented in Section V-A, but we preserve the way in which the intermediate Alloy model is instrumented. The instrumentation of the source code is the following. Whenever an `if-then` statement is found in the code, the `else` branch, as well as appropriate goals, are added:

```
if (P) then {Body;}   ↦   if (P) then{
                            goalP+ = true;
                            Body;
                          } else
                            goalP− = true;
```

Similarly, given a `while` statement, its instrumentation is:

```
while (P) {Body;}   ↦   while (P) {
                          goalP+ = true;
                          Body;}
                        goalP− = true;
```

By following the lines of our reasoning about the correctness of the goal coverage characterization, it is straightforward to see that algorithm `incPartitionCoverage`, adapted as explained before for goal coverage and injecting goals appropriately, produces a test suite that satisfies branch coverage. Moreover, given a method including $k$ conditions, algorithm `incPartitionCoverage` terminates after at most $2k$ incremental invocations to the SAT-solver.

```
one sig PredsAndParams {
   this : C,
   C1,...,Ck : boolean,
   params1 : Types1,..., paramsn : Typesn }
fact defPreds {
   C1=true <=> AC1[PredsAndParams.this,
                   PredsAndParams.params1]
   and ... and
   Ck=true <=> ACk[PredsAndParams.this,
                   PredsAndParams.paramsk] }
fact InvAndPreconditionAndCode {
    I and mp and Code }
pred AC1[c:C, pars1:Types1] {predicate body}
...
pred ACk[c:C, parsk:Typesn] {predicate body}
```

Fig. 7. Instrumentation of the Alloy model for path coverage.

### C. Handling Path Coverage

Path coverage requires the test suite to traverse every executable path in the control flow graph (CFG) of the method under test. In case there are loops in the program, the number of paths is infinite. Notice however that our analysis is *bounded* by a user provided scope, that includes a limit in the number of iterations. This bound is used to unroll loops in the code, making the CFG acyclic, and consequently making the number of paths finite.

As we did for branch coverage, we can use the process for obtaining a test suite complying with goal coverage presented in Section V-A in order to get a suite complying with path coverage. In order to do this we will appropriately inject goals in the program under test. The goals we inject in the code are exactly the same as for the case of branch coverage. Then, we instrument the Alloy model with a slight variant of the processes defined previously in this section. This instrumentation is shown in Fig. 7. Predicates $AC_i$ are the Alloy versions automatically obtained from method decision points. For instance, consider a decision $C_1 = A[i] <= j$ where $A$ is an array and $i$ is an integer variable, both local to the method under test, and $j$ is an integer-typed parameter of the method. Part of the instrumentation of the Alloy model, according to what is described in Fig. 7, would look for this case as follows:

```
one sig PredsAndParams {
   this : C,
   j : integer,
   A : array,
   i : integer,
   C1 : boolean }
fact defPreds {
  PredsAndParams.C1=true <=>
     AC1[PredsAndParams.A, PredsAndParams.i,
        PredsAndParams.j] }
pred AC1[A:Array, i,j:integer]{
   lte[(PredsAndParams.A)[PredsAndParams.i],
       PredsAndParams.j] }
```

Notice that program condition $C_1$ (whose value is stored in a Boolean variable `C1`) becomes Alloy predicate $AC1$.

Let us now define the clause to be added when calling method `getPredicateFrom` in algorithm `incPartitionCoverage`, so that no further inputs are produced for already covered (bounded) paths. When executing the method on value `PredsAndParams.this`, some branches associated to decisions $C_{i_1}, \ldots, C_{i_j}$ are covered. By adding the clause

$$c := \bigvee_{1 \leq k \leq j, v(C_{i_k}) = true} \texttt{!PredsAndParams.Cik} \vee \bigvee_{1 \leq k \leq j, v(C_{i_k}) = false} \texttt{PredsAndParams.Cik} \; . \quad (3)$$

the equivalence class of inputs corresponding to the path covered by `PredsAndParams.this` is "marked" as covered, and algorithm `incPartitionCoverage` will disregard this equivalence class in further generations of inputs.

