The IntiSa approach: Test Input Data Generation for Non-Primitive Data Types by means of SMT solver based Bounded Model Checking

Stefan J. Galler, Thomas Quaritsch, Martin Weiglhofer and Franz Wotawa
Graz University of Technology
Institute for Software Technology
Graz, Austria
{galler, quaritsch, weiglhofer, wotawa}@ist.tugraz.at

***Abstract*—In this paper we present an approach for automatically deriving test input data from Design by Contract™ specifications. Preconditions of a method under test (MUT) require specific object states of the input parameters. In this paper we present INTISA, a novel approach, which calculates test input values to be used with mock objects. The calculated values do not only satisfy the precondition of the method under test, but are guaranteed to be states that could be reached through method calls on the object as well. This property is not supported by previous work, such as random or pure SMT based approaches, and genetic algorithm. But it is important since it reduces false positive test cases. Based on the idea of bounded-model checking, INTISA encodes possible state space changes of method calls for all parameters of the MUT. This model is then verified against an adopted precondition of the MUT. Besides a detailed discussion of the INTISA approach we present results obtained by applying our technique to two case studies, one of them a real world application provided by our industry partner. First results show that INTISA is twice as fast as random in generating primitive and non-primitive test input that satisfies the precondition of the method under test.**

***Keywords*-software testing; test data generation; design by contract; SMT solver; bounded model checking**

I. INTRODUCTION

Testing is a time consuming, tedious and error-prone task. Consequently, testing is often neglected. In industrial-sized programs the creation of test cases targeting a single method is already a complex task. Especially, if the method takes non-primitive types (i.e. objects) as arguments. Usually, a method requires that the arguments, i.e. the objects, are in a specific state. Thus, the creation of test cases includes the task of setting up the parameters for the method under test. Basically, there are two ways in dealing with the complexity introduced by non-primitive type arguments. First, one can instantiate the real object and establish the object state by calling methods of the object. Second, one can use Mock objects [1] in order to mock the behavior of the original object.

However, in the former case one needs to find a method sequence, such that the required object state is established. In the latter case one needs to find values that are returned by the Mock objects. Both tasks are not trivial for complex objects because the method sequences are usually very long and minor modifications of the sequence lead to completely different object states. Note that the second problem is related to the first one. The configuration of the Mock object needs to be a valid object state, i.e. this state has to be reachable by means of method calls on the real object. Otherwise, the inconsistent Mock object state may cause test cases to fail while there is no bug in the method under test [2].

In this paper we present an automated approach, which we call INTISA, that solves these two problems. INTISA relies on Design by Contract™ specifications [3] to automatically derive test input data for a method under test. Design by Contract™ adds semantic information to class and method declarations. By means of Design by Contract™ a programmer can specify what a method requires from its caller and what it guarantees after execution.

INTISA uses this specification to derive test input data. The generated input data satisfies the precondition of the method under test. Furthermore, INTISA ensures that the generated input data represent a reachable object state within the program. As a side-product INTISA generates the method sequence that is required to bring the real object into this state. INTISA encodes possible state space changes of method calls for all parameters of the MUT. By means of an SMT (Satisfiability Modulo Theories)-solver this model is then verified against an adopted precondition of the MUT. In that way the SMT solver returns a valid object state and the methods to be called in order to reach this state.

This paper continues as follows. First we introduce a running example in Section II, which is used through out the paper to explain INTISA. Preliminaries required to understand INTISA are shortly explained in Section III. The introduction of INTISA and its empirical evaluation follows in Sections IV and V, respectively. Before the paper concludes in Section VII related work is mentioned in Section VI.

```
1  @Model(name="mSize", type="int")
2  class Stack {
3    @Post("mSize=@Old(mSize)+1")
4    void push(double elem) { ... }
5
6    @Pre("size()>0")
7    @Post("size()=@Old(size())-1")
8    void pop() { ... }
9
10   @Post("@Return=mSize␣&&␣mSize␣>=␣0")
11   @Pure int size() { ... }
12
13   @Post("@Return=(mSize=0)")
14   @Pure bool empty() {...}
15
16   @Pre("size()>0")
17   @Pure double peek() {...}
18 }
19
20 class SummationOperator {
21   @Pre("stack.size()>=2")
22   @Post("stack.size()=1")
23   void sumUp(Stack stack) {
24     double result = 0;
25     while(!stack.empty()) {
26       result += stack.peek();
27       stack.pop();
28     }
29     stack.push(result);
30 }}
```

Figure 1. Running example, with *sumUp* as method under test, which requires the *Stack* object to be in a specific state.

II. Running Example

We use the example depicted in Figure 1 for explaining our IntiSa approach. This example is a simplification of a small part of the source code of the *StackCalc* case study used in Section V.

For our example we consider the *sumUp()* method at Line 23 to be the method under test (*MUT*). It calculates the sum of all elements on the stack. The method may only be called if there are two or more elements on the stack (Line 21).

Throughout the paper the given examples show the different steps needed to automatically derive an instantiation of the *Stack* class (Lines 2-18) satisfying a given constraint. Note that the specification of it is tailored for this paper to let us show all important steps of IntiSa. For example, we skip the so-called represents clause which links the model field ($mSize$) with the real implementation for sake of simplicity. While this clause is needed to make a specification executable, it has no use at all for our approach.

The syntax of the annotations is based on *Modern Jass* [4] but the specification throughout the paper is written in logic notation, i.e. the single equals character (=) states equality.

III. Preliminaries

In this section we briefly discuss necessary preliminaries to make the presentation of our IntiSa approach self-contained. Therefore, we introduce the concept of Design by Contract™ and show how we use relations to formally reason about Design by Contract™ specifications. Further, a short introduction of the concept of bounded model checking with SMT solvers is given.

