A Symbolic Model Checker for Database Programs

Angshuman Jana, Md. Imran Alam and Raju Halder
Indian Institute of Technology Patna, India
{ajana.pcs13, imran.pcs16, halder}@iitp.ac.in

Keywords: Model Checking, Database Program, Boolean Program, Verification, Refinement.

Abstract: Most of the existing model checking approaches refer mainstream languages without considering any database statements. As the result, they are not directly applicable to database applications for verifying their correctness. On the other hand, few works in the literature address the verification of database applications focusing atomicity constraints, transaction properties, etc. In this paper, as an alternative, we propose the design of a symbolic model checker for database programs to verify integrity properties defined over database attributes. The proposed model checker is designed based on the following key modules: (i) Abstraction, (ii) Verification, and (iii) Refinement.

1 INTRODUCTION

Model checking is an algorithmic method for proving that a system satisfies its specification (Clarke and Emerson, 1981; Wang et al., 2006). As stated in (Jhala and Majumdar, 2009), the goal of the model checking research is to expand the scope of automated techniques for program reasoning, both in the scale of programs handled and in the richness of properties that can be checked. Model checker receives application source codes and exhaustively explores their execution states, searching for possible violations of properties of interest. For examples, properties may include simple assertions, which state that a predicate on program variables holds whenever the computation reaches a particular location, or global invariants, that state certain predicates hold on every reachable state (e.g. each array access is within bounds), or termination properties.

Based on the state representations, model checkers are of two types: (i) enumerative (in which individual states are represented) and (ii) symbolic (in which sets of states are represented using constraints). State space explosion is the critical limitation of the enumerative techniques, which led researches to explore symbolic algorithmic approaches. The symbolic model checking approach manipulates the representation of sets of states, rather than individual state, and performs state exploration through the symbolic transformation of these representations (Queille and Sifakis, 1982). For example, the constraint $0 \leq a \leq 9 \wedge 6 \leq b \leq 10$ represents the set of all states over a, b satisfying the constraint, which implicitly represents a list of 50 possible states. Therefore, the symbolic model checking algorithms are more efficient as compared to the enumerative approaches.

Over the past, several enumerative model checkers, *e.g.* Verisoft (Chandra et al., 2002), SPIN (Holzmann, 1997), Cmc (Yang et al., 2006), etc. are developed. On the other hand, many tools like SLAM (Ball and Rajamani, 2002), CBMC (Clarke et al., 2003), F-Soft (Ivancic et al., 2005), JavaPathFinder (Anand et al., 2007), etc. are developed based on symbolic algorithms. Notably, the above tools are built for the verification of various mainstream languages without considering any database statements. Even though intensive research is already done, researchers have not paid much attention towards database programs embedding queries and data-manipulation commands. Few works in the literature include symbolic model checking of stored procedure and SQL queries for automatic validation according to the specification (Diana et al., 2012), explicit-state model checker DPF for the verification of database atomicity constraints in web applications (Gligoric and Majumdar, 2013), etc.

In this paper, as an alternative, we extend the existing symbolic model checking algorithm for imperative language to the case of database programs. In particular, we aim at verifying the correctness of database programs respecting integrity properties defined on the underlying databases.

To summarize, our contributions in this paper are:

1. Abstraction of database programs into boolean programs.
2. Generation of verification conditions (VCs) form the Control Flow Graph (CFG) of boolean database programs and their verification using SMT solver.
3. Counter Example Guided Abstract Refinement (CEGAR) of boolean database programs.

Roadmap: In section 2, we discuss the current state-of-the-art in the literature. In section 3, we describe a motivating example. In section 4, we recall some preliminaries. We introduce our approach in section 5. Section 6 provides an overall tool architecture. Finally section 7 concludes the work.

2 RELATED WORKS

Chandra et al. (Chandra et al., 2002) proposed a tool Verisoft which pioneered the idea of execution-based stateless model checking of software. It uses a scheduler which is able to exhaustively explore all possible interleaving of the processes executions. In (Holzmann, 1997), author proposed SPIN, a tool which supports model-based verification of distributed software systems. It has been used to detect design errors in applications ranging from high-level descriptions of distributed algorithms to detailed code for controlling telephone exchanges. Another execution based model checker Cmc (Yang et al., 2006) is proposed for C programs. The Cmc tool explores different executions by controlling schedules at the level of the OS scheduler. Ball and Rajamani (Ball and Rajamani, 2002) introduced first Counter Example Guided Abstract Refinement (CEGAR)-based symbolic model checker SLAM for C programs. The SLAM tool works in the following three steps: (i) abstracting programs into the form of boolean programs, (ii) verifying properties using model checking algorithm on the boolean programs, and (iii) refinement of the boolean programs based on CEGAR. The model checker Magic (Chaki et al., 2004) was proposed to enable modular verification of concurrent, message passing C programs. Ivancic et al. (Ivancic et al., 2005) proposed a model checker F-Soft that used predicate abstraction along with other abstract domains that efficiently yield the kinds of invariants needed to check standard runtime errors in C programs. JavaPathFinder tool (Anand et al., 2007) is the model checker for Java programs that modifies the Java Virtual Machine to implement systematic search over different thread schedules. Several other model checkers CBMC (Clarke et al., 2003), Chess (Musuvathi and Qadeer, 2007), etc. are also developed in the literature.

