Combining Testing and Verification

Tihomir Gvero

LARA, I&C, EPFL

***Abstract*—The paper presents techniques used in both software testing and verification. Korat is the tool that has an effective technique for generation of complex data structures. CVC3 is the state of the art system for solving complex quantified formulas. Partial evaluation is the technique used to speedup a computation of systems by specialization. Testing (represented by Korat) is used to find bugs in a software, where verification (represented by CVC3) is used to prove their absence in software. The both approaches have their own advantages. The natural question is how to use them together. My research plan is to use Korat together with partial evaluation to simplify formulas that will be delivered to CVC3.**

***Index Terms*—Verification, Software Testing, Partial Evaluation**

I. Introduction

TWO main approaches used to increase software quality and reliability are software testing and verification. However, they have different ways to achieve this goal. Software testing represents running a program with a set of inputs to gain confidence that a result meets its expectation. On the other hand, verification represents formally proving that a software meets its specification. Importance of testing and verification is growing as the consequences of software bugs become more severe. The goal of this paper is to present techniques used in software testing and verification, and to show a possible way to combine them.

Korat [2] is a tool for automated software testing. The heart of Korat is a technique for bounded-exhaustive input test generation. This technique is suitable for generation of complex data structures. The specification of data structure is defined by a class (determines links in the structure) and a predicate (specifies additional structural constraints). Given an input specification and a finitization that bounds the desired test input size, Korat generates all inputs (within the bounds) for which the predicate holds. This systematic search is known as the Korat algorithm.

We have developed UDITA [4] a language for test input generation. UDITA is based on Korat algorithm, but it is more expressive. It allows a user to arbitrarily combine a predicate and a finitization, which was not the case with Korat. UDITA is implemented in Java PathFinder [5], the backtrackable interpreter for a Java bytecode. However, UDITA is slower than Korat because of an overhead introduced by the interpreter. We are planning to use Korat algorithm for variable instantiation, especially if the variable is complex data structure.

There are two approaches to checking first order logic formulas. One is the automated theorem proving (ATP) and the other is automated theorem proving based on satisfiability modulo theories (SMT). The advantage of ATP systems is that they are better in reasoning about the quantified formulas than SMT systems. On the other hand, advantage of SMT systems is that they are more effective in reasoning with respect to a given background theory. CVC3 [1] is the SMT solver that combines the advantages of both ATP and SMT systems. The developers of the CVC3 were motivated by another SMT solver, called Simplify. Simplify was the first SMT solver with an acceptable performance when reasoning about the quantified formulas. The goal of CVC3 developers was to improve techniques used in Simplify [10] and introduce new ones, among which the best results gave a *level instantiation* technique.

Partial evaluation is a technique used for optimization of programs by specialization. The idea is that a given computation process can be evaluate with some, not all, values of process's variables. The result is the new computation process that might the starting process. In my research I plan to use partial evaluation to simplify complex formulas and recursions. In this way the computation done by a SMT solver and an interpreter will be faster.

The goal of my research is to develop a system that will be used for testing and verification of modern software systems. The base of the system will be an interpreter for Isabelle

Proposal submitted to committee: May 25th, 2010; Candidacy exam date: May 27th, 2010; Candidacy exam committee: Prof. Willy Zwaenepoel, Prof. Viktor Kuncak, Prof. Martin Odersky

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(name and signature)

Thesis director: _____
(name and signature)

Thesis co-director: _____
(if applicable) (name and signature)

Doct. prog. director: _____
(R. Urbanke) (signature)

code. Isabelle is the interactive theorem prover. As an input Isabelle takes a high order formula. Most of these formulas are complex and hard to solve. I am planning to use three approaches to address this problem. The first approach is to simplify the formula by partial evaluation and to pass it to a SMT solver. While doing this my system will attempt to evaluate only variables that have small and bound domain. The second approach is to split the formula into the simpler formulas and to try to solved them with the SMT solver. Finally, the last approach is to use a combination of the two previous approaches.

Similar to my research is the work done with SMASH tool [9]. The tool aims to find bugs in a complex code. It decomposes the code into peaces (function calls) and try to verify properties or to test them under inferred preconditions. The inferred preconditions bound the inputs to this peaces. However, the approach does not guarantee termination when recursions are analyzed. I hope that the partial evaluation will allow me to make recursion simpler and to prove or disprove its properties. Also, their approach stays always in the domain of decidable problems. On the other hand, my approach will attempt to find counterexamples in undecidable formulas by partially evaluating them to decidable formulas.