A. Design by Contract™

Design by Contract™ adds semantic information to class and method declarations. Similar to legal contracts between two partners, a programmer can specify what a method requires from its caller and what it guarantees after execution. Depending on the programming language, Design by Contract™ specifications are either built-in, such as in Eiffel [5], or can be added through third party libraries, such as *JML* [6] and *Modern Jass* [4] for Java.

The three main concepts of Design by Contract™ are [5]:

1) *precondition*: The precondition P is evaluated before the method is called. It specifies the expected value range of the method's parameters. The method is only executed when the caller satisfies the precondition.
2) *postcondition*: The postcondition Q is evaluated directly after the method body is executed and before returning to the caller. It specifies the guaranteed behavior of the method. Whenever the client satisfies the precondition the method has to satisfy the postcondition, i.e. $P \implies Q$.
3) *class invariant*: A class invariant specifies a property that applies for all instances of the class through out their lifetime. All class invariants are checked before and after a method execution. The invariant may be violated while executing a method's body.

Three other features of Design by Contract™ are important for the presented approach: pure methods, model fields and pre-state access.

Pure methods are side effect free methods, with respect to the object/program state [6]. They may return a value but do not assign values to member variables. Only pure methods are allowed to be called within a Design by Contract™ specification.

Example 1. A typical example for a pure method is the *size()* method (Line 11) of the *Stack* object in Figure 1. It lets the client observe a specific attribute of the object state, but does not change it. Therefore, *size()* can be used within the postcondition of *pop()* (Line 7). □

Cheon et al. [7] introduced the concept of model fields. They define a model field as a "specification-only field that describes the abstract state of some program fields." It is similar to common class fields, but can only be used for specification purposes.

In postconditions it is helpful to be able to express changes of the object state. Hence, it is important to access the value of a variable before the method execution. This is called pre-state access and is typically implemented by means of an *@Old* keyword.

Example 2. For example, the *push()* method's postcondition `size()=@Old(size())+1` in Figure 1 specifies that the value returned by *size()* after method execution will be increased by one with respect to the value returned before method execution. □

Commonly, Design by Contract™ has special semantics in the case of automated test case generation. Given a precondition P and postcondition Q for a method the usual semantics says, if the precondition is not satisfied, then there is no constraint on the methods behavior: $P \implies Q$ However, for automated testing the postcondition Q is used as test oracle. In other words, if the result of the method execution does not satisfy Q, then the test verdict is *fail*. As a consequence, test input that does not fulfill the precondition is discarded as meaningless. Thus, we have $(P \implies Q) \wedge P$ which reduces to $P \wedge Q$. This formula shows that, in the case of automated testing, the precondition P is used as a guard rather than as the antecedence. Furthermore, some Design by Contract™ specification languages support multiple pre-/postcondition pairs per method. This allows one to specify multiple behaviors for one and the same method.

Definition 1 (method behavior): A method behavior $MB = \langle \langle P_1, Q_1 \rangle, \ldots, \langle P_i, Q_i \rangle \rangle$ is a set of tuples, each consisting of a precondition P and a postcondition Q. The logical interpretation of a method behavior is given by

$$MB =_{df} (P_1 \wedge Q_1) \vee \cdots \vee (P_n \wedge Q_n)$$

where all variables in P are unprimed, thus talking about the pre-state.

B. Classes and Their Virtual State

For the discussion of the IntiSa approach we need to formally reason about classes. Therefore, we use the following updated definition of a class [8]:

Definition 2 (Class): A class C is a triple $\langle c, f, m \rangle$ where c denotes the non-empty set of constructors represented by the m-tuple $\langle cid, p_1, \ldots, p_n \rangle$, $f = f^{pub} \cup f^{prot} \cup f^{priv}$ is the union of public (f^{pub}), protected (f^{prot}), and private (f^{priv}) fields, and m the set of methods represented by the n-tuple $\langle mid, vis, ret, p_1, \ldots p_m \rangle$ where mid is a unique identifier, $vis \in \{public, protected, private\}$ defines the visibility of the method, and $ret \in \{void, nvoid\}$ determines the abstracted return type of the method, where $nvoid \in \{prim, nprim\}$ distinguishes between primitive and non-primitive return type. p_i in the definition of c and m determines the i-th parameter type. We introduce the short notation m^{vis}_{ret} to reference the set of all methods in m that satisfy the two given attributes.

Furthermore, we use the following definition to denote the Design by Contract™ specification of a particular class.

Definition 3 (Contract): Given a class $C = \langle c, f, m \rangle$ then the Design by Contract™ specification of this class is denoted by $DbC(C) = \langle mf, inv, pp \rangle$ where mf is a set of model fields, inv is a set of invariants, and pp is a set of sets where each $p_{m_i} \in pp$, denoted as method behavior, is a set of pre-/postcondition pairs for a method $m_i \in m$.

The *state* of a class implementation is given by the values of all it's members. As the IntiSa approach abstracts from a class' implementation, it relies on the virtual state of a class. The *virtual state* is given by all publicly observable elements, i.e. public members, public methods with return values and model fields of the specification. More formally,

Definition 4 (Virtual State): Given a class $C = \langle c, f^{pub} \cup f^{prot} \cup f^{priv}, m^{priv}_{void} \cup \cdots \cup m^{pub}_{nprim} \rangle$ and its contract $DbC(C) = \langle mf, inv, pp \rangle$, then the virtual state of the class C, denoted by $VS(C)$, is given by $VS(C) = f^{pub} \cup m^{pub}_{nvoid} \cup mf$.

