Compositional Verification of Data Invariants in Promela using Slicing Technique

Bass Srongsil, Wiwat Vatanawood

Abstract—**In this paper, we propose an alternative of compositional verification of data invariants in Promela code. The Promela source code, c-like source code, is analyzed and decomposed into a set of code chunks using the program slicing technique. Each code chunk of Promela will be verified to satisfy each simple logical condition term separately found in the linear temporal logic properties of data invariants. The separate checks are performed and eventually consolidated into the final result. It obviously shows that the state space of each separated model checking task has been reduced and more tractable. The SPIN model checker tool is used to evaluate our final results.**

Index Terms—**Compositional verification, Data invariant, Program slicing, Promela, SPIN**

I. Introduction

SPIN is an automated verification tool developer by [1], which needs Promela as an input language for processing verification in model checking technique. It is one of the most well-known linear temporal logic (LTL) model checkers which can simulate various properties: safety properties that are often characterized as "something bad never happens", liveness properties that are characterized as "something good will happen in the future" [2].

For example, safety property can be described as the mutual exclusion property—at most one process is always in its critical section. Thus, the bad thing which is having two or more processes in their critical section concurrently should never occurs. Deadlock freedom is also a typical safety property in dining philosophers problem for instance. The occurrence of each of philosophers holding left chopstick and waiting for his right chopstick to be available or vice versa must never occur. Checking safety property or data invariant checking can be performed by traversing the state space and checking whether all of them that are reachable hold the data invariant. By holding the data invariant, state space must satisfy safety property which can be specified by propositional temporal logic. This approach can grow the system state space exponentially which leads to the state space explosion problem. For example, a system composed by n processes and each of them have m states. The asynchronous composition from processing the system may be m^n states which quickly leads to the maximum capabilities of verification tool.

There are several proposed solutions to the state explosion problem in model checking verification. For example, using reduction techniques to do on-the-fly reduction of the structure of the original state space [3], using abstraction techniques to get the simpler formal model from the original one [4], etc. In this paper we consider the divide-and-conquer techniques by decomposing the formal model written in Promela code into a set of code chunks. The slicing technique of source code is considered since the Promela code is a c-like source code. The data invariants are expressed in terms of temporal formula consisting of atomic propositions and temporal operators. Our proposed slicing technique will consider only relevant variables found in atomic proposition as the slicing criteria for each program slice. Obviously, we intend to reduce the size of the model beforehand into a set of smaller code chunks and separately perform the smaller model checking tasks.

The rest of paper is organized as follows. Section II describes backgrounds. Section III describes related works. Section IV describes our slicing methodology. Section V discusses the results of our compositional verification. Section VI is our conclusion.

II. Background

A. Propositional Logic

Proposition logic is a simple logical system that allows to reason logical expression whether it is true or false [5]. Logical expression can contain logical operators such as AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). Atomic propositions express simple known facts about the states of the system under consideration for example "x equals 0", or "x is smaller than 100" for some given integer variable x. Given AP is a set of atomic propositions. Latin letters such as a, b and c (with or without subscripts) are used to denote elements of AP. Four rules are defined for the set of propositional logic formula: (1) true is formula, (2) any atomic proposition a $\in AP$ is a formula, (3) if Φ_1, Φ_2 and Φ are formula, then so are ($\neg\Phi$) and ($\Phi_1 \wedge \Phi_2$) and (4) nothing else is formula [6]. Proposition might hold or not depends on which of the atomic propositions are assumed to hold. For example, $\Phi_1 \wedge \Phi_2$ holds if and only if Φ_1 and Φ_2 hold, $\Phi_1 \wedge \neg\Phi_2$ holds if and only if Φ_1 holds and Φ_2 does not hold and true holds in any context.

B. Temporal Logic

Temporal logic extends propositional logic by modalities that allows for the specification of the relative order of events [2]. These modalities allow specifying the order in

Bass Srongsil is with the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, Thailand (e-mail: bass.s@sudent.chula.ac.th).
Wiwat Vatanawood is with the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, Thailand (e-mail: wiwat@chula.ac.th).

which state labels occur during system execution, or to assess that certain state labels occur infinitely often in system execution. Propositional logic which is extended by temporal logic is *propositional temporal logics*. The elementary temporal modalities that are written in most temporal logics include operators as X (next), G (Globally), F (Finally) [7] presented in Fig. 1, which is based on [2].

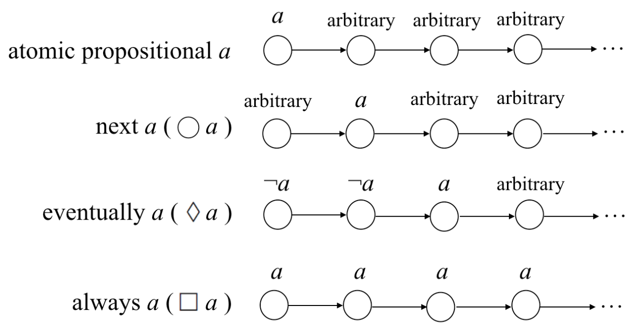


Fig. 1. An example of semantics of temporal modalities [2]

Two types of temporal logics are most commonly used for model checking: Computational Tree Logic (CTL) that is a branching temporal logic for specifying system properties, and Linear Temporal Logic (LTL) that is interpreted over the set of CTL paths [8].