In (Paleari et al., 2008), the authors proposed a dynamic approach to detect race condition vulnerabilities on web-based applications. The approach analyzes a log file of a single run and identifies dependencies among SQL queries based on the set of relations and attributes that are read/written. QED (Martin and Lam, 2008) is a first model checker for Java web applications that systematically explore sequences of requests to a web application and looks for taint-based vulnerabilities in web applications. Artzi et al. (Artzi et al., 2010) proposed an explicit-state model checker for PHP web applications. The model checker generates test inputs for web applications, monitors the applications for crashes, and validates that the output conforms to the HTML specification. Petrov et al. (Petrov et al., 2012) developed dynamic race detector tool for web applications. It is mainly formulation of a happens-before relation to capture the asynchronous behavior of most commonly-used web platform constructs. In (Gligoric and Majumdar, 2013), the authors proposed an explicit-state model checker DPF for verifying atomicity violations in web applications. The model checker interposes between the program and the database layer and precisely tracks the effects of queries made to the database. Another symbolic moder checker is proposed in (Diana et al., 2012) for the verification of stored procedure and SQL queries w.r.t. their specification. The specification is expressed in CTL temporal logic. In (Scully and Chlipala, 2017), the authors introduced Sqlcache, the automatic complier optimization for database result caching. The main aim of Sqlcache is to analyze web applications for compatibility checking between queries and updates, by instrumenting updates with cache invalidations.

3 MOTIVATING EXAMPLE

Consider the database program depicted in Code Snippet 1. The code implements a module which performs withdrawal of cash from an ATM system.

The function withdraw() updates the balance of authorized customers after satisfying two conditions: (i) maximum withdrawal amount (*amt*) request of the customer should be less than or equal to 10,000 USD for each transaction, and (ii) minimum balance (*minbal*) will be more than or equal to 1000 USD after each successful transaction. Now consider the following integrity constraint on the database attribute *balance*:

> *The minimum account balance of all customers should be 1000 USD.*

Our proposed framework will allow us to verify whether the database program satisfies or violates

Code Snippet 1: A Database Application $\mathbb{PR}$

```
0. int withdraw() {
1.   int acc_no, amt, bal, minbal=1000;
2.   acc_no = read();
3.   amt = read();
4.   Statement con = DriverManager.get
     Connection("jdbc mysql :..... ", " scott ",
     " tiger ") . createStatement () ;
5.   bal = con.executeQuery("select balance form
     Account where Accno=acc_no");
6.   if (amt ⩽ 10000){ // Maximum withdraw amount
7.     if (balance − amt ⩾ minbal){
8.       con.executeQuery("update Account
           balance = balance − amt where Accno =
           acc_no");
           // ... do some work ...
         }
       else
11.      ERROR_msg;
     }
12.  bal = con.executeQuery("select balance form
     Account where Accno=acc_no");
13.  display (bal);
       // ... do some work ...
16.  return 0;
```

such integrity constraints.

4 PRELIMINARIES

In this section, we recall the notions of predicate abstraction, weakest preconditions and boolean program.

Predicate Abstraction (Ball et al., 2001): The predicate abstraction algorithm generates a finite state abstraction from a large or infinite state system. It is an automated abstraction technique in which the abstract domain is constructed from a given set of predicates over program variables. It is used to reduce the state space of a program. The basic idea in predicate abstraction is to remove some variables from the program by just keeping information about a set of predicates about them. Let $\mathfrak{R}$ be a region. Consider the predicate abstraction domain which is parameterized by a fixed finite set Π of first order formulas. The predicate abstraction of $\mathfrak{R}$ with respect to Π is the smallest region $\mathbb{A}(\mathfrak{R}, \Pi)$ which contains $\mathfrak{R}$ and it is representable as a boolean combination of predicates from Π:

$$\mathbb{A}(\mathfrak{R}, \Pi) = \bigwedge \{\psi \mid \psi \text{ is a boolean formula over } \Pi \wedge \mathfrak{R} \Rightarrow \psi\}$$