Another related work is the symbolic execution technique [7]. It collects constraints over variables on an execution paths. Additionally, it collects branching conditions on the paths. The conjecture of this constraints represents the formula that can be solved by a SMT solver. The difference with my approach is that it does not use concrete values to partially evaluate the formulas. Concolic execution [8] is the technique that uses combination of symbolic execution and testing with concrete inputs. However, the concrete test inputs are used for guiding the symbolic execution, and not to simplify the formulas extracted by symbolic execution.

II. Background

In this section I will first present Korat, the tool for software testing, then CVC3 the system for solving quantified formulas, and finally the partial evaluation technique, used to optimize and simplify systems.

A. Korat: Automated Testing Based on Java Predicates [2]

Manual testing is labor intensive and error prone. Therefore, a research on automated testing is of a great importance. The authors propose Korat technique for automated testing of Java programs. The technique tests a method that is annotated with a precondition and a postcondition. A method is correct if a precondition holds before and a postcondition holds after the execution of method. To check this, Korat uses a formal description of a precondition and automatically generates all possible (non-isomorphic) test cases. These test cases are used one by one as the method inputs. Korat executes the method with the input and checks if output satisfies the postcondition. If the postcondition is not satisfied Korat generates a counterexample. After, the counterexample can be used in debugging purpose.

A precondition and a postcondition of a Java method a user expresses in the Java Modeling Language (JML) code. The JML code is convenient for specification because it can be translated into the Java code. The translated Java code contains assertions that check the specification. Besides a precondition and a postcondition, the user writes a class invariant as an imperative predicate (a Java method that returns a boolean value). In Korat, this method is called *repOk*.

The core of the Korat technique is the algorithm for automated test case generation. With a given bound of a domain and a predicate Korat generates all non-isomorphic test cases. These test cases can be complex data structures. For description of a domain Korat uses a finitization, a set of bounds that limits the size of the inputs. The finitization is a Java method that is automatically derived from a Java class (and can be manually modified if needed). The user specifies the number and a type of object for every field of the class. For example, the class *BinaryTree* has integer field *size* that represents a number of nodes in the tree and filed *value* where it stores a value. The class *BinaryTree* also contains field *root* of type *Nodes* that points to a first node in a tree. The class *Node* has fields *left* and *right* that points to a left and a right subtrees. In the finitization method user can defines that field *size* can only have the value *N* and field *value* can have any value from a range *[min, max]*. Also it can define that the domain of *root*, *left* and *right* fields is set of *N* objects of the type *Node* plus value.

In Korat, each test case is represented by a candidate vector. Each position in candidate vector represents a field of an object from unified set of objects. The unified set of objects represents the union of all field domains. For instance, if the unified set has *5* objects and if they all have *3* fields, the size of candidate vector will be *15*. Also, Korat assigns different indices (from zero to a size of a domain minus 1) to every object in every object set. The elements of the candidate vector are these indeces. For example, at the position *i* the candidate vector defines a filed which domain is a set *D*. The possible values at that position are *0* to maximum size of *D* minus 1.

Korat executes *repOk* and uses backtracking mechanism to generate all valid candidate vectors, i.e. those candidate vectors for which *repOk* returns *true*. At the beginning Korat sets all values in the candidate vector to *0*. Korat algorithm monitors a first access to all fields that are done during the execution. It makes a *field-ordering* a list of the filed identifiers ordered by the first time *repOk* accessed the corresponding filed. The filed identifier represents its position in the candidate vector. If *repOk* returns *true* Korat outputs the current candidate vector as a valid test case. It also generates all candidates that can be generated by fixing the values of fields in the *field-ordering* and varying all possible values of non-accessed fields. After this, Korat backtracks and increment a field domain index of the last field in the *field-ordering*. If the index exceeds the maximum value, Korat resets it to *0*, and increment the domain index of the previous field in the *field-ordering*. The procedure repeats until the one of the domain indices is incremented without exceeding. Then *repOk* is executed with the new candidate. Algorithm terminates when the all domain indices exceed maximum values. If the predicate value is false then the

procedure is the same, but Korat instantly increase the domain index of a field, without generating any candidate. Thus, Korat omits candidates that could be generated by varying all possible values of non-accessed fields, and fixing values of fields in *field-ordering*. The experimental results shows that in this way Korat prunes large portion of the search space. Additionally, algorithm implementation is refined such that it produces only structurally non-isomorphic test cases.