Again, by following the lines of our argument on the correctness of the goal coverage characterization, it is straightforward to show that, by instantiating algorithm `incPartitionCoverage` as just described, a path coverage is obtained. Moreover, given a bound $l$ on loops, and a method containing $k$ feasible paths with loops bounded by $l$, algorithm `IncPartitionCoverage` terminates after at most $k + 1$ invocations to the SAT-Solver.

## VI. Evaluation

We now present an evaluation of our test input generation mechanism. The section is organized as follows. In Section VI-A we describe the hardware and software platforms used for the experiments. In Section VI-B we present experiments showing how FAJITA compares to other state-of-the-art tools for test input generation. Finally, in Section VI-C we discuss the validity and relevance of the experiments performed.

### A. The Hardware and Software Platforms

All experiments were run on identical hardware, featuring an Intel Core i5-750 processor running at 2.67 GHz, Intel DP55WB motherboard and 8 GB 1333 MHz DDR3 total main memory. The operating system was Windows 7 Professional for the Pex tool (which does not run under Linux), and Debian's GNU/Linux (version 6, "squeeze") in all other cases.

### B. Experimental Results

In this section we compare FAJITA with some closely related tools, namely:

**Pex [27]:** Microsoft's Pex uses dynamic symbolic execution. It combines code execution with static analysis techniques, and uses the Z3 SMT-Solver [7] in order to solve constraints arising from path conditions.

**Kiasan [8]:** Kiasan uses symbolic model checking with lazy initialization as the underlying technology for test input generation, and uses the Yices SMT-Solver [11] for constraint solving. It works on a JML annotated method, but since the JML constructs supported by Kiasan are not sufficiently expressive to specify some of the complex class invariants we will use in our benchmark classes, we provided appropriate repOk methods (imperative class/object invariants).

**Randoop [23]:** Randoop is a tool for random generation of unit tests. It uses, as the underlying technique, feedback-directed random test generation.

**Autotest [21]:** Autotest is a fully automatic testing framework integrated inside EiffelStudio that uses random generation.
**EvoSuite [12]:** EvoSuite imitates the mechanisms of natural adaptation by evolving whole test suites. Starting from a random population, evolution is performed until a solution is found, or the allocated resources are exhausted.

In all FAJITA experiments we initially set the number of loop unrolls and data domain scopes to 1. FAJITA iteratively increases both bounds until either optimal coverage is achieved, 5 minutes pass without new tests being generated (in which case termination is enforced), or a provided timeout is reached. We make use of symmetry breaking and tight bounds as described in Section II. As we mentioned, a cluster is employed in order to pre-compute tight bounds for the structures used in the experiments. This computation is reused for different methods and coverage criteria involving the same structure. Although we do not include here the tight bound computation time, it is worth mentioning that it is 70 seconds in average for our experiments. The specific times for each structure and scope can be found in [16].

For those tools that include parameters that are set manually, we looked for the combinations that led to the best coverage. For instance, in Kiasan, we looked for the best combination of k-bound, loopbound and callbound. We only report the time of the most successful experiment. Since FAJITA is push-button, we report the result it produces in the first execution.

Randoop, Autotest and EvoSuite use random seeds. For these tools, we ran each experiment 10 times and reported the average time amongst those runs that achieved maximum coverage. For all tools we set a 1 hour timeout (indicated as TO when reached).

We considered the following container classes adopted from the ROOPS benchmark, aiming at achieving high goal and branch coverage:
`SinglyLinkedList`: An implementation of sequences based on singly-linked lists.
`DoublyLinkedList`: The implementation provided in class `AbstractLinkedList` of interface `List` from the Apache package `commons.collections`, based on circular doubly-linked lists.
`NodeCachingLinkedList`: A caching, circular doubly linked list, implementing interface `List` from the Apache package `commons.collections`.
`BinarySearchTree`: An implementation of binary search trees used as part of a benchmark in [29].
`AVLTree`: An implementation of AVL trees obtained from the case study used in [2].
`BinomialHeap`: An implementation of binomial heaps used as part of a benchmark in [29].
`FibonacciHeap`: An implementation of Fibonacci heaps used as part of a benchmark in [29].
`IntRedBlackTreeMap`: A map implementation part of the ROOPS repository, that follows the specification for the `java.util.TreeMap` class, but using primitive values of type `int` as keys.