The region $\mathbb{A}(\mathfrak{R}, \Pi)$ can be computed by recursively splitting as follows:

$$\mathbb{A}(\mathfrak{R}, \Pi) = \begin{cases} true, & \text{if } \Pi = \emptyset \text{ and } \mathfrak{R} \text{ satisfiable} \\ false, & \text{if } \Pi = \emptyset \text{ and } \mathfrak{R} \text{ unsatisfiable} \\ (p \wedge \mathbb{A}(\mathfrak{R} \wedge p, \Pi')) \vee (\neg p \wedge \mathbb{A}(\mathfrak{R} \wedge \neg p, \Pi')), & \text{if } \Pi = \{p\} \cup \Pi' \end{cases}$$

Weakest Preconditions (Ball et al., 2001): Given a program statement *stmt* and a predicate ψ. The weakest precondition of *stmt* with respect to ψ is denoted as WP(*stmt*, ψ) which determines a predicate such that after successfully executing *stmt* on the initial state satisfying ψ results into the final state where ψ is true and *stmt* terminates. For example, given an assignment statement $x = e$,

$$\text{WP}(x = e, \psi) = \psi'$$

where ψ' is obtained by replacing all occurrences of x in it with e, *i.e.* $\psi' \triangleq \psi[e/x]$.

Example 1. *Consider the statement* $stmt \triangleq y = y+1$ *and predicate* $\psi \triangleq y \leqslant 30$. *The weakest precondition of stmt with respect to* ψ *is computed as*

$$\text{WP}(y = y+1, y \leqslant 30) = (y+1) \leqslant 30 = (y \leqslant 29)$$

Observe that, given a statement *stmt* and predicate $\psi \in \Pi$, it may be the case that WP(*stmt*, ψ) is not in Π. For instance, consider $\Pi = \{(x < 5), (x == 2)\}$ and $\text{WP}(x = x+1, x < 5) = (x < 4)$ where the predicate $(x < 4)$ is not in Π. Therefore, the decision procedures (*i.e.* a theorem prover) can be used to strengthen or weaken the resulting precondition in order to express over the predicates in Π (Ball et al., 2001).

Boolean Programs (Ball and Rajamani, 2000): Boolean programs can be thought of as an abstract representation of a program that explicitly captures correlations between data and control, in which boolean variables can represent arbitrary predicates over the unbounded state of a program. The syntax is depicted in Figure 1. Boolean variables are either local or global with statically scoped as C program and variable declarations need not specify a type. A variable identifier is either a C-style identifier or an arbitrary string between the characters "{" and "}". Two constants '0' (*false*) and '1' (*true*) are in the language. The expressions are built from these constants, variables and logical connectives. A parallel assignment statement allows the simultaneous assignment of a set of values to a set of variables. The statements 'if', 'while' and 'assert' can affect

Commands:			
B	::=	V* P*	A list of global variable delectations followed by a list of procedure definitions
V*	::=	id^+ ;	Declaration of variables
P*	::=	id (id^*) begin V* S end	Procedure definition
S	::=	$lstmt^+$	Sequence of statements
lstmt	::=	*stmt* | *id* : *stmt*	
stmt	::=	skip; | $id^+ := exp^+$; | if(*con*) then S else S | while(*con*) do S	
	|	assert(*con*); | $id(exp^*)$; | print(exp^+); | return;	
con	::=	? | *exp*	Non-deterministic choice
exp	::=	*exp op exp* | !*exp* | (*exp*) | *id* | *const*	
op	::=	& | | | $\wedge$ | = | != | $\Rightarrow$	Logical connectives
id	::=	$[a-zA-Z]\,[a-zA-Z0-9]^*$ | {*string*}	
const	::=	0 | 1	False/True

Table 1: Syntax of Boolean Programs (Ball and Rajamani, 2000)

the control flow of the language. Observe that the predicate of control statements is a decider which can be used to model non-determinism. A decider evaluates to '0' or '1' deterministically or '?' which evaluates to '0' or '1' non-deterministically.

5 PROPOSED APPROACH

In this section, we illustrate our proposed model checker which consists of three key modules: (i) abstraction of database programs into their equivalent boolean programs, (ii) automatic VC generation from the Control Flow Graph (CFG) of the boolean programs and its verification using SMT, and (iii) abstraction refinement introducing additional predicates, by identifying spurious execution path in the original database program. Let us explain each of the modules in the following sub-sections:

5.1 PROGRAM ABSTRACTION

This section describes the abstraction of database programs into boolean programs using predicates.