The authors used several data structures for performance tests. The results show that Korat is able to generate all structures within a very large state spaces. This is due to the effective pruning based on fields accessed during *repOk*'s execution. They also show that generating only non-isomorphic structures reduces the number of valid data structures.

The authors also compared test case generation of Korat and the Alloy Analyzer (AA) [11]. Allow is the declarative language based on relations. It can be used for modeling structural properties of a program, and is convenient for modeling class invariants. These invariants correspond to *repOk* method in Korat. With Alloy user can also declare field types, which corresponds to assigning a domains to fields in Korat.

From these descriptions AA can generate all (mostly non-isomorphic) *instances* of the model. An instance evaluates relation such that all constraints of the model are satisfied. AA translates the Alloy model into a boolean formula and uses a SAT solver to find a assignment that satisfies the formula. The assignment is then translated back to an instance of the input model. To reduce number of isomorphic instances AA adds symmetry breaking predicates to a formula.

Korat and AA were compared on a set of date structures. Korat outperforms AA, showing that it is faster in date structure generation. There are two reasons for this. Because AA translates an Alloy model to a formula it could be that the translator generates many unnecessary large formulas. The second reason is that AA generates more instances than Korat.

The authors tested Korat effectiveness in testing correctness of methods. The experiments were run on several methods. Korat generated all structures from size *0* up to a given size maximum size. The structures were use as inputs to the methods. The results show that Korat is practical for correctness checking of methods.

There are few drawbacks of Korat. The first is that it does not support floating point types. This can be overcome in combination with symbolic execution. The second is that it can be used only for a black-box testing. This is partly overcome in the UDITA language that allows user to test a code in a white-box manner. Because of the impressive results we are planning to use Korat algorithm in combination with symbolic execution for variable instantiation.

B. Solving Quantified Verification Conditions Using Satisfiability Modulo Theories [1]

Many conditions like preconditions or postconditions can be expressed by quantified formulas. Automated theorem proving (ATP) system are efficient for solving quantified formulas, but have a problem in reasoning with respect to a given theory. On the other hand, satisfiability modulo theories (SMT) systems are efficient for reasoning with respect to a theory, but are less efficient for solving quantified formulas. The previous research attempts to combine the advantages of these two systems resulted in the Simplify tool, the SMT solver that supports reasoning about quantifiers. The goal of the authors was to build even more effective SMT solver based on the DPLL(T) architecture [13] with efficient instantiation technique. The first step was to extended the Abstract DPLL Modulo Theories framework [12] with a rules for quantification. Then, the authors implemented Simplify's instantiation techniques into a new SMT system, CVC3, and improved it with a *instantiation level* technique. The technique is responsible for prioritizing large number of terms that are candidates for quantifier instantiation.

The SMT problem consists of solving some closed first order formula φ, with respect to some fixed background theory T with signature Σ. Also it is desirable to allow the formula to contain additional free symbols, i.e. constant, function, and predicate symbols not in Σ. We say that φ is $T - satisfiable$ if there is an expansion of a model of T to the free symbols in φ that satisfies φ. Tools used for solving these formulas are called SMT solvers.

The Abstract DPLL Modulo Theories framework represents a SMT solver as transition system. The state of the system is either $Fail$ or $M \parallel F$, where M is the set of premises (literals that hold) and F is a formula in CNF which needs to be checked. The transition relations are defined over the states by the transition rules. Starting from initial state $\emptyset \parallel F_0$, by applying transition rules the system goes toward a final state. If the final state is $Fail$ the formula is $T - unsatisfiable$, otherwise if it is $M \parallel F$ then it is $T - satisfiable$ if every clause in F is satisfiable by literals from set M. The two representative rules are:

$$Unit Propagation:$$

$$M \parallel F, C \vee l \Rightarrow Ml \parallel F, C \vee l \quad if \begin{cases} M \models \neg C \\ l \quad is \quad undefined \quad in \quad M \end{cases}$$

$$T - Propagation:$$

$$M \parallel F \Rightarrow Ml \parallel F \quad if \begin{cases} M \models_T \neg C \\ l \quad or \quad \neg l \quad occurs \quad in \quad F \\ l \quad is \quad undefined \quad in \quad M \end{cases}$$

Where C is s clause, l is a literal, $\models$ is propositional entailment, $\models_T$ is first-order entailment modulo the background theory T.