Example 3. Consider again the *Stack* of Figure 1 where $f^{pub} = \{\}$, $m^{pub}_{nvoid} = \{size, empty, peek\}$, and $mf = \{mSize\}$, thus the virtual state of the class *Stack* is given by $VS(Stack) = \{size, empty, peek, mSize\}$. Note that $empty$ is a boolean variable, $size$ and $mSize$ are integer values while the domain of $peek$ is double. □

As in our previous work [8] our approach heavily relies on the concept of sequential composition defined in the Unifying Theories of Programming (UTP) [9]. The sequential composition of two program statements expresses that the program starts with the first statement and after successful termination the second statement is executed. The final state of the first statement serves as initial state of the second statement, however, this intermediate state cannot be observed. All this is formalized in UTP using existential quantification.

Definition 5 (Sequential Composition): Two program statements P and Q are sequentially composed by

$$P(v, v'); Q(v, v') \quad =_{df} \quad \exists v_0 \bullet P(v, v_0) \wedge Q(v_0, v')$$

Note that all variables in preconditions are given as unprimed variables and therefore describe the initial state, whereas all variables in the postconditions are given as primed variables and describe the final state. By the use of the *@Old*-keyword one can refer to the initial state values within the postcondition.

Example 4. Consider the *push* method of the Stack from Figure 1. Its method behavior is given by $MB_{push} = true \wedge mSize' = mSize + 1$. For example, calling push twice on the Stack would be modeled as sequential composition of *push*s method behavior.

$$MB_{push}(mSize, mSize'); MB_{push}(mSize, mSize') =$$
$$\exists mSize_0 \bullet true \wedge mSize_0 = mSize + 1 \wedge$$
$$true \wedge mSize' = mSize_0 + 1$$

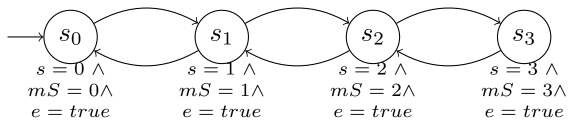
s_0
s_1
s_2
s_3
$s = 0 \wedge$ $mS = 0\wedge$ $e = true$
$s = 1 \wedge$ $mS = 1\wedge$ $e = true$
$s = 2 \wedge$ $mS = 2\wedge$ $e = true$
$s = 3 \wedge$ $mS = 3\wedge$ $e = true$

Figure 2. A simple Kripke structure.

□

C. Bounded Model Checking via SMT Solver

Model-Checking is a technique to verify if a finite state machine model satisfies a specification. The specification is usually given in terms of a temporal logic (e.g. LTL or CTL). If the model does not satisfy the given specification, a counterexample is generated.

For model-checking a finite state machine is represented by a *Kripke structure*.

Definition 6 (Kripke Structure): A Kripke structure M is a tuple $M = \langle S, I, T, L \rangle$ where S is a set of states, $I \subseteq S$ is the set of initial states, $T \in S \times S$ is the transition relation and $L : S \to 2^{|A|}$ is a labeling function. A is the set of atomic propositions and $2^{|A|}$ denotes the power set of A. The labeling function attaches observations to states, i.e. $L(s)$ denotes the atomic propositions that hold in state $s \in S$.

Example 5. Figure 2 shows a simple Kripke structure $M = \langle S, I, T, L \rangle$ with $S = \{s_0, s_1, s_2, s_3\}$, $I = \{s_0\}$, $T = \{(s_0, s_1), (s_1, s_0), (s_1, s_2), (s_2, s_1), (s_2, s_3), (s_3, s_2)\}$, and $L(s_i) =_{df} s = i \ \wedge mS = i$ where $i = 0, 1, 2, 3$. □

The sequential behavior of a Kripke structure is given in terms of paths, i.e. sequences of states.

Biere et al. [10] showed how to reduce the problem of bounded model checking to propositional satisfiability. This reduction allows to use efficient SAT solvers for bounded model-checking.

Bounded model-checking with a SAT solver of a Kripke structure M against a formula f with bound k is conducted by constructing a propositional formula $[\![M, f]\!]_k$. The constructed formula $[\![M, f]\!]_k$ encodes a constraint on a path $\pi = (s_0, \ldots, s_k)$ such that $[\![M, f]\!]_k$ is satisfiable if π does not satisfy f. As LTL formulas are defined over all paths, M does not satisfy f, if we find such a π.

For the formulas involved in the IntiSa approach, we have the following translation formula for a Kripke structure and an LTL formula:

Definition 7 (Translation): Given a Kripke structure $M = \langle S, I, T, L \rangle$, an LTL formula f and a $k \geq 0$, then

$$[\![M, f]\!]_k =_{df} I(s_0) \wedge \bigwedge_{i_0}^{k-1} T(s_i, s_{i+1}) \wedge [\![f]\!]_k^0$$

where $[\![f]\!]_k^i =_{df} false$ if $i > k$ and otherwise:

$$[\![p]\!]_k^i =_{df} p(s_i) \qquad [\![f \vee g]\!]_k^i =_{df} [\![f]\!]_k^i \vee [\![g]\!]_k^i$$
$$[\![\neg p]\!]_k^i =_{df} \neg p(s_i) \qquad [\![f \wedge g]\!]_k^i =_{df} [\![f]\!]_k^i \wedge [\![g]\!]_k^i$$
$$[\![\mathbf{F} f]\!]_k^i =_{df} [\![f]\!]_k^i \vee [\![\mathbf{F} f]\!]_k^{i+1}$$

We use SMT (Satisfiability Modulo Theories) solvers, which are an extension to SAT solvers. SMT solvers support arithmetics and other first-order theories like uninterpreted functions, arrays, or recursive data types [11]. Hence, for our application, SMT solvers are better suited than pure SAT solvers as they allow a direct use of arithmetics.

IV. The IntiSa Approach

IntiSa calculates initial values for all objects and primitive variables referenced by the method under tests precondition, such that

1) the precondition is satisfied and
2) composes a reachable state for all involved objects.

A reachable state is a state that can be achieved by calling public methods on the object.

