Science Research Annals
A Scholarly Peer-reviewed Research Journal
Volume 14, No 1, Pages 28- 41, 2023
Available online at: http://journals.aaua.edu.ng/sra
ISSN: 2805 - 3508

Faculty of SCIENCE AAUA

A Survey of Comparative Studies of Symbolic Algorithms

E. O. ALIYU*

Department of Computer Science, Adekunle Ajasin University, Akungba–Akoko, Ondo State, Nigeria

*Corresponding author E-mail address: olubunmi.aliyu@aaua.edu.ng

A b s t r a c t

Symbolic verification is one of the most important techniques used in formal software development. Many approaches have been introduced into this research area, but the problem of state explosion cannot be over emphasized. Therefore, in this article, the use of qualitative analysis (Table-driven) to describe some techniques used for symbolic algorithms and to identify knowledge gap in research for future research questions is presented. Bases of comparison include state space, time complexity, model checking methods, data intensive operations, state explosion problem, class of properties (invariant) and language specification respectively on symbolic algorithms such as Symbolic Model Verifier (SMV), Cadence SMV, A New (NuSMV), Construction Analysis Distributed Processes (CADP), ALLOY, Vienna Development Method Specification Language Analyzer (VDM-SL), TLA+ Model Checker (TLC) and Failure Divergence Refinement (FDR) were examined. From this qualitative analysis investigations, ALLOY algorithm established the best optimal symbolic states exploration in terms of low rate of state explosion problem. It contributes to knowledge by improving the scalability and efficiency of verification, providing insights into symbolic analysis and impacting academic research.

Keywords:

Symbolic Model
Verifier, Language Analyzer
Temporal Logic
Of Action,
Alloy Model Checker

Accepted:27 June, 2023

INTRODUCTION

A symbolic verification is a type of algorithm that operates on symbols, rather than specific numerical values (Yang *et al.*, 2006; Yu *et al.*, 2020) Typical example are: symbolic model checking, symbolic execution, SAT solvers. It manipulates symbolic representations of mathematical expressions or logical formulas to perform computations or solve problems. Instead of working with concrete values, symbolic algorithms work with variables, constants, and operators, allowing for the representation and manipulation of abstract mathematical or logical concepts.

This article focuses on symbolic model checking; a technique used to verify the correctness of systems by representing their behaviours symbolically, rather than exploring concrete executions as regard to explicit method resulting to state explosion problem (Deharbe 2003; Souri *et al.*, 2019). The symbolic model checking technique was developed in 1987 according to McMillan (1992) to alleviate the problem, directed toward proving temporal properties of finite state systems using ordered binary decision diagram (OBDD) developed by Bryant in 1986 to represent sets of states and transitions (Clarke and Lerda 2007; Padon et al., 2021).

The work of (McMillan, 1992) over the years led to the creation of symbolic model checking algorithms which has produced significant result in the field. Therefore, this article uses qualitative analysis to categorize

symbolic algorithms theme as well as identifying research gap for future research in this area.

Section two describes the basic algorithm for symbolic model checking as well as the groundbreaking work in this area. Section 3 presents comparative study of the symbolic algorithms. Section 4 discusses the findings of the table-driven approach while conclusion is in section 5.

SYMBOLIC MODEL CHECKING ALGORITHMS

Symbolic model checking problem can be stated as an ordered decision tree for the function $a\wedge b\vee c\wedge d$ and the requirement specification f is a linear tree logic (LTL) formula, determine whether the system model satisfies all the specified properties; $(a\wedge b)\vee(c\wedge d)$.

Mathematically, a reduced ordered binary decision diagram (ROBDD) (a Boolean encoding for set of states $a\wedge b\vee c\wedge d$) can be defined as follows:

Definition:

Let B be a set of Boolean variables $\{x_1, x_2, \ldots, x_3\}$. A Reduced Ordered Binary Decision Diagram (ROBDD) is a directed acyclic graph (DAG) $G = (V, E, \omega, v_0, v_1)$ with the following properties:

- Vertex Set (V): The set of vertices V is a set of nodes, where each node represents a Boolean variable x_i or a constant value (0 or 1). There are two special nodes: the terminal nodes v_0 and v_1 representing the constant values 0 and 1, respectively.

- Edge Set (E): The set of edges E is a set of directed edges connecting the nodes in V. Each edge is labeled with a Boolean variable x_i or a constant value (0 or 1). An edge from node u to node v labeled with variable x_i indicates that the value of x_i is used to determine the path from u to v.

- Variable Ordering (ω): The variable ordering ω is a total order of the Boolean variables in B, specifying a total order on the nodes representing these variables in V. The variable ordering ensures that the paths in the ROBDD are structured in a way that allows for efficient manipulation and sharing of common subgraphs.

- Terminal Nodes: The terminal nodes v_0 and v_1 represent the constant values 0 and 1, respectively. They have no outgoing edges.

Reduced Structure: The ROBDD has a reduced structure, meaning that isomorphic subgraphs are merged into a single node. This reduction is achieved by applying various optimization techniques, such as Shannon expansion, variable elimination, and variable substitution (McMillan, 1992).