The authors define a new transition system in order to support quantified formulas. In the system they allow quantified formulas to occur in M and F wherever atomic formulas occur. Also, abstract atomic formulas are quantified or atomic formulas. Additionally, an abstract literal is an atomic formula or its negation, and an abstract closure is a disjunction of abstract literals. Finally, the authors extended The Abstract DPLL Modulo Theories framework with rules that allow quantified instantiation:

$$\exists - Inst:$$

$$M \parallel F \Rightarrow M \parallel F, \neg\exists \bar{x}.\varphi \vee \varphi[\bar{x}/\bar{c}] \quad if \begin{cases} \exists \bar{x}.\varphi \quad in \quad M \\ \bar{c} \quad are \quad fresh \quad constants \end{cases}$$

$$\forall - Inst:$$

$$M \parallel F \Rightarrow M \parallel F, \neg\forall \bar{x}.\varphi \vee \varphi[\bar{x}/\bar{s}] \quad if \begin{cases} \forall \bar{x}.\varphi \quad in \quad M \\ \bar{s} \quad are \quad ground \quad terms \end{cases}$$

Where, $\exists\bar{x}.\varphi$ stands for $\exists\bar{x}.\exists\bar{x}.\ldots\exists\bar{x}.\varphi$, and analogous for $\forall\bar{x}.\varphi$. Furthermore, the rule $\exists - Inst$ instantiates $\exists\bar{x}.\varphi$ with fresh constants $\bar{c}$, to get a ground formula $\varphi[\bar{x}/\bar{c}]$. ($\varphi[\bar{x}/\bar{c}]$ denotes that $\bar{c}$ replaces $\bar{x}$ in the formula φ). Then the added closure on the right hand side is $\neg\exists\bar{x} \vee \varphi[\bar{x}/\bar{c}]$, that preserves the satisfiability of F. Similarly, $\forall - Inst$ rule instantiates $\forall\bar{x}$ with ground terms $\bar{s}$ to get a clause $\neg\forall\bar{x}.\varphi \vee \varphi[\bar{x}/\bar{s}]$.

It is important to note that for a given existentially quantified formula it is enough to apply the $\exists - Inst$ rule only once. This is because all fresh constants are identical. On the other hand, a universally quantified formula can be instantiated with instantiated with many ground literals. They are not semantically or syntactically identical as fresh constants. This paper focuses on new efficient strategies for instantiation of universally quantified formulas or clauses.

One approach is a naive strategy that whenever $\forall - Inst$ is selected for application to an abstract literal $\forall\bar{x}.\varphi$ the rule is repeatedly applied until $\bar{x}$ is instantiated with every possible tuple from some finite set G. Mostly, G represents a set of all ground literals that appears in M. But, this approach can be inefficient. The Simplify uses refined strategy: it tries to find a subterm t of $\forall x.\varphi$ (that is in M) that contains x, and a ground term g in M and its subterm s, such that $t[x/s] =_T g$, i.e. $t[x/s]$ is equivalent to g modulo the background theory T. The term t is called a trigger. For matching of a trigger and a ground literal, which is not trivial, Simplify uses syntactic matching based on the congruence closure of the ground equations in M.

An orthogonal question is *when* to apply the $\forall - Inst$ rule. There are two different strategies: *lazy instantiation* and *eager instantiation*. *Lazy instantiation* applies the $\forall - Inst$ rule only when it is the only applicable rule. On the other hand, *eager instantiation* applies the rule as soon as it is added to M. Simplify implements a variation of a *eager instantiation*.

In CVC3 every subterm or non-equational atom t that contains all variables in $\bar{x}$ is a potential trigger. CVC3 also allows appearance of additional variables beside $\bar{x}$, where Simplify is more restrictive end does not allow their appearance. Experiments in the paper show that this restriction is unnecessary.

Simplify performs syntactic check to prevent instantiation loops. It eliminates any trigger whose syntactical instances occur elsewhere in the formula, which is insufficient for detecting more subtle loops. For instance, if M contains the abstract literal $\psi = \forall x.(x > 0 \rightarrow \exists y.f(x) = f(y) + 1)$ where f is free, the only trigger is $f(x)$. Next, if the set of ground terms contains $f(3)$, then with an application of $\forall - Inst$, it is possible to add the abstract clause $\neg\psi \vee \exists y.f(3) = f(y)+1$ to F. Then, if Simplify applies UnitPropagate and $\exists - Inst$ rules the literal $f(3) = f(c1) + 1$, where $c1$ is fresh constant, will be added to M. The introduction of $f(c1)$ in the set of ground terms can now give rise to a similar round of rule applications generating a new term $f(c2)$, and so on. In order to avoid this, beside synthetic check, CVC3 also dynamically recognizes loops that can be formed by a group of formulas. We will refer to this strategy as *smart triggers*.