Given a database program $\mathbb{P}$ and a set of predicates $\mathbb{E} = \{\psi_1, \psi_2, \psi_3, ..., \psi_n\}$, let $\mathbb{BP}$ ($\mathbb{P}$, $\mathbb{E}$) generates a boolean program which consists of boolean variables and it has similar control flow as in $\mathbb{P}$. In particular, $\mathbb{BP}$ ($\mathbb{P}$, $\mathbb{E}$) contains n boolean variables $\mathbb{B} = \{b_1, b_2, b_3, ..., b_n\}$ where each boolean variable b_i represents the predicate ψ_i ($1 \leqslant i \leqslant n$). Note that, $\mathbb{BP}$ ($\mathbb{P}$, $\mathbb{E}$) is guaranteed to be an abstraction of $\mathbb{P}$ in the sense that the set of execution traces of $\mathbb{BP}$ ($\mathbb{P}$, $\mathbb{E}$) is a superset of the set of execution traces of $\mathbb{P}$. We now describe an abstraction of various programming constructs of $\mathbb{P}$ with respect to $\mathbb{E}$ below:

Assignments (Ball et al., 2001):

Consider an assignment statement $x = e$ at program point l in $\mathbb{P}$ where e denotes an arithmetic expression. Given a predicate $\psi_i \in \mathbb{E}$, a boolean variable b_i in $\mathbb{BP}$ ($\mathbb{P}$, $\mathbb{E}$) can have the value *true* after l if WP(x = e, ψ_i) = *true* before l. Similarly, b_i can have the value *false* after l if WP(x = e, $\neg\psi_i$) = *false* before l. Observe that, if neither of these predicates holds before l then b_i in $\mathbb{BP}$ ($\mathbb{P}$, $\mathbb{E}$) contains parallel assignment at l as:

$$
\begin{aligned}
b_1 \ldots b_n = \\
&\mathbb{T}_{ch}(WP(x = e, \psi_1), WP(x = e, \neg\psi_1)), \\
&\ldots, \\
&\mathbb{T}_{ch}(WP(x = e, \psi_n), WP(x = e, \neg\psi_n))
\end{aligned}
$$

where the function $\mathbb{T}_{ch}$ is defined as bellow:

```
bool T_ch (bool pos, bool neg) {
  if (pos) {return true; }
  if (neg) {return false; }
  return unknown();  }
```

The unknown function is defined as follows:

```
bool unknown() {
  if (*) {return true; }
  else {return false; }
 }
```

Observe that the unknown function uses the control expression "$*$" which non-deterministically determines the result either *true* or *false*.

Conditional (Ball et al., 2001):

Consider a conditional statement if (ψ) {*stmt*} else {*stmt*} in $\mathbb{P}$. If the predicate ψ is evaluated to *true* in $\mathbb{P}$ then predicate ψ in the corresponding $\mathbb{BP}$ ($\mathbb{P}$, $\mathbb{E}$) should also be evaluated to *true*. Similarly, ψ evaluated to *false* in $\mathbb{P}$ then predicate ψ in the corresponding $\mathbb{BP}$ ($\mathbb{P}$, $\mathbb{E}$) should also be evaluated to *false*. In $\mathbb{BP}$ ($\mathbb{P}$, $\mathbb{E}$), the condition is encoded as:

```
if (*) {
  assume(ψ);
    ...
  } else {
    assume(¬ψ);
      ...
  }
```

Observe that, as "*" in 'if' condition non-deterministically evaluates to either *true* or $false$, both execution paths will be explored. The functions assume(ψ) under 'if' and assume($\neg\psi$) under 'else' preserve the semantics of the conditional statement in $\mathbb{P}$.

Databases Statements:

We now define the abstraction of database statements (SELECT, UPDATE, DELETE and INSERT) into their boolean form. To do so, let us first recall from (Halder and Cortesi, 2012) the formal syntax of database programs as below:

$$
\begin{array}{lll}
Q & ::= & \langle A, \phi \rangle \\
A & ::= & \texttt{select}(v_a,\ f(\vec{e}),\ r(\vec{h}(\vec{x})),\ \phi',\ g(\vec{e})) \mid \\
 & & \texttt{update}(\vec{v_d},\vec{e}) \mid \texttt{insert}(\vec{v_d},\vec{e}) \mid \texttt{delete}(\vec{v_d}) \\
\tau & ::= & n \mid v_a \mid v_d \mid f_n(\tau_1,\tau_2,...,\tau_n), \\
 & & \text{where } f_n \text{ is an n-ary function.} \\
a_f & ::= & R_n(\tau_1,\tau_2,...,\tau_n) \mid \tau_1 = \tau_2, \\
 & & \text{where } R_n(\tau_1,\tau_2,...,\tau_n) \in \{true, false\} \\
\phi & ::= & a_f \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \forall x_i\ \phi \mid \exists x_i\ \phi \\
c & ::= & Q \mid v = e \mid \text{if } cond \text{ then } c_1 \text{ else } c_2 \\
 & & \mid \text{while } cond \text{ do } c
\end{array}
$$