Example 6. Consider the *Stack*™ and the precondition $(stack.size() >= 2)$ of the *sumUp*™ method from Figure 1. The value assignment $size() := 2, empty() := true$ composes a state which satisfying but unreachable, since *empty()* may return true if and only if *size()* returns zero. A satisfying and reachable state is given by $size() := 2, empty() := false$. □

Existing test data generation techniques, such as constraint or SMT solvers, return satisfying states, but they may not be reachable. IntiSa, however, returns only states that are satisfying (the precondition of the *MUT*) and reachable.

IntiSa achieves that by featuring the well-known technology of model-checking. Precisely, by SMT solver based model-checking as introduced by Biere et al. [10]. Existing testing approaches that feature model-checking focus on finding a state in the system under test in which a so called safety property is violated. Checking against a safety property, checks whether there exists a state in which the property is violated. Therefore, these approaches model the system under test and return a method sequence for transforming the system under test into a state in which a property is violated. For each method call in that sequence, the approach requires satisfying and reachable test input data.

IntiSa provides it. It does not model the system under test, but all parameters of a method call. For simplicity, we focus in this paper on the call to the method under test, neglecting that the class under test has to be instantiated and transformed by method calls as well. But in general, IntiSa can be used and is used for the empirical evaluation at each of those steps as well. IntiSa also specifies a so called safety property, but with a slightly different meaning.

The objective of INTISA is to find find a state which satisfies the precondition of the method under test.

To achieve this INTISA has to encode objects, build the Kripke structure and specify the property to be checked. An SMT solver is then used to calculate the satisfiability of the given problem and returns a model. A model is a counter example. In other words one possible value assignment to all variables of the problem, such that the formula is satisfied. Those steps are explained in detail in Section IV-A. Furthermore, INTISA has to enhance the given Design by Contract™ specification. The Kripke structure builds upon the enhanced specification, but to foster the understandability we explain those enhancements in Section IV-B, after introducing the solution of the general concept.

A. Building the SMT problem

The SMT problem to build from Definition 7 consists of three parts : (i) the *initial state* declaration, (ii) the *transition relation* by means of a Kripke structure, which defines all possible reachable states of the relevant objects, (iii) and formula f or *goal state*, which describes the state that satisfies the precondition of the method under test.

1) Initial state: All SMT variables are initialized with their default value of the programming language the system under test is written in. We do not initialize those values based on the constructor, because multiple constructor methods may exist that assign different values to the same state variable. Therefore, we model the constructor call similar to method calls within the Kripke structure. Still, this initialization process is required, since the constructor may not assign values to all state variables and the constructor specification may be incomplete.

Example 7. Given the virtual state $VS(Stack) = size, empty, peek, mSize$ from Example 3, INTISA initializes those for the SMT problem as follows: $size = 0 \wedge empty = false \wedge peek = 0.0 \wedge mSize = 0$. □

2) Kripke structure: The Kripke structure is obtained by transforming each method behavior of all methods provided by all relevant objects to a transition relation in such a way that only one method behavior after the other can be chosen at a time. The following paragraphs explain what relevant objects are and how the transition relation is build up.

First, relevant objects are those which states are in any way constrained by the method under tests precondition. This includes all non-primitive parameters and recursively all objects that are returned by methods of objects in this set of relevant objects.

Definition 8 (relevant objects): The set of relevant objects RO is a set of tuples $ro_i = \langle id, C \rangle$, where id is an identifier and C denotes the type the identifier refers to. Given a method m with k non-primitive parameters $np_1, \ldots, np_k$ of types $C_1, \ldots, C_k$ and ${}_{C_i}m^{pub}_{nprim}$, the set of public methods with non-primitive return value of class C_i, the set of relevant objects is given by

$$RO(m) =_{df} \{\langle np_1, C_1 \rangle, \ldots, \langle np_k, C_k \rangle\} \bigcup_{i=1}^{k} RO({}_{C_i}m^{pub}_{nprim})$$

Example 8. The *sumUp* method of the running example from Figure 1 has a single non-primitive parameter $np_1 = stack$ of type $C_1 = Stack$. The set of public methods returning non-primitive types is the empty set ${}_{Stack}m^{pub}_{nprim} = \{\}$. The set of relevant objects therefore consists of only one tuple $RO(sumUp) = \{\langle stack, Stack \rangle\}$ □

To ensure that exactly one method is executed at each step we add a Boolean variable bm_i. This guard does not only provide the construction sequence in terms of method calls, but is required to model a valid solution since we only talk about programming languages with sequential method execution. Therefore, the exclusive disjunction of all bm_i is added by means of conjunction to the transition relation for one step.

Definition 9 (transition relation): Given a set of relevant objects RO, $bm_{m_1}, \ldots, bm_{m_i}$ and $MB_1, \ldots, MB_i$ denote all Boolean guard variables and method behaviors of all relevant objects, respectively. Only method behaviors of public non-pure methods are considered. The transition relation $T(s_i, s_{i+1})$, where s_i denotes all pre-state variables, and s_{i+1} denotes all post-state variables, is given by

$$\begin{aligned} T(s_i, s_{i+1}) =_{df} (bm_{m_1} &\implies MB_1) \wedge \cdots \wedge \\ (bm_{m_i} &\implies MB_i) \wedge \\ (bm_{m_1} &\oplus \ldots \oplus\ bm_{m_i}) \end{aligned}$$

Example 9. Consider the method behaviors $MB_{pop} = \langle\langle size() > 0, size()' = size() - 1 \rangle\rangle$ and $MB_{push} = \langle\langle true, mSize' = mSize + 1 \rangle\rangle$ from the two non-pure methods *pop* and *push* of the $Stack$ from Figure 1. The transition relation contains the guarded method behavior for both methods and in addition an $\oplus$-ed expression forcing exactly one guard to be true.

$$\begin{aligned} &T(s_i,s_{i+1}) = \\ &\quad (bm_{pop} \implies size() > 0 \wedge size()' == size() - 1) \wedge \\ &\quad (bm_{push} \implies true \wedge size()' == size() + 1) \wedge \\ &\quad (bm_{pop} \oplus bm_{push}) \end{aligned}$$

□

3) goal state: The goal state formula is given by the precondition of the method under test. This precondition specifies the state of all relevant objects required by the method under test.