Symbolic model checking provides a systematic and automated approach for verifying complex system models against desired properties, allowing for thorough analysis and detection of potential errors or violations. BDD and their variants is presented below:

i. **Colorado University Decision Diagrams (CUDD):** It is a C/C++ library for creating different types of decision diagrams such as Binary Decision Diagrams (BDDs), Algebraic Decision Diagrams (ADDs), and Zero-suppressed Decision Diagrams (ZDD) (Somenzi, 2018).

ii. **Binary Decision Diagram Package (BuDDY):** BuDDy is a powerful library for creating and manipulating BDD. It is implemented in C and also offer a C++ interface (Toussi and Bigham, 2014).

iii. **Reduced Ordered Binary Decision Diagram (ROBDD)**: It has been explained above. The advantage of a ROBDD is that it is canonical (unique representation) for a particular function and variable order.

iv. **Ordered Binary Decision Diagram (OBDD):** OBDDs (Meinel and Theobald, 1998; Wegener, 2004) is a read-once branching program with an additional ordering restriction on the variables.

v. **Binary Moment Diagrams (BMD):** BMD represent a given integer function using the arithmetic transform expansion. In Sasaso and Nagayama (2006), elementary functions such as polynomial, trigonometric, logarithmic, square root, and reciprocal functions (real value function) are converted into integer functions.

vi. **Zero-suppressed Decision Diagram (ZDD):** This is a data structure to represent objects that typically contain many zeros. This data structure can be used for combinatorial problems, such as graphs, circuits, faults, and data mining (Sasao and Butler, 2014).

vii. **Free Binary Decision Diagram (FBDD):** These are the most general BDDs without inconsistent paths.

viii. **Parity Binary Decision Diagram (PBDD):** This is a data structure for representing Boolean functions that has an odd number of ones.
ix. **Multiple Terminal Binary Decision Diagram (MTBDD):** This is a procedural representation of multi-output function (representation of a system of Boolean functions by a truth table) in a tree-like algorithmic specification (Levin and Keren, 2008).
x. **Multi-valued Decision Diagram (MDD):** It is also known as multi-way decision diagram. It represents functions on integer domains (Dijk and Pol, 2016).

In essence, this article realizes that many problems can be formulated in terms of Boolean functions in various fields, which enable creation of different variant of BDD. They are useful because they represent a state space of sets of states and transitions in symbolic model checking.

This paper also studies other reengineering and extension of SMV like Cadence SMV, a New Symbolic Model Verifier (NuSMV) Cimatti et al., (2000), Construction Analysis Distributed Processes (CADP), ALLOY, Vienna Development Method Specification Language Analyser (VDM-SL) and Failure Divergence Refinement (FDR). However, survey the literature on SAT-based Bounded Model, SMT-based bounded Model Checking among many others, which was introduced to alleviate state explosion problem.

The next section presents background theories and its techniques.

Bounded Model Checking (BMC)

Bounded Model Checking (BMC) based on SAT/SMT Solvers: SAT (Boolean satisfiability) and SMT (satisfiability modulo theories) solvers is one of the most important techniques for model verification system (Cordeiro *et al.*, 2009; Li *et al.*, 2010).

SAT-based Bounded Model Checking (BMC), model system behaviour using propositional logic in conjunctive normal form (CNF) while SMT-based model system using propositional logic with background theories (Biere and Kroning 2018; Abraham and Kremer, 2017).

The basic idea of BMC according to Cordeiro *et al.*, (2009) is to explore all possible behaviours of a program within a given bound to check for property violations. For instance, given a model M, a property in conjunctive normal form $\wedge\varphi$, and a bound length k, the BMC unrolls the system k times and translates it into a verification condition ψ such that ψ is satisfiable if and only if $\wedge\varphi$ has a counterexample of depth less than or equal to k. Carnegie Mellon University (CMU) version of Symbolic Model Verifier (SMV) SAT checkers utilized in Clarke *et al.*, (2003) can be used to check if ψ is satisfiable.

In order to cope with very large constraint and make symbolic execution possible, SMT (Satisfiability Modulo Theories) solvers can be used.

Satisfiability Modulo Theories (SMT) Background Theories

The notion behind background theories is to solve constraint-satisfaction problems in different application areas such as scheduling, planning and resource allocation, just to mention a few. De Moura and Bjorner (2011) pointed out that propositional satisfiability is a commonly known constraint satisfaction problem, aiming to decide whether a formula over Boolean variables using logical connectives can be made true by choosing true/false values for its variables. Also, that problems can be described using arithmetic formula. In that case, an arithmetic theory is required to capture the meaning of the formula. Hence, solvers for such formulation are regarded as satisfiability modulo theories (SAT).

According to Abraham and Kremer, (2017) SMT-solving has two different approaches, namely: (Eager SMT-solving and SAT solver). Eager SMT-solving, often used for bit vectors or uninterpreted functions. This technique aggressively removes theoretical constraints by transforming the formula into a logical propositional formula that equates to satisfaction. SAT solver can then be used to check the satisfaction of the formula. Another approach, called lazy SMT-solving, relies on the interaction of a SAT solver and one or more theoretical solvers as shown in Figure 1.