Where ϕ denotes an well-formed first order formula defined over constants (n), application variables (v_a) and database attributes (v_d). The SQL clauses GROUP BY, ORDER BY, DISTINCT/ALL and the aggregate functions (SUM, COUNT, MAX, MIN, AVG) are represented in the form of functions $g()$, $f()$, $r()$, $h()$ respectively parameterized with either none or one arithmetic expression e or an ordered sequence of arithmetic expressions $\vec{e}$. The abstract syntax of a database statement is denoted by $\langle A, \phi \rangle$ where A represents Action-part and ϕ represents Condition-part. The Action-part include SELECT, UPDATE, DELETE and INSERT. For example, consider the query Q="update t set sal:=sal+100 where $age \geqslant 40$". According to abstract syntax, Q is denoted by $\langle A,\ \phi \rangle = \langle \texttt{update}(\vec{v_d},\vec{e}),\ \phi \rangle$, where $\vec{v_d} = \langle sal \rangle$ and $\vec{e} = \langle sal{+}100 \rangle$ and ϕ=$age \geqslant 40$.

Given a set of predicate $\mathbb{E}$ consisting of the integrity constraints under consideration, the abstraction of database statements into their equivalent boolean form involves the following two major tasks:

(1) *Computation of weakest precondition of database statements w.r.t.* $\psi \in \mathbb{E}$: We define below the computation of weakest precondition for various database statements for a given post condition.

$$\text{WP}(\langle \texttt{select}(v_a,\ f(\vec{e}),\ r(\vec{h}(\vec{x})),\ \phi',\ g(\vec{e})),\ \phi \rangle,\ \psi) = \psi$$
$$\text{WP}(\langle \texttt{update}(\vec{v_d},\vec{e}),\phi \rangle,\psi) = ((\psi \wedge \neg\phi) \vee (\psi[\vec{e}/\vec{v_d}] \wedge \phi))$$
$$\text{WP}(\langle \texttt{insert}(\vec{v_d},\ \vec{e}),\ false \rangle,\ \psi) = \psi[\vec{e}/\vec{v_d}] \vee \psi$$
$$\text{WP}(\langle \texttt{delete}(\vec{v_d}),\ \phi \rangle,\ \psi) = \psi$$

(2) *Conversion of database statements into boolean form*: Given a database statement Q and a postcondition ψ, suppose WP(Q, ψ)=ψ'. Let us now define $\mathbb{BP}$ for database statements to convert into their equivalent boolean form w.r.t. ψ' below:

$$\mathbb{BP}(\langle \texttt{select}(v_a,\ f(\vec{e}),r(\vec{h}(\vec{x})),\phi',g(\vec{e})),\phi \rangle,\psi') = \text{skip}$$

$$\mathbb{BP}(\langle \texttt{update}(\vec{v_d},\vec{e}),\phi \rangle,\psi') = \begin{cases} b = \phi\ ?\ * : true, & \text{if } \psi' \in \mathbb{E} \\ \text{skip}, & \text{otherwise} \end{cases}$$

$$\mathbb{BP}(\langle \texttt{insert}(\vec{v_d},\ \vec{e}),\ false \rangle,\ \psi') = \begin{cases} b = *, & \text{if } \psi' \in \mathbb{E} \\ \text{skip}, & \text{otherwise} \end{cases}$$

$$\mathbb{BP}(\langle \texttt{delete}(\vec{v_d}),\ \phi \rangle,\ \psi') = \text{skip}$$

Since SELECT and DELETE operations do not violate any integrity constraints defined over database states at row level, the corresponding statements are replaced by 'skip'. On the other hand, in case of UPDATE and INSERT, the following two cases may arise: (i) if ψ' already exists in $\mathbb{E}$ and ϕ is evaluated to *true*, this means that the update operation is taking place which may lead the updated values possibly to violate the integrity properties. Therefore, the corresponding boolean variable b in the boolean program is assigned to '*'. Otherwise, the evaluation of ϕ to $false$ indicates no update operation (hence, no property violation) and therefore b is assigned to *true*, and (ii) if ψ' does not exist in $\mathbb{E}$ (with or without applying strengthen or weaken operation) then we replace the given database statement by "skip".

Let us illustrate this using an example as follows.