The results of these experiments are summarized in Table II. Data in the table is reported as follows. Columnn `#B` reports the number of *branches* in each method. We report for each tool and experiment the percentage of branches covered and the required time. We highlighted those positions in the table where a suboptimal performance regarding coverage was detected. Due to space limitations, for goal coverage we only report the summary presented in Table I. As shown in Table II, FAJITA is the tool that achieved the best branch coverage on average in the 25 methods under analysis, and the same is true for goal coverage (see Table I). Also, FAJITA produced the smallest test suites.

Since EvoSuite measures branch coverage at the byte code level, the tool finishes when every Java conditional jump instruction is covered both ways. Because of this we were unable to measure the branch coverage at the source code level until the timeout was reached.

Kiasan failed to achieve maximum goal and branch coverage in 5 methods, and Pex failed in 14 methods. As expected, complex class invariants require loops in their imperative descriptions that lead to a path explosion problem. As explained before, FAJITA performs a bounded analysis and the scope (limit on the number of objects and iterations) is iteratively increased. This helps in tackling path explosion, since the size of inputs and length of runs is increased as the search for optimal coverage requires it.

Randoop and AutoTest reach the timeout in all experiments. This is because they do not allow one to specify that analysis must stop when all goals are reached; the termination conditions that can be selected are either a maximum generation time, or a maximum number of generated tests.

For method `setMaxCacheSize` from class `NodeCachingLinkedList`, Pex and Kiasan reported covering 5 goals, while FAJITA covered 4. A manual inspection of the code of this method enabled us to confirm that the goal that FAJITA did not reach is located inside an infinite loop; that is, any input covering this goal leads to a non-terminating execution. Similarly, Pex and Randoop cover a branch that leads to an infinite execution. The translation from code to a propositional formula implemented in FAJITA explicitly prevents the generation of such test inputs, which guarantees that all the generated tests can be executed without the need of human intervention to discard/stop non-terminating tests.

### C. Threats to Validity

We used several container classes as a benchmark. Container classes have become ubiquitous [29], and are representatives of a wider class of programs that include, for instance, parse trees and XML documents. Moreover, a number of analysis tools have used these classes in highly regarded software analysis conference papers as well (see for instance [3], [9], [19], [25], [26], [29]). Container classes are also good examples of code in which strong heap properties must be enforced.

FAJITA uses a class invariant specified as a JML-formula extended with reachability constructs. In Pex and Kiasan we used `repOk` predicate methods ruling out those structures that did not satisfy the class invariant. To minimize the risk of programming not amenable `repOk` methods we used minor modifications of the `repOk` methods provided for Korat [3]. In