To be able to identify the step in which a state has been reached that satisfies the method under tests precondition a boolean guard variable ($stepGuard_j$) is introduced. By adding the step guard variable by means of conjunction, and allowing only one step guard variable to be satisfied by means of exclusive disjunction of all step guard variables, the SMT solver will set only one step guard to true, which identifies the step in which the goal condition is satisfied. This is only necessary to later read the values for all state variables, which compose the state in which the goal condition is satisfied. Note, step guard $stepGuard_j$ is different from the method behavior guard bm_i in Definition 9. The step guard is not required to provide a valid solution but to identify the set of values that compose the state satisfying the goal condition. Whereas the method behavior guard is required to provide a valid solution, since we talk only about programming languages that only support sequential method calls, not parallel execution. This model is not valid for interleaving method execution, which might occur when testing multi-threaded programs. The goal state formula is built as given in Definition 10.

Definition 10 (Goal State Formula): Given the method behavior MB of the method under test the goal state formula is built as follows,

$$[\![f]\!]_k^0 =_{df} \left(\bigvee_{step=0}^{k} stepGuard_{step} \wedge (P_1 \vee \cdots \vee P_n) \right)$$
$$\wedge \bigvee_{step=0}^{step=k} stepGuard_{step}$$

where P_1, ..., P_n denote the n precondition specifications of MB and k the bound.

Example 10. Using the precondition of the method under test $P_{sumUp} = size() >= 2$ from Figure 1 and a bound $k = 3$ results in the following goal state formula

$$\begin{aligned}[\![f]\!]_2^0 = ((&stepGuard_0 \wedge size() >= 2) \vee \\ &(stepGuard_1 \wedge size() >= 2) \vee \\ &(stepGuard_2 \wedge size() >= 2) \\ &\wedge (stepGuard_0 \vee stepGuard_1 \vee stepGuard_2)\end{aligned}$$

□

4) The SMT solver: An SMT solver is used to check satisfiability of the given Kripke structure and the safety property. In case it is satisfiable, the SMT solver returns a model. We extract the values of the the step that satisfies the goal property. This values define the input values for all objects relevant for the call to the method (under test).

The advantage of SMT solvers is that they can be enhanced with theories, such as equality, (non-)linear arithmetic, bit vectors and arrays. The availability of theories depends on the SMT solver you actually use. Our evaluation of INTISA is carried out with *Yices* [11]. *Yices* supports uninterpreted functions, difference logic, linear real and integer arithmetic, extensional arrays, and bit vectors through the SMT-Lib interface. First order quantification is supported but considered as semi-decidable. *Yices* returns unsatisfiable if it fails to show satisfiability.

INTISA implements the following type mappings from Java to *Yices* types: $\{boolean\} \mapsto bool$, $\{byte, character, integer, short, long\} \mapsto int$, and $\{String, arrays\} \mapsto uninterpreted\ functions$. Objects are encoded similar to ESC/Java [12] in a global array with indices field and objectid.

Objects that are not instantiated are mapped to objectid zero. Therefore, INTISA can handle specifications that require objects to be null or not null. In one of the case studies, e.g., a specification requires one and only one of two parameters to be instantiated, but not both.

Since *Yices* does not provide a String theory, we model it as character array and therefore use the uninterpreted function theory of *Yices*. Note, that the current implementation does only reason about if the String is instantiated and its length. Reasoning about elements within the array is not yet implemented. Future work will not only deal with implementing character array accesses, e.g. stringValue[1]==”a”, but also on mapping rules to model them as SMT problem.

B. Specification Enhancement

Design by Contract™ specification is incomplete and can be added/strengthened iteratively. Specification shortcuts, such as special keywords, make Design by Contract™ languages user-friendly and therefore well applicable to industrial projects. For example, Design by Contract™ specification languages

- provide a *@Pure* keyword to state that calling this method does not change the object state,
- allow pure methods to be called within a specification,
- post conditions do not have to state all state changes explicitly.

The following paragraphs introduce specification enhancement rules for each of the stated issues. Those rules aim for generating a more complete specification by automatically adding information available through combining given specifications without changing the semantics.

The presented rules are improvements compared to those introduced for our SYNTHIA approach [8]. The contributions compared to the earlier version are

- we can now remove the assumption that everything not stated in the postcondition, does not change
- formal definition of chaining for methods with parameters

1) @Pure keyword: Annotating a method with the *@Pure* keyword is an abbreviation to stating that the object state does not change. In our terminology the object state is defined by the virtual state (see Definition 4).

Therefore, INTISA enhances the method behavior specification of all pure methods by adding for each state variable of the according class, that it does not change.

Definition 11 (pure framed method behavior): Given a method behavior MB of a method of class C, the framed method behavior, denoted by $\boxed{MB}$, is given by

$$\boxed{MB} =_{df} \begin{cases} MB \bigwedge \forall v \in VS(C) \bullet v' = v) & \text{if m is pure} \\ MB & \text{otherwise} \end{cases}$$

Example 11. For example, the pure framed method behavior for the *size()* method of Figure 1 with the precondition *true* and the postcondition *@Return=mSize && mSize>=0* and the set of state variables $VS(Stack) = \{size, empty, peek, mSize\}$ (from Example 3) is given by

$$\boxed{\langle true, @Return = mSize\&\&mSize >= 0 \rangle} =$$
$$(true \wedge @Return = mSize\&\&mSize >= 0) \wedge$$
$$size' = size \wedge empty' = empty \wedge$$
$$peek' = peek \wedge mSize' = mSize$$

□

2) Method calls within specification: A method's pre- or postcondition may make use of pure methods. That is, one may call pure methods within a specification.