Example 2. *Let* $\mathbb{E}$=$\{sal \leq 8000\}$ *and its corresponding boolean variable be b. We assume that the initial database is in consistent state, and therefore, the boolean variable b is set to true.*

Consider the database table Emp *which stores employees information and the following* update *statement:*

*Q=*update Emp set *sal=sal+100* where *age>30*

The abstract syntax of Q is represented as

$$Q = \langle \texttt{update}(\langle sal \rangle, \langle sal{+}100 \rangle), age > 30 \rangle$$

Code Snippet 2: Boolean Program of $\mathbb{PR}$

```
0. int  withdraw() {
     bool {bal ⩾ minbal} // b := bal ⩾ minbal
1.   skip;
2.   skip;
3.   skip;
4.   skip;
5.   skip;
6.   if (∗){
7.       if (∗){
8.           b = ϕ ? ∗ : true // ϕ = Accno=acc_no
               }
         else{
11.          skip;
              }
            }
12.  skip;
13.  skip;
16.  skip; }
```

The weakest precondition of Q w.r.t. "sal $\leq$ 8000*" is* WP($\langle$`update`$(\langle sal\rangle, \langle sal$+100$\rangle), age > 30\rangle$, $sal \leq 8000$) = $((sal \leq 8000 \wedge \neg(age > 30)) \bigvee (sal + 100 < 8000 \wedge age > 30))$. *Assuming age* $>$ 30 *evaluates to true, we get sal*+100 $<$ 8000 *which results sal* $<$ 7900. *Observe that the resulting precondition sal* $<$ 7900 *does not exist in* $\mathbb{E}$. *Therefore, after applying strengthen operation, b is set to "*".*

Illustration on motivational example: Let us consider the motivating example in code snippet 1. Consider $\mathbb{E} = \{bal \geqslant minbal\}$ taking the integrity constraint under consideration and its corresponding boolean variables set $\mathbb{B} = \{b\}$. The boolean program of $\mathbb{PR}$ with respect to $\mathbb{E}$ is shown in the code snippet 2. Observe that the program statements 1, 2, 3, 4, 5, 11, 12, 13 and 16 in the boolean program are replaced by 'skip' because they do not affect the predicate set $\mathbb{E}$. On other hand, the conditions in the statements at program points 6 and 7 are set to "$*$". Note that, due to the absence of the predicates "$amt \leqslant 10000$" and "$balance\text{-}amt \geqslant minbal$" in $\mathbb{E}$, there is no assume statement included in the boolean code. Program statement 8 denotes the boolean abstraction of the `update` statement.

5.2 VERIFICATION

In this section, we describe the generation of verification conditions and their satisfiability. After converting the program into a boolean form, we will verify whether all possible executions paths respect the defined properties. For this objective, we perform the below steps:

- Control Flow Graph (CFG) construction of boolean programs.
- VC generation from CFGs of boolean programs.
- Satisfiability checking of the VCs using SMT.

The Control Flow Graph G_{BL} of boolean program $\mathbb{BL}$ is constructed by following the similar approach proposed in (Ball and Rajamani, 2000). After constructing $\mathbb{G}_{BL}$, we list all possible execution paths. We convert each path p_i into a verification formula f_i by anding all boolean statements encountered along that path. The generated f_i is suppled to SMT solver for the satisfiability checking. If the formula f_i is satisfied, then the corresponding path p_i satisfies the property and the algorithm terminates. Otherwise, we check whether the corresponding path p_i is a feasible path in the original program. If yes, we conclude a violation of the property and an error trace is generated. Otherwise, a refinement of the boolean program is performed which we describe in the next section.

Illustration on motivational example: Consider the boolean program in the code snippet 2. Its CFG is depicted in Figure 1. Consider the CFG path $p \triangleq 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 12 \rightarrow 13 \rightarrow 16$ (denoted by blue color). The generated VC along this path is $b = \phi$? $*$: $true$. This is encoded as $f \triangleq (b \wedge \phi) \vee (\neg b \wedge \phi) \vee (b \wedge \neg\phi) \Rightarrow b == true$. We pass the negation of this f (*i.e.* $\neg f$) to SMT solver which reports "satisfied". This shows that there exist some solution for which the predicate b evaluates to *false*. Therefore, the model checker indicates that the program $\mathbb{PR}$ may violate the integrity constraint. However, observe that this path is not a feasible path in $\mathbb{PR}$, because b which corresponds to $bal \geqslant minbal$ is *false*. This indicates that the abstraction is not precise enough and there is a need of refinement.