Another strategy that CVC3 implements is *smart matching*. In order to perform effective term matching, after M is modified, CVC3 computes and stores the congruence closure E of the positive ground literals of M over the set G of all ground terms in M. For any theory T, any two terms equal modulo E are also equal modulo $T \cup M$. Then, to apply the rule $\forall - Inst$ to an abstract literal $\forall x.\varphi$, CVC3 generates ground instantiations for x by matching modulo E the triggers of $\forall x.\varphi$. against the terms in G. Additionally, CVC3 implements syntactic unification algorithm. Given a trigger t of the form $f(t1, ..., tn)$ where f is a free symbol, CVC3 selects from G all terms of the form $f(s1, ..., sn)$. For each of these terms CVC3 tries to solve the unification problem $t1 =^? s1, ..., tn =^? sn$. The unification does not immediately fail if $g(\bar{t}) =^? g'(\bar{s})$, where g and g' are distinct symbols. It does not fail in the following two subcases: (i) $g(\bar{t})$ is ground and $g(\bar{t}) =_E s$,2 and (ii) g is a free symbol and there is a term of the form $g(\bar{u})$ in G such that $s =_E g(\bar{u})$. In the first case CVC3 removes the equation $g(\bar{t}) =?\bar{s}$, and in the second case, it replaces it by the set of equations $\bar{t} =^? \bar{u}$.

When verifying an application, generated conditions are of type $\Gamma \wedge \neg\varphi$, where Γ represents large number of axioms for which there is no built-in solver. Furthermore, many of these axioms may be irrelevant for proving φ. The authors point out that many resources can be easily spent on producing and processing instances of these unrelated axioms. Therefore reducing the number of axioms to those that are relevant is the key challenge. To solve this problem, Simplify uses global counter that marks every new clause generated by quantified instantiation. The new clause is marked with a current value of the global counter. Then the global counter is incremented by 1. Later, when case-split occurs literals from clauses that are marked with the lower value has priority. On the other hand, CVC3 uses different strategy. For each clause it assigns local value, called an *instantiation level*. If the *instantiation level* for a term t is n, that means that t is the result of n rounds of instantiations. At the beginning, *instantiation level* of all terms it equal to 0. If a formula $\forall\bar{x}.\varphi$ is instantiated with a ground terms whose maximum instantiation level of all terms is n then all the new terms in $\varphi[\bar{x}/\bar{t}]$ will have instantiation level $n + 1$. Then CVC3 strategy visits ground terms by instantiation levels. Terms with lower instantiation level have higher priority. For a given bound b the strategy will first visit all the ground terms which *instantiation level* is between 0 and b. The bound will be increased only in case if all the ground terms with the level up to b are exercised.

The experimental results shows that this strategy plays an important role in avoiding instigation loops. The reason is that every new ground term generated within an instantiation level n belongs by construction to the next level, $n + 1$, and so will not be considered for matching until all other terms in the level n have been considered. Therefore, performing static or dynamic checking for instantiation loops is unnecessary. These checks remove triggers that are essential for efficient satisfiability check. The *instantiation level* strategy avoids this elimination.

CVC3 was evaluated on a 5599 benchmarks from SMT-LIB. These benchmarks are publicly available and used for testing SMT solvers. In the first set of experiments the authors compared instantiation heuristics: 1) basic trigger/matching

algorithm, 2) basic trigger with smart matching, 3) same as 2) but with smart triggers, 4) same as 3) but with the instantiation level. They ran all experiments both on the lazy and eager instantiation strategies. The eager instantiation is more efficient than lazy both in average time required for solving and number of solved cases (benchmarks). The results proved that the instantiation level is useful technique. Together with eager instantiation it solved 5435 cases, much more than other techniques.

The authors also compared a SMT solver (CVC3) with ATP systems (Vampire [14] and SPASS [15]) and Simplify on benchmarks that require both quantified and theory reasoning (*nasa* benchmarks). All solvers were able to prove most of the benchmarks. Simplify was faster than Vampire and SPASS, but it also proved fewer cases. However, CVC3 was able to solve the most benchmarks with the average time less than a hundredth of a second. CVC3 was also compared with other SMT solvers. The only two solvers at the time that supported quantifiers and SMT-LIB format were: yices and Fx7. CVC3 proved 34 more cases than yices, although yices was faster than CVC3.

C. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler [3]

Two procedures for describing semantics of programming language are a description of a compiler and description of an interpreter. An interpreter represents a procedure that evaluates statements of the defined language. On the other hand, a compiler represents the procedure that translates a code from a defined language to a target language (for which there already exists compiler or interpreter). It is well know that writing a compiler is harder that writing an interpreter. The reason is that while writing a compiler, a developer needs to know which actions are performed during a code translation and which are done at run time. On the other hand, while describing semantics of programming language by an interpreter, an developer does not need to know a distinction between those actions. But at run time an interpreter is often less efficient than a compiler. Therefore, the author describes an algorithm that automatically transforms an interpreter to a compiler. The bases of the algorithm is the partial evaluation technique.

Partial evaluation is a transformation of a computation process π with respect to m variables $c_1, c_2, ..., c_m$ that are substituted with values $c'_1, c'_2, ..., c'_m$. The values must be known before or during the transformation. The computation process π can also have n variables $r_1, r_2, ..., r_n$ that are not known before or during the transformation. Partial evaluation evaluates only portion of π that can be evaluated using the variables $c_1, c_2, ..., c_m$ and constants in π. After partial evaluation, the process π is transformed into the new evaluation process that has the variables $r_1, r_2, ..., r_n$. If we assign values $r'_1, r'_2, ..., r'_n$ to these n variables, a result after evaluating the new computation process using them will be the same as a result of the evaluation of the π with $c'_1, c'_2, ..., c'_m$ and $r'_1, r'_2, ..., r'_n$ assign to $c_1, c_2, ..., c_m$ and $r_1, r_2, ..., r_n$, respectively. We can write this as:

$$\pi(c'_1, c'_2, ..., c'_m, r'_1, r'_2, ..., r'_n) = \alpha(\pi, c'_1, c'_2, ..., c'_m)(r'_1, r'_2, ..., r'_n) \quad (1)$$

where α is *partial evaluation algorithm*, $c'_1, c'_2, ..., c'_m$ are *partial evaluation variables* and $r_1, r_2, ..., r_n$ are *remaining variables*.

An interpreter, int, can be seen as a computation process with variables that can be classified in two groups. The variables from the first group are those which values are a source program and information for syntax and semantic analysis. We denote this group by s, and their values by s'. The variables in other group we denote by r, and corresponding values that will be determined at run time we denote by r'. If we partially evaluate int with respect to s at the values s', then from previous equation (1) we have:

$$int(s', r') = \alpha(int, s')(r') \quad (2)$$

The result of partial evaluation, $\alpha(int, s')$, can be seen as the computation process that is translated into the metalanguage describing the interpreter int. Actually, it can be seen as an object program corresponding to s'.

Furthermore, if α is partially evaluated with respect to int we have:

$$\alpha(int, s')(r') = \alpha(\alpha, int)(s')(r') \quad (3)$$

Here $\alpha(\alpha, int)$ can be considered as a compiler because it generates an object program from s'.

The author specifies the two properties that are desirable for an effective partial evaluation algorithm: 1) While partially evaluating an process π, α evaluates as larger portion of π which can be evaluated with constants and values assigned to partial evaluation variables, and 2) α evaluates as smaller portion of π as possible which is not evaluated when new computation process is evaluated with values of remaining variables. The first property reduces the computation time of the new computation process, where the second property reduces the computation time of partial evaluation. If the algorithm has both properties then the compiled object program is more efficient then interpreter.

The author describes a partial evaluation algorithm in following way. A computation process can be represented as a graph that has nodes n_i (i = 0,1,..., k) that represent conditional branching points and branches b_j (j = 0,1,..., m) between the nodes that represent sub-computational processes not containing branching points. Furthermore, the author introduces leaves that represent the termination of the process.

At each stage the algorithm distinguishes two kind of variables:

- Partial evaluation variables are partial evaluation variables of the previous stage or variables which values depend only on constants and/or partial evaluation variables of the previous stages.
- Remaining variables are variables that are not partial evaluation variables.