Example 12. Consider for example a postcondition that states that two elements in an array are sorted $Q = elementAt(index) <= elementAt(index + 1)$. It calls the $elementAt$ method twice with a different argument. □

Therefore, we introduce the term method reference, which denotes a single reference to a method and includes the actual arguments in case the referenced method requires parameters.

Definition 12 (method reference): A method reference MR is given by the j-tuple $\langle m, arg_1, \ldots, arg_i \rangle$ where m denotes the referenced method and arg_1, …, arg_i denote the possibly empty i-tuple of arguments passed to m.

Example 13. The postcondition from Example 12 includes two method references to the same method:

$$MR_{elementAt_1} = \langle m_{elementAt}, 1 \rangle$$
$$MR_{elementAt_2} = \langle m_{elementAt}, 2 \rangle$$

where $m_{elementAt} = \langle elementAt, pub, double, \langle index, int \rangle \rangle$.
□

Furthermore, the specification may contain multiple method calls to the same method with different parameters. In that case, each method reference has to be *instantiated*, i.e., all parameters in the corresponding method behavior have to be replaced with the actual arguments passed and the *@Return* keyword has to be replaced with a unique identifier to be able to reason about the result of all method calls with different parameters.

Definition 13 (instantiated method reference): Given a method reference $MR = \langle m, arg_1, \ldots, arg_i \rangle$, the corresponding method $m = \langle mid, vis, ret, p_1, \ldots, p_i \rangle$, and MB denotes the method behavior of m. Then the instantiated method reference $[MR]$ is given by replacing all parameter occurrences in MB with the actual argument values arg_1, …, arg_i. More formally,

$$[MR] =_{df} MB \left[\begin{array}{c} un\,(MR)\,/@Return \\ arg_0/p_0 \\ \ldots \\ arg_n/p_n \end{array} \right]$$

where $un\,(MR)$ returns a unique identifier for the given method reference.

Example 14. Consider again the first method reference from Example 13 $MR_{elementAt_1} = \langle m_{elementAt}, 1 \rangle$ where $m_{elementAt} = \langle elementAt, pub, double, \langle index, int \rangle \rangle$ and the method behavior of the *elementAt* method $MB_{elementAt} = size() > index \wedge @Return = mArray[index]$ where $mArray$ is a model field of type array of double. The instantiated method reference replaces all general parameters in the method behavior with actual values from the method reference:

$$[MR_{elementAt}] = size() > 1 \wedge @Return = mArray[1]$$

□

All those referenced methods may have a specification as well. Even though, methods referenced within a specification may not change the objects state, the contract may include information that can be useful.

Example 15. Consider the postcondition $Q_{pop} = size()' = size() - 1$ of the *pop* method from Figure 1. It only specifies the result of executing the method with respect to size(). The model field $mSize$ is not updated. The postcondition of *push* $Q_{push} = mSize' = mSize + 1$, however, specifies with respect to the model field only. The link is missing. □

Thus, we have to extend the specifications with the contracts of all referenced methods. We call this *contract chaining*.

Definition 14 (contract chaining): Given a specification R and let $MR_1, \ldots, MR_m$ and $MR_n, \ldots, MR_z$ be the method references for all methods used in terms of undecorated and decorated variables in R, respectively. Furthermore, $un\,(MR)$ returns a unique identifier for the given method reference, the same identifier as in Definition 14. Then the chained specification, denoted by R^∞, is given by

$$R^\infty =_{df} \mu \bullet \left(\begin{array}{c} [MR_1]^\infty; \ldots; [MR_m]^\infty; \\ R \left[\begin{array}{c} un(MR_1)/MR_1 \\ \ldots \\ un(MR_z)/MR_z \end{array} \right]; \\ [MR_n]^\infty; \ldots; [MR_z]^\infty \end{array} \right)$$

where $MR^\infty =_{df} MB^\infty$ with MB being the method behavior of the referenced method in MR and

$$MB^\infty =_{df} (P_1 \wedge Q_1)^\infty \vee \cdots \vee (P_n \wedge Q_n)^\infty$$

Example 16. For example, consider the postcondition $Q_{pop} = (size()' = size() - 1)$ for the *pop()* method of our running example in Figure 1. The instantiated method behavior for *size* is given by $[MB_{size}]true \wedge size()' = mSize' \wedge mSize' >= 0$, and $un(size()) = size()$. The chained contract is given by

$$\begin{aligned} {Q_{pop}}^{\infty} =& MB_{size}; Q_{pop}[un(size())/size()]; MB_{size} \\ =& true \wedge size()' = mSize' \wedge mSize' >= 0; \\ & (size()' = size() - 1); \\ & true \wedge size()' = mSize' \wedge mSize' >= 0 \end{aligned}$$

□

V. Experimental Evaluation

For our experimental evaluation we have implemented the IntiSa approach by means of the *Yices* SMT solver [11] and the *jConTest* [13] test generation tool. *jConTest* automatically generates unit tests for all methods of a given Java application. The case study is specified in Modern Jass, a Design by Contract™ specification language that features Java Annotations facility and supports Java 1.6. For this evaluation we configured *jConTest* to combine the presented IntiSa approach with Synthia fake [8] objects. In other words, for evaluating IntiSa *jConTest* generates unit tests that instantiate Synthia fake objects for all non-primitive test input parameters, and configures their initial state with the values calculated by IntiSa at test generation time.