5.3 COUNTER EXAMPLE GUIDED ABSTRACT REFINEMENT

The process of refining the abstraction is done using a method called counterexample-guided abstraction refinement (CEGAR) (Wang et al., 2006). If the boolean program contains an error path p_i and this path is not feasible in the original program, then the refinement of the Boolean program will be initiated to eliminate this false error paths. More specifically, we determine suitable predicates, by analyzing the infeasibility of paths in the original database program which illustrates feasible paths in the corresponding boolean program. We then refine the abstraction by

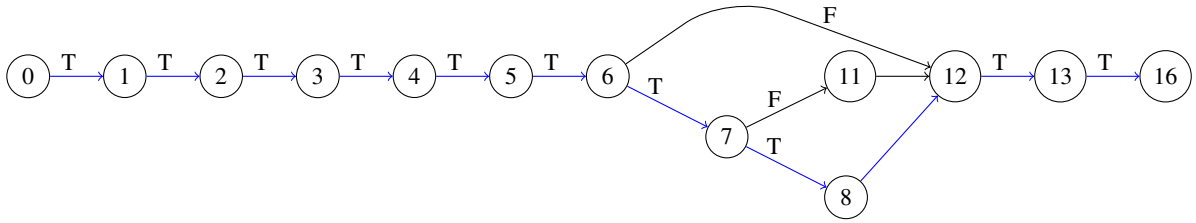

Figure 1: CFG of the Boolean Program in Code Snippet 2

adding these new predicates to $\mathbb{E}$. Let us illustrate below the application of CEGAR on the motivating example:

Illustration on motivational example: Consider the boolean program in code snippet 2. As we already observed in section 5.2 that this abstraction is not precise enough, in order to refine this abstraction we discover the predicate ($amt \leqslant 10000$) form $\mathbb{PR}$ and we add it to $\mathbb{E}$. This results $\mathbb{E} = \{bal \geqslant minbal, amt \leqslant 10000\}$ and $\mathbb{B} = \{b, b_1\}$ where b and b_1 corresponds to $bal \geqslant minbal$ and $amt \leqslant 10000$ respectively. Given the refined boolean program w.r.t. these new $\mathbb{E}$ and $\mathbb{B}$, observe that after verifying, by following the same procedure as before, we get the error path $p_1 \triangleq 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 12 \rightarrow 13 \rightarrow 16$ which is also not a feasible path in $\mathbb{PR}$ because the predicate ($amt \leqslant 10000$) is always *true* along the path. By initiating the refinement process once again, we discover a new predicate ($balance$ - $amt \geqslant minbal$). Now considering the new $\mathbb{E} = \{bal \geqslant minbal, amt \leqslant 10000, balance$ - $amt \geqslant minbal\}$ and $\mathbb{B} = \{b, b_1, b_2\}$, we get a refined boolean program depicted in code snippet 3. We now generate the VC from the CFG of this code snippet 3, which is encoded as "$f_2 \triangleq ((b_1 == true) \wedge (b_1 == true \wedge b == true) \wedge (b_2 == true \wedge b_1 == true \wedge b == true) \wedge ((b \wedge \phi) \vee (\neg b \wedge \phi) \vee (b \wedge \neg\phi) \Rightarrow b == true)) \Rightarrow b == true$". The SMT solver reports unsatisfied when $\neg f_2$ is passed as input. Therefore, the model checker indicates that $\mathbb{PR}$ is safe w.r.t. the integrity constraint $bal \geqslant minbal$.

Code Snippet 3: Refined Boolean Program of $\mathbb{PR}$

```
0. int withdraw() {
   bool {bal ⩾ minbal, amt ⩽ 10000, balance − amt ⩾
        minbal } // b := bal ⩾ minbal, b1 := amt ⩽
        10000, b2 := balance − amt ⩾ minbal
1.  skip;
2.  skip;
3.  skip;
4.  skip;
5.  skip;
6.  if (∗){
     assume(amt ⩽ 10000) // b1 = true
7.    if (∗){
        assume(balance − amt ⩾ minbal) // b2 = true
8.      b = ϕ ? ∗ : true; // ϕ = Accno = acc_no
          }
      else {
            assume(¬(balance − amt ⩾ minbal))
11.         skip;
          } }
12. skip;
13. skip;
16. skip; }
```

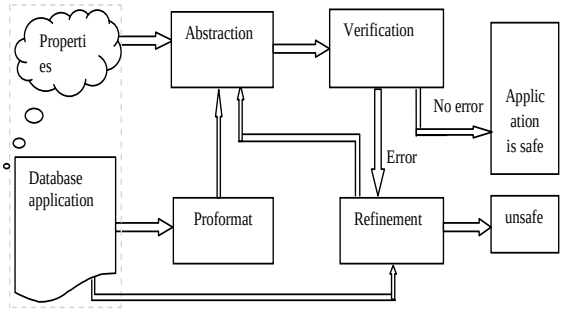

Figure 2: Overall Architecture of Database Model Checker

6 TOWARDS IMPLEMENTATION

In general, the proposed framework accepts as inputs a database program and properties of interest, and this verifies whether the input program respects the properties by generating safe/unsafe message as output. The overall architecture of our proposed framework is shown in Figure 2. We have identified the following key modules which play important roles in implementing the proposed framework:

1. **Proformat**: This modules preprocess input database programs and annotates them by adding line numbers to all statements.
2. **Abstraction**: The module "Abstraction" abstracts the database program into a boolean program using a set of predicates.