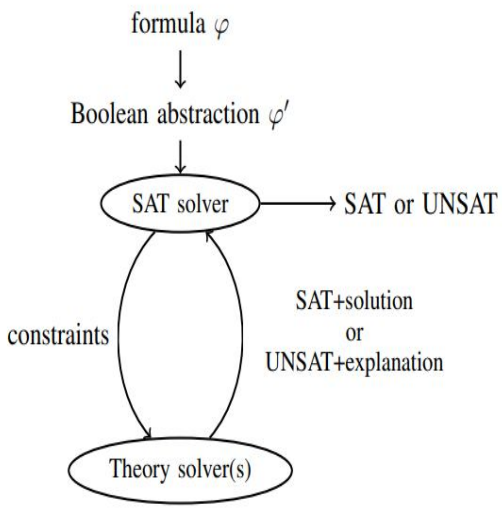

Figure 1: Lazy SMT Solving (Abraham and Kremer, 2017

The input formula φ in Figure 1 is computed into Boolean abstraction φ' by replacing every theory constraint c by a new proposition x_c. A SAT solver is used to compute a solution for the abstraction φ'. The theory solver is initialized to decide whether this set of theory constraints is consistent in the theory and reported φ as satisfiable. Otherwise, the abstraction is refined and SAT solver searches for further solutions.

A typical example of SMT solver background theories found in Cordeiro *et al.,* (2009) for the theory of linear, non-linear and bit-vector arithmetic in Backus Naur Form (BNF) grammar is given below:

$$
\begin{array}{rcl}
Fml & ::= & Fml\ con\ Fml \mid \neg Fml \mid Atom \\
con & ::= & \wedge \mid \vee \mid \oplus \mid : \mid \Leftrightarrow \\
Atom & ::= & Trm\ rel\ Trm \mid Id \mid true \mid false \\
rel & ::= & < \mid \leq \mid > \mid \geq \mid = \mid \neq \\
Trm & ::= & Trm\ op\ Trm \mid Const \mid Id \mid Extract\,[i,j] \\
 & & \mid SignExt\,[k] \mid ZeroExt\,[k] \\
 & & \mid ite\,(Fml,\ Trm,\ Trm) \\
op & ::= & +_{o,u} \mid -_{o,u} \mid *_{o,u} \mid /_{o} \mid rem \\
 & & \mid << \mid >> \mid \& \mid \mid \mid \oplus \mid @
\end{array}
$$

where Fml depicts Boolean-valued expressions, Trm depicts integers, reals and bit-vectors while op denotes binary operators. The semantics of the relational operators$(<, \leq, >, \geq)$, the non-linear arithmetic operators $(*, /, rem)$ and the right-shift operator $(>>)$ depends on whether the program variables are unsigned or signed bit-vectors, integers or real numbers. The expression $Extract[i, j]$ denotes bit-vector extraction from bits i to j while $@$ denotes the concatenation of the given bit-vectors. Atoms is any sequences of character in the set integers, reals and bit-vectors.

Related Works in SMT-solving

Merz *et al.*, (2012) introduced Bounded Model Checking of C and C++ programs using a compiler Intermediate Representation. It made efforts to handle the complexity in the syntax and semantics of these programming languages with the development of Low Bounded Model Checking (LLBMC) for C and C++ programming languages. They used Low Level Virtual Machine (LLVM) complier front-end such as clang or llvm-gcc to translate C and C++ programs into LLVM's Intermediate Representation (LLVM-IR); usually in static single assignment (SSA) form, LLVM-IR program is then converted into LLBMC's internal logical representation (ILR). The ILR formula is simplified by LLBMC using rewrite rules (such as constant propagation or simple arithmetical and logical properties) before being passed to Satisfiability Modulo Theory (SMT) solver (Boolector) for the logic of bit-vectors and arrays. If the SMT solver finds a satisfying assignment (corresponding to a bug in the program), then a counterexample is generated first on the ILR level and then on the LLVM-IR level. Also, LLBMC used bit-precise memory model to re-interpret casts as used in C and static/dynamic casts as used in C++. To evaluate LLBMC's performance, two other BMC tools are compared: The C Bounded Model Checker (CBMC) and the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) 1.16. The result shown that LLBMC performed favourably compare to CBMC and ESBMC in terms of detecting an error such as arithmetic overflow, invalid memory allocations, invalid memory access, invalid free and memory leak detection.

Li (2013), presented Symbolic Abstraction with SMT Solver, the author made effort to perform symbolic abstraction with Symba using the power of Satisfiability Modulo Theory (SMT) solvers to compute the most precise abstraction of Quantifier Free Linear Real Arithmetic (QF-LRA) formula in the general Template

Constraint Matrix (TCM) domain with the following objectives: (i) Implementing precise abstract transformer over loop free program fragment described as QF-LRA formula (ii) Symbolically and efficiently enumerating program paths (iii) enabling invariant generation in TCM domain. Li employed symbolic abstraction algorithm called symba using the power of SMT solver (Z3), encoded all execution as a formula in QF-LRA, a commonly used theory for encoding program execution defined as follows:

$$\varphi \in \mathcal{L} ::= true \mid false \mid P \wedge P^1 \mid P \vee^1$$

$$P, P^1 \in Atoms ::= c_1 x_1 + \cdots c_n x_n \leq k, n \in \mathbb{N}$$

$$x_i \in Vars ::= \{x_i \ldots x_n\}$$

The TCM domain is a numerical abstract domain parameterized by a set of linear terms T denotes templates of the form $c_1 x_1 + \cdots c_n x_n$, where $c_i \in \mathbb{R}$ and $Vars = \{x_1 \ldots x_n\}$

are real valued variables. A set of template T gives rise to the domain $\mathcal{A}_T$ where each

$A \in \mathcal{A}_T$ is a vector of constants $k = (k_1, \ldots k_{|T|})$ with $k_i \in \mathbb{R} \cup \{\infty, -\infty\}$.