| | #B | Pex | Kiasan | Randoop | AutoTest | EvoSuite | FAJITA |
|---|---|---|---|---|---|---|---|
| `SinglyLL` | | | | | | | |
| contains | 12 | 25% (7s) | 100% (4s) | 58% (TO) | 92% (TO) | 92% (8s) | 100% (4s) |
| insertBack | 6 | 50% (1s) | 100% (2s) | 100% (TO) | 100% (TO) | 100% (8s) | 100% (3s) |
| remove | 10 | 40% (1s) | 100% (3s) | 100% (TO) | 100% (TO) | 100% (8s) | 100% (8s) |
| `DoublyLL` | | | | | | | |
| contains | 10 | 10% (6s) | 100% (3s) | 90% (TO) | 100% (TO) | 90% (TO) | 100% (6s) |
| addLast | 2 | 50% (9s) | 100% (3s) | 100% (TO) | 100% (TO) | 100% (TO) | 100% (4s) |
| removeIndex | 14 | 93% (6s) | 71% (3s) | 100% (TO) | 79% (TO) | 100% (TO) | 100% (13s) |
| `NodeCachingLL` | | | | | | | |
| contains | 10 | 30% (1s) | 100% (9s) | 70% (TO) | 100% (TO) | 100% (TO) | 100% (7s) |
| setMaxCacheSize | 8 | 100% (9s) | 88% (4s) | 100% (TO) | 38% (TO) | 88% (TO) | 88% (5m7s) |
| addLast | 6 | 100% (12s) | 100% (4s) | 100% (TO) | 100% (TO) | 100% (TO) | 100% (5s) |
| removeIndex | 20 | 35% (9s) | 25% (3s) | 90% (TO) | 60% (TO) | 90% (TO) | 95% (7m26s) |
| `SearchTree` | | | | | | | |
| find | 8 | 100% (1s) | 100% (3s) | 36% (TO) | 38% (TO) | 100% (TO) | 100% (4s) |
| add | 12 | 100% (4s) | 100% (3s) | 25% (TO) | 25% (TO) | 100% (TO) | 100% (6s) |
| remove | 20 | 85% (58m35s) | 85% (6s) | 10% (TO) | 30% (TO) | 85% (TO) | 85% (5m9s) |
| `AVLTree` | | | | | | | |
| findNode | 8 | 100% (5s) | 100% (3s) | 36% (TO) | 75% (TO) | 36% (TO) | 100% (4s) |
| findMax | 6 | 83% (3m36s) | 100% (3s) | 50% (TO) | 100% (TO) | 50% (TO) | 100% (6s) |
| findMin | 6 | 83% (1m33s) | 100% (3s) | 50% (TO) | 100% (TO) | 50% (TO) | 100% (6s) |
| `BinHeap` | | | | | | | |
| findMin | 6 | 83% (5s) | 100% (3s) | 100% (TO) | 100% (TO) | 100% (TO) | 100% (10s) |
| decreaseKey | 6 | 100% (6s) | 100% (3s) | 67% (TO) | 83% (TO) | 100% (TO) | 100% (7s) |
| extractMin | 46 | 35% (8s) | 100% (6s) | 74% (TO) | 52% (TO) | 100% (TO) | 100% (1m47s) |
| insert | 28 | 82% (TO) | 82% (TO) | 82% (TO) | 82% (TO) | 82% (TO) | 100% (3m41s) |
| `FibHeap` | | | | | | | |
| minimum | 2 | 100% (1s) | 100% (3s) | 100% (TO) | 100% (TO) | 100% (TO) | 100% (3s) |
| insertNode | 6 | 100% (4s) | 100% (3s) | 33% (TO) | 33% (TO) | 100% (TO) | 100% (6s) |
| removeMin | 38 | 87% (TO) | 87% (4s) | 8% (TO) | 11% (TO) | 87% (TO) | 87% (6m4s) |
| `IntRBTreeMap` | | | | | | | |
| put | 38 | 100% (3s) | 89% (TO) | 86% (TO) | 42% (TO) | 100% (TO) | 100% (42s) |
| remove | 66 | 98% (2m27s) | 81% (14m35s) | 71% (TO) | 23% (TO) | 95% (TO) | 94% (6m10s) |
| Average | – | 74,76% | 92,32% | 69,44% | 70,52% | 89,80% | 97,96% |

TABLE II
BRANCH COVERAGE FOR CONTAINER CLASSES

general, the modifications were required to match field names, or to improve the performance of Pex and Kiasan. For instance, for `AVLTree` we replaced the iterative method `balanced` (which checks the balance criterion) with a recursive version. This improved Kiasan's performance significantly. These `repOK` methods were also used to help Randoop and AutoTest in their random generation processes (both tools benefit from the representation invariant during the generation). The conversion from Java code to C# (input language for Pex) and Eiffel (input language for Autotest) was done manually. Although we did not prove the correctness of the translation, the analysis results are consistent throughout all programming languages.

## VII. RELATED WORK

Automated test case generation is currently a very prolific area of research, and various tools and techniques have been developed in recent years. Among these approaches, one might cite those based on random generation [22], model checking [29], constraint solving (including SMT [27] and SAT solving) or some forms of exhaustive search [3].

In [25] a set of experiments compare random testing and shape abstraction for container classes. In this work we use two random based tools (namely, Randoop and AutoTest). Our results show that incremental SAT-based test input generation, plus our symmetry breaking scheme, outperform branch coverage in most of the cases.

Incremental SAT has been used in other contexts [6] for generating combinatorial interaction tests for product families.

While most tools mentioned in this section generate a single test suite whose quality can be measured against coverage criteria, FAJITA's goal-oriented use of incremental SAT drives the generation process towards achieving high coverage.