The partial evaluation algorithm is described in 5 steps:

1) Let $j(1), j(2), ..., j(m)$ be m integer variable that are all set to 1. (note that m is the number of branches in the graph). Set integer variable g to 1, and make an empty list L. Go to the step 2.
2) Enter a triplet $(b_g, S_g^{j(g)}, a_g^{j(g)})$ in to the list L, where $S_g^{j(g)}$ is a set of pairs of partial evaluation variables and their values (at the entry point of the $j(g)$-th entry to b_g), and $a_g^{j(g)}$ is an address where the result of $j(g)$-th partial evaluation of b_g is generated. Note that the

address can point to a new graph or a value stored in memory. Go to the step 3.

3) Evaluate b_g with partial order variables and constants. Then mark new generated computation process with $b_g^{j(g)}$. Its first address is $a_g^{j(g)}$. At the end, increment the value of $j(g)$ by 1 and go to the step 4.
4) If the process that b_g points to is a leaf (a terminating process) then stop partial evaluation. If it is not a leaf, but a conditional branching point n_i then we have two following cases:
   a) If n_i can be evaluated only with the partial evaluation variables and constants then it determines a branch that will be chosen next. If that branch is b_q then set g to q and go to the step 5.
   b) If n_i can not be evaluated only with the partial evaluation variables and constants then leave it untouched. Then choose one branch that it points to. If that branch is b_q set g to q and go to the step 5. Then do the same with second branch, b_p, but this time set g to p and go to the step 5.
5) For $b_g^{j(g)}$ check if the list L contains a triplet whose first and second terms match with b_g and $S_g^{j(g)}$ respectively.
   a) If it contains the triplet then move the control of the generated computation process to point to third term in the triplet, a_g^x. Stop the partial evaluation.
   b) If the triplet does not exist then return to the step 2.

The problem with a given algorithm is that it can not terminate always due to loops in a computation process. It can terminate if a branch in a loop is selected with a same entry variables more then once. I plan to use partial evaluation to simplify a complex code and formulas, expecting that a SMT solver will be able to solve them easily.

III. Research Proposal

The paper presented several techniques and two systems for testing and verification. The first among them is the Korat, a tool for software testing based on a program specification. The impressive experimental results show that Korat's technique for test case generation is very practical. The second system is CVC3, the SMT solver with powerful technique that allows effective reasoning about quantifiers. Finally, partial evaluation technique can be used to optimize and simplify a computation process. The goal of my research is to develop effective and expressive system for testing and code checking that will combine the power of these systems and techniques.

The first step in development is to implement a system with an architecture that will allow combining ideas described in the previous section. Therefore, I implemented an interpreter for Isabelle [16] code. Isabelle is an interactive theorem prover that allows a user to express mathematical formulas in a formal higher order language. Although, Isabelle aims in proving or disproving some simple formulas, it mostly requires a lot of human assistance. The reason for building an interpreter is to make Isabelle more automated. We also found that translation of verification conditions of programs written in programming languages (like Scala or Java) to Isabelle code is quite convenient. Thus, Isabelle code can be seen as the intermediate code for verification and testing.

I have implemented an interpreter for the Isabella code that supports operations over integers and booleans. It supports conditional branching statement and function definitions. This allows a user to write more sophisticated properties. I also implemented a backtracking engine, in the interpreter, that allows a nondeterministic instantiation of variables. A user can specify a range of a nondeterministic integer variable by giving its minimum and maximum value. For boolean values a user has an option to select which value (*true* or *false*) will be nondeterministically assigned to a variable first. The nondeterminism is used in combination with copy propagation technique that postpones an assignment of value until the first non-copy operation that uses the variable. The interpreter instantiate a variable when the first such non-copy (arithmetic or conditional) operation is reached. The variable is instantiated with one of the values from its domain. After the code is executed, with one value, the backtracking engine will backtrack to the assignment point and choose the next value from domain, and so on. For instantiation of a variable that represents algebraic data structure we plan to use Korat algorithm.

For testing and verification I am planning to extract two different formulas from an Isabelle code. The first formula can be obtained by dynamic symbolic execution that will follow one possible execution path. Symbolic execution collects branching conditions on the path and forms a formula called the *path condition*. The *path condition* is a conjecture of the branching constraints on the path. With conditions over variables it forms a formula whose solution represents a collection of values. When these values are assign to the variables it is guaranteed that the path will be executed. In my system the solution (values) will be found by passing the formula to a SMT solver that supports quantified formulas. This is the place where I plan to use CVC3 or Z3 [6] solvers. This kind of formula could be suitable for finding an counterexample, if a corresponding path violates an property. It is also suitable if there are few possible paths in a code. The second formula is the one that represents entire or part of a code. The observation is that an Isabelle code represents a symbolic formula. Thus, we can directly translate the code into the SMT-LIB format, supported by CVC3 and Z3. In this case to prove that a property is valid, we would need to negate it and check unsatisfiability of its negation. If the negation is satisfiable, that means that the starting code violates a property.