Then, formalize the relationship between the TCMs and $\mathcal{L}$ formulas using Galois connections

$C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}_T$ between the concrete domain C and the abstract domain $\mathcal{A}_T$. Symba computed the best abstraction of a $\mathcal{L}$ formula in TCM domain. The symbolic abstraction was implemented with Symba and integrated it as an abstract domain in UFO; a C analysis and verification framework built in the LLVM compiler infrastructure, applied to a large number of programs from the software verification competition with a total of 2.7MLoC and compared with standard abstract interpretation technique. From the experiment carried out, Symba produced more precise invariants and proved 47 programs safe out of 373 programs compared to standard interpretation techniques such as interval abstraction. This technique only generated loop invariant in the general Template Constraint Matrix domain.

Noureddine and Zaraket (2016), presented "Model Checking Software with First Order Logic Specification Using And-Inverter-Graph (AIG) Solvers". They addressed the issue of static verification techniques of Boolean formula satisfiability solvers such as SAT and SMT solvers that operate on conjunctive normal form and first order logic formulae respectively to validate programs with the development of AIG encoding the validation problem of a program. The method took an imperative program S written in J; a subset of C++/Java that includes integers, arrays, loops, and recursion with a first order logic specification consisting of a precondition and a post condition pair (P, Q), within a bound on the program variable denoted by b and specification variables $(S \vDash (P, Q)|_b)$. They translated the validation problem $(S \vDash (P, Q)|_b)$ into AIG; a sequential circuit whose gates are restricted to NAND gates. The procedure took place in the ABC; a transformation-based verification framework to reduce the generated AIG using AIG synthesis reduction and abstraction techniques embedded in the framework. Then, ABC verification techniques was used to decide the satisfiability of the designated output. Benchmark verification tasks from SVComp; a systematic comparative evaluation of the effectiveness and efficiency of the state of the art in software verification was adopted to evaluate the correctness, effectiveness and efficiency of $\{P\}S\{Q\}$ and then, compared its performance with SVComp leading tools such as CBMC, ESBMC, BLAST and CPAChecker. From the experiment carried out, an automated script {P}S{Q} reported no wrong results while CBMC, ESBMC and BLAST reported wrong results. CPAChecker reported no wrong results but reported runtime errors.

TLA+ Model Checker (TLC)

TLA checker (TLC) leverage symbolic representations, such as BDDs (Binary Decision Diagrams) and SAT solvers, for effective storage and manipulation of system states. It combines symbolic techniques with explicit state enumeration to provide thorough model checking capabilities. Temporal Logic of Action (TLA+) is a formal specification language for specifying system behavior and properties, developed by Leslie Lamport in 2002.

PlusCal is a high-level language that serves as a "pseudocode" for specifying TLA+ systems. Algorithm

1 shows the PlusCal code for the simple counter system as follows:

Algorithm 1: PlusCal PCode for Counter System

```
algorithm Counter {
variables counter = 0;
process Increment {
  while (TRUE) {
    with (counter < 10) do {
      counter:= counter + 1;
    }
  }
}
process Decrement {
  while (TRUE) {
    with (counter > 0) do {
      counter:= counter - 1;
    }
  }
}
}
```

In this PlusCal code, we define an algorithm named "Counter" that consists of two processes: "Increment" and "Decrement". Each process continuously executes its respective action while maintaining certain conditions.

- "Increment" process increments the counter by 1 as long as the counter value is less than 10. The with statement ensures that the condition counter < 10 is satisfied before executing the action counter := counter + 1.
- "Decrement" process decrements the counter by 1 as long as the counter value is greater than 0. The with statement ensures that the condition counter > 0 is satisfied before executing the action counter := counter - 1.

Both processes run indefinitely (while (TRUE)) to capture the continuous behavior of the counter system.

Algorithm 2 depicts the translation of the PlusCal code into TLA+ as follows:

Algorithm 2: PlusCal PCode Translation into TLA+

```
MODULE Counter
EXTENDS Naturals

VARIABLE counter

Increment ==
  /\ counter < 10
  /\ counter' = counter + 1

Decrement ==
  /\ counter > 0
  /\ counter' = counter - 1

Next == Increment \/ Decrement

Spec == Init /\ [][Next]_counter

Init == counter = 0
```

In the TLA+ translation, we use the same variable counter to represent the state of the counter system.

- The Increment definition captures the condition for incrementing the counter: the current counter value is less than 10, and the next state is defined as counter = counter + 1.
- The Decrement definition captures the condition for decrementing the counter: the current counter value is greater than 0, and the next state is defined as counter = counter - 1.
- The Next definition combines the Increment and Decrement actions using the disjunction operator ∨, representing the possible next-state transitions.
- The Spec definition represents our system specification, stating that the initial condition (Init) must hold, and in all subsequent states, the Next relation must hold ([] denotes "always" or "invariant").
- The Init definition specifies the initial condition where the counter is initialized to zero, just as in the PlusCal code.

TLC checks all reachable states of a TLA+ specification to verify its properties.

ALLOY Model Checker

ALLOY is a formal modeling language, analysis and verification of complex systems. Developed by Daniel Jackson (Jackson, 2019).

An executable system in ALLOY is built by defining the systems structure, specifying properties and using ALLOY analyzer SAT-based constraint solver to determine its satisfiability.

Supposing executable model for social network is built in ALLOY as shown in Algorithm 3.