Other tools also use the Alloy tool-set as part of the test generation process. TestEra [13] uses incremental SAT-solving for exhaustive bounded generation of input data (a purely black-box criterion). If more than one test input per equivalence class is required, then FAJITA can make use of TestEra's enumeration process. Whispec [26] builds on specification-based testing and focuses on maximizing code coverage by iteratively running a conjunction of method preconditions and path conditions. TestEra and Whispec, unlike FAJITA, provide a single test generation mechanism. Whispec is not publicly available for experimental comparison with FAJITA. In [14] some of the authors of the current article presented a previous approach for white-box test input generation for Java programs called JAT. Coverage criteria were modeled by appropriately slicing the control flow graph. FAJITA can be considered as a successor of JAT that provides a much simpler extension

mechanism, and is substantially more efficient.

One can also compare FAJITA and TACO. While TACO provides greater quality assurance, FAJITA scales better. Each time a test is generated, part of the state space (the part that corresponds to the equivalence class just covered) is pruned by the incremental SAT-solver. As a simple example illustrating the difference in scalability between FAJITA and TACO, consider method `extractMin` from class `BinomialHeap`. This method contains a fault reported in [16], and it takes TACO 43m to detect the fault, while FAJITA achieves branch coverage for this method in less than 2m.

## VIII. Conclusions and Further Work

Test input generation tools based on model checking, symbolic or concolic execution are subject to path explosion. Therefore, significant effort is put towards efficiently traversing the state space induced by the program structure. These techniques are somewhat oblivious to the coverage criteria. While the abstracted code is executed they gather information from which test inputs are generated. FAJITA is *flexible*, and targets specific coverage criteria. The selection of the input predicates for the SAT-solver, as well as the clauses added to advance the search in the incremental steps, make the solver *target a coverage criterion of interest*. FAJITA proved to be particularly successful when used for generating test input data that satisfies strong heap invariants and is aimed at reaching intricate code fragments. We have carried our experiments with container classes, in which FAJITA in several occasions outperformed state of the art tools such as Pex and Kiasan.

Fully automated analysis techniques have scalability issues. In the case of FAJITA, the scalability is related to the program under test and its specification, which given a scope are encoded as a boolean formula. Complex programs and specifications may make the analysis feasible only for very small scopes. The techniques underlying FAJITA, namely tight bounds with symmetry breaking, scope and loop unrolls iteratively increased to reduce formula and path explosions, and analysis targeting a specific coverage criterion, all improve significantly the scalability of the SAT based test generation, and are essential for the performance of the tool. Nevertheless, many other problems still need to be tackled, to make the tool scale further, and be useful for larger programs. Some approaches we plan to work on in the future include better ways of computing tight bounds, dealing with interprocedural code modularly (as opposed to the code inlining that FAJITA performs), and working on appropriate parallelizations of the the SAT problems resulting from FAJITA analyses. Further experimental assessment, which we also plan to do, is necessary to compare FAJITA with other techniques, analyze the bug finding effectiveness of suites computed by the tool, and evaluate how our tool performs on medium sized programs other than container classes.

## Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. This work was partially supported by the Argentinian Agency for Scientific and Technological Promotion (ANPCyT), through grants PICT PAE 2007 No. 2772 and PICT 2010 No. 1690, and by the MEALS project (EU FP7 programme, grant agreement No. 295261).