A. Case Study and Procedure

IntiSa is evaluated on two case studies. *StackCalc* is a stack based calculator written by students at our institute as course work and it was later enhanced with Design by Contract™ specification by the authors. *StreamingFeeder* is provided by or industry partner from the telecommunication industry. It is their second project, which they annotated with Design by Contract™ specifications. The first one, where they wrote the specification before the implementation.

StackCalc is a very interesting case study since it is full of programming concepts, e.g., the command design pattern and the factory design pattern. It consists of 51 classes with 147 methods. It counts about 860 lines of code (without comments) and 136 lines of specification.

StreamingFeeder is even more challenging since it is a real-world application, developed by professional software developer that use all possible features of the Java programming languages. It consists of a very complex class hierarchy, features template methods, constructors which require multiple other objects to be instantiated, as well as generic classes and methods. Furthermore, the specification is written by Java developers and not computer scientists. Therefore, they use for example the *instanceof* operator of Java in the specification, just because Modern Jass allows it and it is convenient to use it. The *StreamingFeeder* library, which we tested, consists of 167 methods in 36 classes and totals in 1456 non-commenting lines of code. But it also includes two other libraries, which were not tested but are required to analyze the case study and therefore were used by IntiSa to model the Kripke structure. This adds another 540 classes to it. Please note that we had to replace one super interface with a self-written interface since our current implementation does not support generic methods where the return type is a generic type parameter. In these cases it is not possible to find out the actual type of the return value by means of Java reflection. We would have to parse the actual source code to find out the actual return type for each occurrence of the method call. Furthermore, we implemented subclasses for *java.util.List* due to the same reason as mentioned above, and to be able to add Modern Jass specification to it.

We compare IntiSa with a random approach based on JET [14]. The evaluation was conducted on four servers with 2,2GHz AMD Athlon 64bit Dual Core CPU and 2GB RAM. Each approach gets n chances for each method under test to generate test input data. We start with $n = 1$ and increase n by one up to 10. This is due to the fact that random obviously will require more trials to generate precondition satisfying test, but on the other hand requires less time for one trial. We consider all public methods as methods under test. We do not generate tests for constructors. The generated tests are then exported to jUnit files, compiled and executed.

We present the results by plotting the achieved method coverage with respect to n, which can be also interpreted as with respect to required time. By method coverage we understand the amount of methods covered by a unit test that satisfies the methods precondition with respect to the amount of methods under test. The presented results are average values over 30 runs, which is again to give the random a fair chance.

B. Empirical Results

The empirical evaluation is as expected. IntiSa is able to generate more tests that satisfy the precondition of the method under test. And it achieves that high coverage with only one trial. The minimal improvement of IntiSa over time as can be seen in Figure 3 and 4 is due to the fact that the implementation of IntiSa still uses minor random decisions, e.g. for String values.

As expected, the random approach heavily improves over time. Especially for the structural less complex *StackCalc* random eventually catches up with IntiSa. For the more complex real-world case study *StreamingFeeder*, random does not show that high improvement rate. This is mainly due to the more sophisticated specification written for the industry case study.

IntiSa still misses 20% points to be able to cover all methods under test. This is due to missing transformation of regular expression specifications to an equivalent SMT formula, to specifications that are not supported by the tool that implements IntiSa, such as the *instanceof* operator,

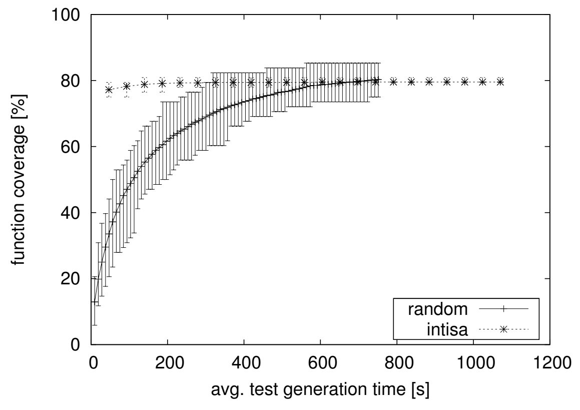

Figure 3. *StackCalc*

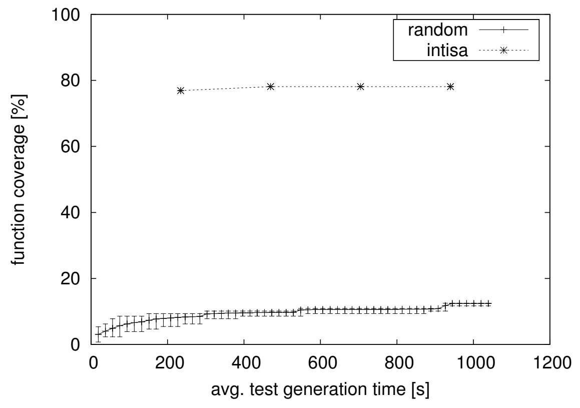

Figure 4. *StreamingFeeder*

and due to runtime exceptions that may be thrown by methods under test.

VI. Related Work

Visser et al. introduced *JPF*, the Java Path Finder [15] in 2004. It uses model checking and symbolic execution to generate input data. Using symbolic execution of branching conditions enables *JPF* to generate input values, that lead to the execution of desired code paths. *PEX* [16] is developed by Microsoft Research. It uses symbolic execution, similar to *JPF*, of branch conditions to generate input data for *.NET* programs that achieves high branch coverage.

Among others, Mark Harman [17] and Tonella et al. [18] work on search-based algorithm for test data generation. Search-based algorithm are based on the theory of evolution. The problem of constructing non-primitive objects is modeled as a search problem. A fitness function determines how good the data is with respect to achieving the given goal, e.g., amount of lines covered, amount of branches covered. Techniques such as crossing-over and mutation help in generating new sets of data based on an initial (random) data set.