Algorithm 3: Alloy Pcode for Social Network

```
sig User {
  name: String,
  age: Int,
  friends: set User
}

pred FriendsAreMutual {
  all u: User | u in u.friends
}
fact AgeRestriction {
  all u: User | u.age >= 18
}
```

The above ALLOY model describes a simple social network with users, their friendships, and an age restriction. It captures the structural properties of the social network and enforces certain constraints on the relationships and attributes of the entities as follows:

- Entity Declaration: An entity is declared as 'User' using the 'sig' keyword. The 'User' entity has three attributes: 'name' of type' String', 'age' of type 'Int', and 'friends' of type 'set User' representing a set of other users.
- Predicate: The 'FriendsAreMutual' predicate states that for every user 'u', 'u' should be in the set of friends of 'u'. This predicate ensures that friendship relationships are mutual.
    - Fact: The 'AgeRestriction' fact specifies that for all users 'u', their age should be greater than or equal to 18. This fact imposes an age restriction on users, assuming that only users above 18 years of age can join the social network.

Exploring the behaviour of the social network using first-order logic, we can translate the ALLOY model into logical formulas. Here are the properties expressed in first-order logic:

Property: Friends are mutual.

First-order Logic: $\forall u, v: User (u \in v.friends \rightarrow v \in u.friends)$

It implies that, for any two users u and v, if u is in the set of friends of v, then v must also be in the set of friends of u. This ensures that friendship relationships are mutual.

Property: Age restriction.

First-order Logic: $\forall u: User (u.age \geq 18)$

Meaning all users u, the age of u must be greater than or equal to 18. This enforces the age restriction that only users above 18 years of age can join the social network.

Using the formulas for formal reasoning and verification of the social network system, we can specify constraints that define certain conditions or restrictions that must hold within the system. Here are some possible constraints for the social network model:

1. Constraint: Unique User Names
   Formula:

$\forall u, v: User | u \neq v \rightarrow u.name \neq v.name$

This constraint ensures that every user in the social network has a unique name. It states that for any two distinct users u and v, their names should not be the same.

2. Constraint: Friendship Symmetry
   Formula:

$\forall u, v: User | u \in v.friends \rightarrow v \in u.friends$

This constraint ensures that the friendship relationship is symmetric. It states that if user u is a friend of user v, then v must also be a friend of u. This constraint enforces the mutual nature of friendships.

3. Constraint: Age Limit
   Formula: $\forall u: User | u.age \geq 18$

This constraint imposes an age limit on users. It states that the age of every user in the social network must be

greater than or equal to 18. This ensures that only users above 18 years of age can join the network.

4. Constraint: Non-empty Friends Set
Formula: $\forall u: User | u.friends \neq \emptyset$

This constraint ensures that every user in the social network has at least one friend. It states that the set of friends of every user should not be empty.

The ALLOY analyzer converts the constraint and logical formulas specified in the ALLOY model into a Boolean formula and employs a SAT solver to determine its satisfiability using various algorithms and optimization such as conflict-driven clause learning, backtracking, symmetry breaking, constraint propagation and random sampling to enhance the efficiency of the analysis.

It is worth noting that ALLOY's analysis capabilities are not limited to satisfiability checking. It also supports various forms of model checking, including temporal logic model checking, and can perform reachability analysis and simulation-based exploration of the system model.

Failure Divergence Refinement

According to Gibson-Robinson *et al.,* (2014), Failure divergence refinement (FDR) is a variation of popular refinement checker in process algebra, which propagates communicating sequential processes (CSP). Deviations from precision errors require a number of sequential communication processes, which were described in the machine communication sequential processes (CSP_M). It is a lazy language function that allows inhibited processes to clear each other according to the learning model of sequential communication processes. FDR is able to check for properties, including deadlock-freedom, livelock-freedom and determinism, by constructing equivalent refinement checks.

Apart from CSP_M specification language, Z formal specification language based on Zermelo Fränkel axiomatic set theory and first order predicate logic, can be used to model checking CSP with the support of the FDR tool (Ruhela, 2012). Z allows specification to be decomposed into small pieces called schemas which distinguishes Z from other formal notations.

As a demonstration, supposing failure-divergence semantics of Z-schemas for “Friendship Request Failure” within the social network is defined as follows in Algorithm 4.

1.To model the system using Z schemas:

Let's define two Z schemas: User and Friendship.

Algorithm 4: Z schemas for User and Friendship

```
schema User
  name: string
  friends: set User

schema Friendship
  sender: User
  receiver: User
  status: {"Pending", "Accepted", "Rejected"}
```

2. To specify failure and divergence scenarios:
In this example, focus is on a failure scenario where a user's friend request fails due to a network error.
3. To refine the Z schemas:
Refining the Z schemas to capture the failure scenario, let's modify the Friendship schema to include a failure behaviour as shown in Algorithm 5.

Algorithm 5: Failure Bahaviours

```
schema Friendship
sender: User
receiver: User
status: {"Pending", "Accepted", "Rejected", "Failure"}
```

4. To verify properties and analyze behaviour:
Now, we can use the FDR tool to analyze the behaviour of the system. For example, verification of properties such as "If a friendship request fails, the status should be set to 'Failure'.

5. To identify potential problems and refine the model:
Upon analysis, if the property is violated or if it is discovering issues with the system behavior, we can refine the Z schemas and repeat the analysis until we achieve the desired behaviour.

The failure-divergence semantics and refinement process using the FDR tool would require a more comprehensive consideration of failure and divergence in a complex real-world scenario, as well as refining the Z schemas accordingly to capture those behaviours accurately.