I will use partial evaluation to simplify a complex formula. My system will recognize the most appropriate variables for instantiation in the formula. After instantiating them, I expect to obtain a much simpler formula. The appropriate variables are those that have a bounded domain. This approach can be also used for testing undecidable formulas. Here, the appropriate variables will be those that form the smallest set that will transform an undecidable into a decidable formula. The simplified formulas will be passed to the CVC3 or Z3. It is possible that the SMT solver will easer find counterexample or prove the simplified formula. If we can not solve a formula with an identified set of the appropriate variables we will chose

a random variable and instantiate it. Partial evaluation can also help with solving recursive calls. The recursive function will be partially evaluated, and thus simplified.

The partial evaluation can be used in two different purposes: either to find a counterexample (testing) or to prove a property (verification). When we search for a counterexample, existentially quantified variables needs to be instantiated with every value from their domain. This can be a problem if the domain is large or unbounded. It can also happen that we can not determine boundaries of the domain. In this case we can partially evaluate a formula, and leave these existentially quantified variables untouched and hope that SMT-solver can solve the partially evaluated formula. The same problem occurs with universally quantified variables when we need to prove validity of a formula. Analogously, we will leave these variables uninstantiated, and pass the negation of the formula to a SMT-solver. Thus, decision to prove or find an counterexample in a formula can take into the account the number of uninstantiated existentially and universally quantified variables. On this decision depends which variables can not be evaluate (instantiated) further: existentially or universally quantified variables (with unbounded or large or unknown domain). But it can happen that a chosen variable triggers the execution that will bound some of these variable. In that case we can update the set of appropriate variables.

REFERENCES

[1] G. Yeting, B. Clark, and T. Cesare, *Solving quantified verification conditions using satisfiability modulo theories*. Annals of Mathematics and Artificial Intelligence, volume 55, pages 101-122, Kluwer Academic Publishers, 2009.

[2] C. Boyapati, S. Khurshid, and D. Marinov, *Korat: automated testing based on Java predicates*. ISSTA, pages 123-133, 2002.

[3] F. Yoshihiko, *Partial Evaluation of Computation Process-An Approach to a Compiler-Compiler*. Higher Order Symbol. Comput., Vol. 12, pages 381-391, Kluwer Academic Publishers, 1999.

[4] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, *Test Generation through Programming in UDITA*. 32nd International Conference on Software Engineering, 2010.

[5] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, *Model checking programs*. J-ASE, 10(2), 2003.

[6] L. de Moura,and N. Bjrner, *Z3: An Efficient SMT Solver*. In Tools and Algorithms for the Construction and Analysis of Systems, Volume 4963/2008, pages 337-340, 2008.

[7] J. C. King, *Symbolic execution and program testing*. Commun. ACM, 19(7), 1976.

[8] K. Sen, D. Marinov, and G. Agha. *CUTE: A concolic unit testing engine for C*. In ESEC/FSE, 2005.

[9] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali, *Compositional may-must program analysis: unleashing the power of alternation*. POPL, 2010.

[10] D. Detlefs, G. Nelson,J. B. Saxe, *Simplify: a theorem prover for program checking*. J. ACM 52(3), 365473 (2005)

[11] D. Jackson, I. Schechter, and I. Shlyakhter. *ALCOA: The Alloy constraint analyzer*. In Proc. 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, June 2000.

[12] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, *Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T)*. Journal of the ACM, Vol.53(6), pages 937977, 2006.

[13] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, C. Tinelli, *DPLL(T): Fast decision procedures.* In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, Volume 3114, pp. 175188. Springer, Heidelberg, 2004.

[14] A. Riazanov, and A. Voronkov, *The design and implementation of VAMPIRE*. AI Commun. 15(2-3), pages 91110, 2002.

[15] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen,C. Theobald, D. Topic, *SPASS Version 2.0*. In: Voronkov, A. (ed.) Automated Deduction - CADE-18. LNCS (LNAI), Vol. 2392, pages 275279. Springer, Heidelberg, 2002.

[16] *Isabelle website:* www.cl.cam.ac.uk/research/hvg/isabelle/