Cheon [19] and Ciupa et al. [20] work on random based approaches. The key idea of Cheon is to construct objects incrementally. First choosing randomly a constructor for instantiating the requested object. Afterwards, transforming the object state by calling a random number of methods on it. This random approach is used for our case studies. Ciupa extended adaptive random testing [19] from Cheon for object types. ARTOO [20] works with two sets of test data: the candidate set and the applied data set. The candidate set is filled with randomly generated values. Values used for test generation are moved from the candidate set to the applied data set. The value from the candidate set which has the highest distance value to the already applied values is used the next time. Therefore, Ciupa introduced the notion of an object-distance, which is calculated recursively based on its members and the distance in the inheritance tree of the two compared objects.

Fraser et al. [21] give a very good overview of testing with model checkers. Basically, testing with model checkers focus on deriving test cases that reveal an error. As already mentioned in the introduction, a test case focuses on the state in which the receiver object of the method under test should be in, but not on the parameters passed to the method under test.

In recent years automating the usage of mock objects as test input caught research attention. Tillmann et al. [22], Saff et al. [23], and others work on techniques that use *Mock Objects* as test input data. All have in common that they require in addition to the program under test, a running application or a set of tests to observe and record the behavior of the system under test. Those approaches do not guarantee that the behavior of the *Mock Object* corresponds to the intended one - since no formal specification is used. In addition, they may find errors, that are caused only by a change in the method call sequence, even though the semantics did not change at all.

Our earlier work on SYNTHIA [8] is free of those limitations. SYNTHIA automatically generates fake, i.e., mock, classes that replace the original implementation at test execution time. Every time a method is called on a SYNTHIA fake, its Design by Contract™ specification is used to update a constraint system. In case the method has to return a value, an SMT solver is asked to provide it.

VII. Conclusion

The presented INTISA approach automatically derives test input data from Design by Contract™ specification. The generated data does not only satisfy the precondition of the method under test but guarantees that the generated data describes a state reachable through the public interface of all involved objects. Thus, INTISA returns the construction sequence for all non-primitive parameters, and values for all primitive parameters. Furthermore, all publicly observable values for the non-primitive parameters in their final state are calculated.

For our evaluation, we combined INTISA with SYNTHIA [8], an earlier approach for automatically generating mock objects from Design by Contract™, which are then used instead of non-primitive parameter instances. We performed the evaluation on two case studies written in Java with Modern Jass [4] Design by Contract™ specification. First results show that INTISA is able to generate more precondition satisfying test input values than random. Very promising is the observation that on the more complex industry size case study does far better than random as compared to the structural less complex *StackCalc* case study.

ACKNOWLEDGMENT

The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsfoerderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

REFERENCES

[1] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, objects," in *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM Press, 2004, pp. 236–246.

[2] R. Sharma, M. Gligoric, V. Jagannath, and D. Marinov, "A Comparison of Constraint-Based and Sequence-Based Generation of Complex Input Data Structures," *2nd Workshop on Constraints in Software Testing, Verification and Analysis*, Apr. 2010.

[3] B. Meyer, *Object Oriented Software Construction*, 1st ed. Englewood Cliffs, NJ: Prentice Hall PTR, 1997.

[4] J. Rieken, "Design by contract for java - revised," Master's thesis, Department für Informatik, Universität Oldenburg, April 2007.

[5] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, October 1992.

[6] G. Leavens and Y. Cheon, "Design by contract with JML," 2003.

[7] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards, "Model variables: cleanly supporting abstraction in design by contract: Research articles," *Softw. Pract. Exper.*, vol. 35, no. 6, pp. 583–599, May 2005.

[8] S. J. Galler, M. Weiglhofer, and F. Wotawa, "Synthesize it: from Design by Contract to Meaningful Test Input Data," in *Software Engineering and Formal Methods*, 2010.

[9] C. A. R. Hoare and J. He, *Unifying Theories of Programming*. Upper Saddle River, NJ, USA: Prentice Hall, 1998.

[10] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 118–149, 2003.

[11] B. Dutertre and L. de Moura, "The yices smt solver," Tool paper at http://yices.csl.sri.com/tool-paper.pdf, SRI International, August 2006.

[12] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata, "Extended static checking for java," in *ACM Sigplan Notices*, vol. 37, no. 5. New York, NY, USA: ACM Press, 2002, pp. 234–245.

[13] T. Quaritsch, "jConTest: Describing Test Data Dependencies and Design by Contract Specifications for Automatic Test Data Generation," 2010.

[14] Y. Cheon and C. E. Rubio-Medrano, "Random test data generation for java classes annotated with jml specifications," Department of Computer Science The University of Texas at El Paso, 500 West University Avenue, El Paso, Texas, USA, Tech. Rep., 2007.

[15] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation with Java PathFinder," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[16] N. Tillmann and J. de Halleux, "Pex – white box test generation for .NET," in *2nd International Conference on Tests and Proofs*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hähnle, Eds., vol. 4966. Berlin, Heidelberg: Springer Berlin Heidelberg, April 2008, pp. 134–153.

[17] M. Harman, F. Islam, T. Xie, and S. Wappler, "Automated test data generation for aspect-oriented programs," in *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2009, pp. 185–196.

[18] P. Tonella, "Evolutionary testing of classes," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 119–128, July 2004.

[19] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," *Advances in Computer Science - ASIAN 2004*, pp. 320–329, 2004.

[20] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo: Adaptive random testing for object-oriented software," in *International Conference on Software Engineering 2008*, May 2008.

[21] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, Sep. 2009.

[22] N. Tillmann and W. Schulte, "Mock-object generation with behavior," in *Proceedings of the 21st IEEE Int'l Conference on Automated Software Engineering (ASE'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 365–368.

[23] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM, 2005, pp. 114–123.