The author also considers Vienna Development Method Specification Language Analyser (VDM-SL) and Construction Analysis Distributed Processes (CADP). The work from (Sengunta and Dasgupta, 2015; Gravel et al., 2011; Mateescu, 2008) give further investigations.

COMPARATIVE STUDY OF SYMBOLIC ALGORITHMS

Figure 2 presents the context diagram showing all the symbolic algorithms examine such as Symbolic Model Verifier (SMV), Cadence SMV, A New (NuSMV), Construction Analysis Distributed Processes (CADP), ALLOY, Vienna Development Method Specification Language Analyzer (VDM-SL), TLA+ Model Checker (TLC) and Failure Divergence Refinement (FDR).

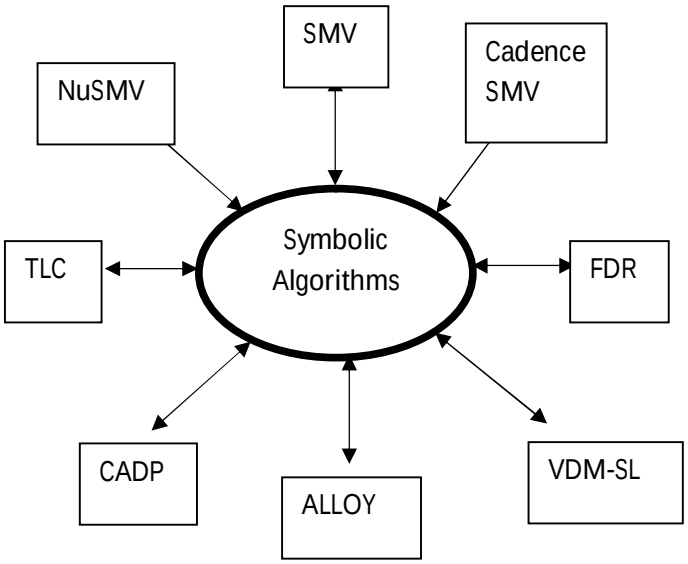

Figure 2: Context Diagram for Symbolic Algorithms

The study considered features such as state space, data intensive operation, model checking process, type of system models, time complexity, state explosion problem, error identification (Counterexample), Sufficient State Condition to Enable an Event (SCE), Necessary State Condition to Enable an Event (NCE), Sufficient State Condition to Enable an Event on some Execution Path (SCEF), class of properties and language specification as comparison measure for the symbolic algorithms using a table-driven approach as shown in Table 1.

Table 1

Table-driven Approach of Symbolic Algorithms

Features	SMV	Cadence SMV	NuSMV	CADP	ALLOY	VDM	TLC	FDR
State Space	ROBDD, BuDDy	BDD, CUDD	BDD, CUDD, OBDD	BCGs	BDDs	ROBDD	BDDs	BDDs, MDD
Data Operations	Conjunction & Disjunction	Conjunction &Disjunction	Conjunction & Disjunction	Conjunction & Disjunction	Relations, Conjunction, Disjunction, Negation, Implication & Bi-implication	Arithmetic operations &Sets	Sets, Boolean functions	Relations, Conjunction, Disjunction, Negation, Implication & Bi-implication
Model Checking Process	Modular	Modular	Modular	Modular	Modular	Monolithic	Modular	Monolithic
System Models	Synchronous & Asynchronous	Real-time, Distributed & Register transfer level (RTL) designs	Synchronous finite state and infinite state system.	Asynchronous concurrent systems	Real-time	Critical &Concurrent	Concurrent & Distributed	Real-time
Time Complexity	High	Low	Medium	Medium	Low	Medium	Low	High
State Explosion	High	High	High	Low	Relatively minimal	High	Low	High
Counterexample	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SSE	No	No	No	No	Yes	No	Yes	No
NCE	Yes	Yes	Yes	No	Yes	No	Yes	No
SSCE	Yes	Yes	Yes	No	Yes	No	Yes	No
Properties	Safety, Liveness and Deadlock Freedom	Liveness properties	Invariant, Fairness & reachability properties	Safety properties	Invariant (signature)	Invariants	Invariance (Safety & Liveness)	deadlock-freedom, livelock-freedom and determinism
Language Spec.	Computational Tree Logic (CTL)	First-order Linear Time Logic (LTL)	CTL & LTL	LOTOS & Executable Temporal Language (XTL)	First-order logic	VDM-SL	TLA+	CSP_M & Z notations

Table-driven Descriptions

This section describes the theme pattern of the table-driven approach as follows:

a) State space: This feature shows the set of all possible states that a system can be in during its execution. The Construction and Analysis of Distributed Processes (CADP), ALLOY, Temporal Logic of Action (TLA+) model checker (TLC) can explore all the varieties of BDD. The Symbolic Model Verifier (SMV) can explore ROBDD, BuDDY, the Cadence SMV can explore BDD and CUDD, the NuSMV can explore BDD, CUDD, OBDD, Vienna Development Method (VDM) can explore ROBDD, TLC can explore BDD while FDR can explore BDD and MDD. The algorithms systematically explore the state space by performing a breadth-first or depth-first search to verify the desired properties.

b) Data intensive operation: The set of operation that can be performed typically expressed as logical formulas. The SMV, Cadence SMV, NuSMV, and CADP performs some other operations but preferably represented in a conjunctive and disjunctive form. ALLOY performs all operations which ranges from relational functions (Union, Intersection) and Logic functions (Conjunction, disjunction among many others.). VDM performs arithmetic and set operations; TLC performs Set, Binary, Unary and Boolean functions and FDR performs Relations, Conjunction, disjunction, negation, implication, bi-implication. To handle data-intensive operations, various techniques and optimization using data structures and indexing methods to reduce memory usage.