3. **Verification**: This module at first constructs CFG of the boolean program. After that, it generates VC from CFG. Finally, it tests the satisfiability of each VC using SMT solver.
4. **Refinement**: The primary task of this module is to discover additional predicates which refine the abstraction avoiding the existence of the spurious paths in the corresponding boolean database programs.

7 CONCLUSIONS AND FUTURE WORKS

In this paper, we proposed a symbolic model checking algorithm for verifying the correctness of database applications with respect to integrity properties. We design our model checker based on the following key modules: (i) Abstraction, (ii) Verification and (iii) Refinement. We are currently implementing our proposed model checker, as per the description provided in the tool architecture, in a modular way to support scalability.

Acknowledgement

This work is partially supported by the research grant (SB/FTP/ETA-315/2013) from the Science and Engineering Research Board (SERB), Department of Science and Technology, Government of India.

REFERENCES

Anand, S., Păsăreanu, C. S., and Visser, W. (2007). Jpf–se: A symbolic execution extension to java pathfinder. In *TACAS*, pages 134–138. Springer.

Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., and Ernst, M. D. (2010). Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE TSE*, 36(4):474–494.

Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. (2001). Automatic predicate abstraction of c programs. In *ACM SIGPLAN Notices*, volume 36, pages 203–213. ACM.

Ball, T. and Rajamani, S. K. (2000). Bebop: A symbolic model checker for boolean programs. In *International SPIN Workshop on Model Checking of Software*, pages 113–130. Springer.

Ball, T. and Rajamani, S. K. (2002). The s lam project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM.

Chaki, S., Clarke, E., Groce, A., Ouaknine, J., Strichman, O., and Yorav, K. (2004). Efficient verification of sequential and concurrent c programs. *Formal Methods in System Design*, 25(2-3):129–166.

Chandra, S., Godefroid, P., and Palm, C. (2002). Software model checking in practice: an industrial case study. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd IC on*, pages 431–441. IEEE.

Clarke, E., Kroening, D., and Yorav, K. (2003). Behavioral consistency of c and verilog programs using bounded model checking. In *Proc. of the 40th annual Design Automation Conference*, pages 368–371. ACM.

Clarke, E. M. and Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer.

Diana, R., Marques-Neto, H., Zarate, L., and Song, M. (2012). A symbolic model checking appproach to verifying transact-sql. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pages 1735–1741. IEEE.

Gligoric, M. and Majumdar, R. (2013). Model checking database applications. In *IC on Tools and Algorithms for the Construction and Analysis of Systems*, pages 549–564. Springer.

Halder, R. and Cortesi, A. (2012). Abstract interpretation of database query languages. *Computer Languages, Systems & Structures*, 38:123–157.

Holzmann, G. J. (1997). The model checker spin. *IEEE TSE*, 23(5):279–295.

Ivancic, F., Yang, Z., Ganai, M. K., Gupta, A., Shlyakhter, I., and Ashar, P. (2005). F-soft: Software verification platform. In *IC on Computer Aided Verification*, pages 301–306. Springer.

Jhala, R. and Majumdar, R. (2009). Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):21.

Martin, M. and Lam, M. S. (2008). Automatic generation of xss and sql injection attacks with goal-directed model checking. In *Proc. of the 17th conference on Security symposium*, pages 31–43. USENIX Association.

Musuvathi, M. and Qadeer, S. (2007). Iterative context bounding for systematic testing of multithreaded programs. In *ACM Sigplan Notices*, volume 42, pages 446–455. ACM.

Paleari, R., Marrone, D., Bruschi, D., and Monga, M. (2008). On race vulnerabilities in web applications. In *IC on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 126–142. Springer.

Petrov, B., Vechev, M., Sridharan, M., and Dolby, J. (2012). Race detection for web applications. In *ACM SIGPLAN Notices*, volume 47, pages 251–262. ACM.

Queille, J.-P. and Sifakis, J. (1982). Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351.

Scully, Z. and Chlipala, A. (2017). A program optimization for automatic database result caching. *ACM SIGPLAN Notices*, 52(1):271–284.

Wang, C., Hachtel, G. D., and Somenzi, F. (2006). *Abstraction refinement for large scale model checking*. Springer Science & Business Media.

Yang, J., Twohey, P., Engler, D., and Musuvathi, M. (2006). Using model checking to find serious file system errors. *ACM Trans. CS (TOCS)*, 24(4):393–423.