## References

[1] C. Barrett, R. Sebastiani, S. Seshia, C. Tinelli, *Satisfiability Modulo Theories*, Handbook of Satisfiability, IOS Press, 2009.
[2] J. Belt, Robby, X. Deng, *Sireum/Topi LDP: A Lightweight Semi-Decision Procedure for Optimizing Symbolic Execution-based Analyses*, in Proc. of ESEC/FSE 2009, ACM Press, 2009.
[3] Boyapati C., Khurshid S., Marinov D., *Korat: automated testing based on Java predicates*, in Proc. of ISSTA 2002, ACM Press, 2002.
[4] P. Chalin, J. Kiniry, G. Leavens, E. Poll, *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*, in Proc. of FMCO 2005, LNCS 4111, Springer, 2006.
[5] J. Crawford, M. Ginsberg, E. Luks, A. Roy, *Symmetry-breaking predicates for search problems*, in Proc. of KR '96, Morgan Kaufmann, 1996.
[6] M. Cohen, M. Dwyer, J. Shi, *Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach*, Trans. Software Eng. 34(5), IEEE Press, 2008.
[7] L. de Moura, N. Bjorner, *Z3: An Efficient SMT Solver*, in Proc. of TACAS 2008, LNCS 4963, Springer, 2008.
[8] X. Deng, Robby, J. Hatcliff, *Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-oriented Systems*, in Proc. of MUTATION 2007,IEEE CS, 2007.
[9] G. Dennis, F. Chang, D. Jackson, *Modular Verification of Code with SAT*, in Proc. of ISSTA 2006, ACM Press, 2006.
[10] J. Dolby, M. Vaziri, F. Tip, *Finding Bugs Efficiently with a SAT Solver*, in Proc. of ESEC/FSE 2007, ACM Press, 2007.
[11] B. Dutertre, L. de Moura, *The YICES SMT Solver*, 2006.
[12] G. Fraser, A. Arcuri, *EvoSuite: automatic test suite generation for object-oriented software*, in Proc. of ESEC/FSE 2011, ACM Press, 2011.
[13] S. Khurshid, D. Marinov, *TestEra: Specification-Based Testing of Java Programs Using SAT*, Autom. Soft. Eng. 11(4), Kluwer Academic, 2004.
[14] J. Galeotti, M. Frias, *DynAlloy as a Formal Method for the Analysis of Java Programs*, in Proc. of SET 2006, IFIP 227, Springer, 2007.
[15] M. Frias, J. Galeotti, C. López Pombo, N. Aguirre, *DynAlloy: Upgrading Alloy with Actions*, in Proc. of ICSE 2005, ACM Press, 2005.
[16] J. Galeotti, N. Rosner, C. López Pombo, M. Frias, *Analysis of invariants for efficient bounded verification*, in Proc. of ISSTA 2010, ACM Press, 2010.
[17] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak,D. Marinov, *Test Generation through Programming in UDITA*, in Proc. of ICSE 2010, ACM Press, 2010.
[18] D. Jackson, *Software Abstractions*, MIT Press, 2006.
[19] D. Jackson, M. Vaziri, *Finding bugs with a constraint solver*, in Proc. of ISSTA 2000, ACM Press, 2000.
[20] C. Kaner, J. Bach, B. Pettichord, *Lessons Learned in Software Testing*, Wiley, 2001.
[21] A. Leitner, I. Ciupa, B. Meyer, M. Howard, *Reconciling Manual and Automated Testing: the AutoTest Experience*, in Proc. of HICSS 2007, IEEE CS, 2007.
[22] L. Liu , B. Meyer, B. Schoeller, *Using contracts and boolean queries to improve the quality of automatic test generation*, in Proc. of TAP 2007, LNCS 4454, Springer, 2007.
[23] C. Pacheco, S. Lahiri, M. Ernst, T. Ball, *Feedback-Directed Random Test Generation*, in Proc. of ICSE 2007, IEEE CS, 2007.
[24] Roops Benchmark: `http://code.google.com/p/roops/`
[25] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, D. Marinov, *Testing container classes: random or systematic?*, in Proc. of FASE 2011, LNCS 6603, Springer, 2011.
[26] D. Shao, S. Khurshid, D. Perry, *Whispec: white-box testing of libraries using declarative specifications*, in Proc. of LCSD 2007, ACM Press, 2007.
[27] N. Tillmann, J. de Halleux, *Pex: White Box Test Generation for .NET*, in Proc. of TAP 2008, LNCS 4966, Springer, 2008.
[28] E. Torlak, D. Jackson, *Kodkod: A Relational Model Finder*, in Proc. of TACAS 2007, LNCS 4425, Springer, 2007.
[29] W. Visser, C. Păsăreanu, R. Pelánek, *Test Input Generation for Java Containers using State Matching*, in Proc. of ISSTA 2006, ACM Press, 2006.
[30] E. Weyuker, B. Jeng, *Analyzing Partition Testing Strategies*, Trans. Software Eng. 17(7), IEEE Press, 1991.
[31] H. Zhu, P. Hall, J. May, *Software Unit Test Coverage and Adequacy*, Computing Surveys 29(4), ACM Press, 1997.