c) Model checking process: The process of model checking could be monolithic or modular. SMV, Cadence SMV, NuSMV, CADP, ALLOY and TLC use the modular/library approach (Modern method of model checking), separate verification engine from the input formalism while VDM and FDR use the monolithic approach (Traditional method of model checking), having single entry point formalism for input and verification processes.

d) System models: SMV model synchronous and asynchronous system including NuSMV and CADP. Cadence SMV works on a real-time, distributed system and particularly for register transfer level (RTL) designs. ALLOY and FDR model a real-time system. NuSMV also model embedded system while VDM and TLC model critical and concurrent system. In essence, building executable system models is one of the requirements for model checking process.

e) Time Complexity: This feature shows the average time for execution of number of instances. SMV and FDR has high time complexity; Candence SMV, ALLOY and TLC has low time complexity while NuSMV, CADP and VDM a medium time complexity.

f) State explosion problem: SMV, NuSMV, Cadence SMV, FDR and VDM experience high exponential growth in the size of the state space. CADP and TLC combinatorial nature of systems is quite low while ALLOY has relatively minimal state explosion problem due to several techniques used such as scope-based exploration, relational logic, constraint solving Barret and Tinelli, (2018) and counterexample guided abstraction refinement (Clarke *et al.*, 2003).

g) Error identification (Counter-examples): This is a trace of states and transitions leading to the violation. All symbolic model checkers produces this feature. However, the method of analyzing and diagnosing the counterexample differ and faster than each other.

h) Sufficient State Condition to Enable an Event (SSE): These are relatively sufficient in state-based languages like NuSMV, SMV, and Cadence SMV. This property must be approximated in ALLOY, otherwise, too large number of atoms will result to computational load. The validity of the approximation relies on the hypothesis that the post condition of an action is satisfiable when the precondition holds. TLC can also handle these properties using stable failure refinement, but sometimes by approximation.

i) Necessary State Condition to Enable an Event (NCE): This is a minimum requirement that must be met in order for an event to be triggered. These are relatively satisfied in state-based languages like NuSMV, SMV, Cadence SMV, with some

approximations for ALLOY (similar to SSE). The author observe that extensible temporal logic (XTL) has not been specified for CADP, because the states were too difficult to characterize using events only. There were no problems to specify these conditions using trace refinement in TLC. However, the study suspects that there may be cases where characterizing states using only events may be difficult.

j) Sufficient State Condition to Enable an Event on some Execution Path (SSCE): This involve certain conditions within the system's state that need to be met for the event to be triggered. With the development of computational tree logic (CTL), SMV, Cadence SMV and NuSMV conditions were specified along a specific execution path. It is possible for TLC when the state condition is easy to characterize using events. It can be specified in ALLOY by providing the path of events leading to the desired event, when such a path does not exceed the number of atoms available through scope-based exploration.

k) Class of Properties: All the symbolic algorithms exhibit one or more properties to characterize the expected behaviour of a system.

l) Language Specification: There are different specification language identified in this study for symbolic model checking. The authors found out that executable temporal logic (XTL) for CADP still need more exploration.

Based on the above findings, the following research topics were generated for future research purposes:

a) Quantitative Analysis on the symbolic algorithms.

b) A comparative study of symbolic algorithms using Configurable Program Analysis (CPA) and Great Stochastic Petri Net (GreatSPN) as tool for implementation.

c) Improving the performance and scalability of symbolic analysis on ALLOY models, allowing the analysis of larger and more complex systems.

CONCLUSION

Symbolic model validation provides a systematic and automated approach to creating models of a complex system based on desired properties, allowing for deeper analysis and detection of potential errors or violations. This article recognizes that many problems in various fields can be formulated in terms of Boolean functions, which led to the emergence of various variants of BDD. They are useful because they represent a state's space and transitions to analyze a typical model.

Eight (8) symbolic algorithms were considered on the basis to know which is the best to alleviate state explosion problem using table-driven analysis. Twelve (12) features were selected for the comparative analysis for the symbolic algorithms considered. The study presents an exemplary model of the system and the specification of the language used in the selected algorithms. The results shown that the ALLOY is the best in terms of relatively minimal rate of state explosion which is as a result of sufficient state condition on some execution path provided in ALLOY through scope-based and constraint solving explorations.

REFERENCES

Abraham, E., & Kremer, G. (2017). SMT Solving for Arithmetic Theories: Theory and Tool Support. 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania.

https://ieeexplore.iee.org/document/8531255/authors

Barrett, C., & Tinelli, C. (2018). Satisfiability Modulo Theories. Handbook of Model Checking, pp. 305-343. Springer. https://link.springer.com/chapter/10.1007/978-3-319-10575-8_11

Biere, A., & Kroning, D. (2018). SAT-Based Model Checking. In: Clarke E., Henzinger T., Veith H., Bloem R. (eds), Handbook of Model Checking, pp. 277-303, Springer, Cham.

Clarke, E. M., & Lerda, F. (2007). "Model Checking: Software and Beyond". Journal of Universal Computer Science, vol. 13, no. 5, 639-649.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y. & Veith, H. (2003). Counterexample-guided Abstraction Refinement for Symbolic Model Checking. Journal of the ACM, Volume 50, Issue 5, pp. 752-794.

Cimatti, A., Clarke, E. M., Giunchigha, F., & Roveri, M. (2000). NuSMV: A New Symbolic Model Checker. International Journal on Software Tools for Technology Transfer (TACAS), Volume 2, pp. 410-425.

Cordeiro, L., Fischer, B., & Marques-Silva, J. (2009). SMT-based Bounded Model Checking for Embedded ANSI-C Software. https://arxiv.org/abs/0907.2072v2

Deharbe, D., (2003) A Tutorial Introduction to Symbolic Model Checking. In: Ruy J. B. B., de Queiroz. Logic for Concurrency and Synchronization, pp. 215-237, Kluwer Academic Publisher, Springer.

De Moura, L., & Bjorner, N. (2011). Satisfiability Modulo Theories: Introduction and Applications, Communications of the ACM, Volume 54(9), pp. 69-90.

Dijk, T. V., & Pol, J. V-D. (2016). "Sylvan: Multi-core Framework for Decision Diagrams". International Journal on Software Tools for Technology Transfer, 19:675-696. https://links.springer.com/content/pdf/10.1007/s10009-016-0433-2.pdf

Gibson-Robinson, T., Armstrong, P., Boulgakov, A., & Roscoe, A. W. (2014). FDR3- A Modern Refinement Checker for CSP. International Conference on Tools an Algorithms for the Construction and Analysis of Systems (TACAS), pp. 187-201.

Gravel, H., Lang, F., Mateescu. R. & Serwe, W. (2011). CADP 2010: A Toolbos for the Construction and Analysis of Distributed Processes. International Conference on Tools an Algorithms for the Construction and Analysis of Systems (TACAS), pp. 372-387

Jackson, D. (2019). ALLOY. Communications of the ACM, Volume 62(9), pp. 66-76.

Leslie, L. (2002). Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison Wesley. ISBN 978-0-321-14306-8.

Levin, I. & Keren, O. (2018). "Split Multi-terminal Binary Decision Diagrams". 8th International Workshop on Boolean Problems, Freiberg.

Li, G., Gopalakrishnan G., Kirby R. M. & Quinlan D. (2010). A Symbolic Verifier for CUDA Programs. Conference Paper in ACM SIGPLAN

Li, Y. (2013). Symbolic Abstraction with SMT Solvers. PhD Thesis, University of Toronto, Canada.

Mateescu, R. (2008). Specification and Analysis of Asynchronous Systems Using CADP https://inria.hal.science/inria-00264235

McMillan, K. L. (1992). "Symbolic Model Checking: An approach to the state explosion problem" PhD Thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Merz, S. (2000). Model Checking: A Tutorial Overview. Modeling and Verification of Parallel Processes, pp 3-38. https://members.loria.fr/SMerz/papers/mc-tutorial.pdf

Minel, C., & Theobald T. (1998). "Algorithms and Data Structures in VLSI Design". Springer Verlag Berlin Heidelberg.

Noureddine, M., & Zaraket, F. A. (2016). Model Checking Software with First Order Logic Specifications Using AIG Solvers. IEEE Transaction on Software Engineering, 42, 741-763.

Padon O., Hoenicke J., McMillan K. L., Podelski A., Sagiv M., & Shoham S. (2021). Temporal Prophecy for Proving Temporal Properties of Infinite-state Systems. https://arxiv.org/pdf/2106.00966.pdf

Ruhela, V. (2012). Z formal Specification Language-An Overview. International Journal of Engineering Research and Technology (IJERT), Vol 1, Issue 6, Page 1-7.

Sasao, T. & Nagayama, S. (2006). "Representations of Elementary Functions Using Binary Moment Diagrams". IEEE 36th International Symposium on Multiple-valued Logic, Singapore.

Sasao, T. & Butler, J. T. (2014). Applications of Zero-suppressed Decisions Diagrams. Synthesis Lectures on Digital Circuits and Systems, Vol. 9(2), pp 1-23

Sengupta, S., & Dasgupta, R. (2015). A VDM-based Approach for Specifying and Testing Requirements of Web-Applications. Elsevier proceeding of the International Conference on Information and Communication Technologies, Volume 46, pg. 774-783.

Somenzi, F. (2018). "CUDD:CU Decision Diagram Package Release 2.7.0". University of Colorado at Boulder.

Souri, A., Rahmani, A. M., Navimipour, N. J., & Rezaei, R. (2019). A Symbolic Model Checking Approach in Formal Verification of Distributed Systems. Human-centric Computing and Information Sciences 9, Article number: 4(2019).

Toussi, H. A., & Bigham, B. S. (2014). Design, Implementation and Evaluation of MTBDD-based Fuzzy Sets and Binary Fuzzy Relations. https://arxiv.org/abs/1403.1279v1

Wegener, I. (2004). BDDs-design, Analysis, Complexity and Applications. Discrete Applied Mathematics, Volume 138, Issues 1-2, Pages 229-251.

Yang Z., Wang C., Gupta A., & Ivancic F. (2006). "Mixed Symbolic Representations for Model Checking Software Programs". 4th IEEE/ACM International Conference on Formal Methods and Models for Co-Design, Napa, Califonia.

Yu, H., Chen, Z., Fu, X., Wang, J., Su, Z., Sun, J., Huang, C., & Dong W., (2020). Combining Symbolic Execution and Model Checking to Verify MPI Programs. https://arxiv.org/pdf/1803.06300.pdf