C. *Invariant*

Invariant, such as safety properties, are linear time properties that require condition to hold for all reachable states [2]. An example for mutual exclusion, at most one process is always in its critical section, can be described by an invariant using the propositional logic formula as (1). Given Φ is invariant condition and $inCritical_i$ is the proposition that characterizes the state(s) of having process in its critical section.

$$\phi = \neg inCritical_1 \wedge \neg inCritical_2 \qquad (1)$$

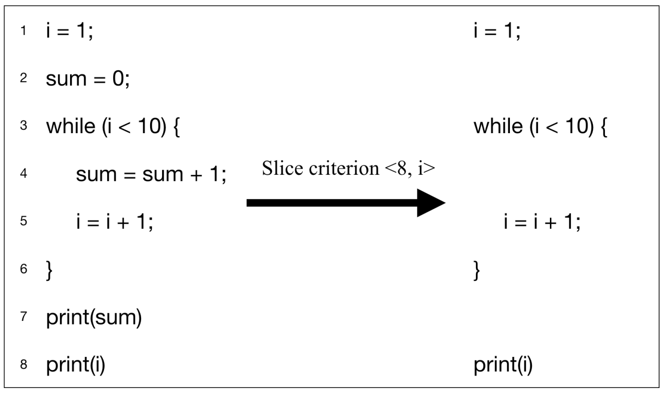
For deadlock freedom of dining philosophers problem, the invariant ensures that at least one of the dining philosophers is not waiting to pick up the chopsticks [2]. It can be described by using the propositional logic formula as (2). Given $waitForChopstick_i$ is the proposition that characterizes the state(s) of philosopher I in which he is waiting for a chopstick.

$$\begin{aligned}\phi = \neg waitForChopstick_1 \wedge \neg waitForChopstick_2 \\ \wedge \neg waitForChopstick_3 \wedge \neg waitForChopstick_4\end{aligned} \qquad (2)$$

D. *Program Slicing*

Program slicing is basically a decomposition technique. It elides program components that are not related to a preserved subset of the original program's behavior [9]. A specification of the subset is known as a slicing criterion, and the resulting subprogram is a slice. Slicing criterion is defined by [10] as a pair ‹*i*, *V*›, combining a program point, *i*, and a set of variables, *V*. A slice is computed with respect to a slicing criterion that consists of a selected variable and program location. There are several kinds of slicing such as backward slices and forward slices. Both of them include variable assignment in the slice in order to preserve semantics of the chosen variable at the location in the slicing criterion.

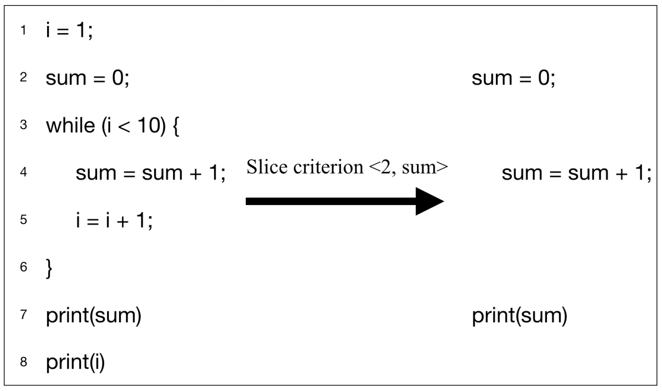
Backward slicing is one kind of program slicing which computes to answer the question "what program components might effect a selected computation" [17]. The example of backward slicing is shown in Fig. 2, which is based on [9]. Slicing criterion is specified as ‹*8*, *i*›, which is related to variables *i* in statement *print(i).* Then each statement before execution of statement *print(i)* is checked if it contains variable *i.* So statement that is related to variable *sum* such as *sum = 0* and *sum = sum + 1* will not be included in a slice.



```
1  i = 1;                                          i = 1;
2  sum = 0;
3  while (i < 10) {                                while (i < 10) {
4      sum = sum + 1;   Slice criterion <8, i>
5      i = i + 1;       ────────────────────▶          i = i + 1;
6  }                                               }
7  print(sum)
8  print(i)                                        print(i)
```

Fig. 2. An example of backward slicing [9]

Whilst, forward slicing is another kind of program slicing which computes to answer question "what program components might be effected by a selected computation" [17]. An example of forward slicing is shown in Fig. 3, which is based on [9].



```
1  i = 1;
2  sum = 0;                                        sum = 0;
3  while (i < 10) {
4      sum = sum + 1;   Slice criterion <2, sum>       sum = sum + 1;
5      i = i + 1;       ────────────────────▶
6  }
7  print(sum)                                      print(sum)
8  print(i)
```

Fig. 3. An example of forward slicing [9]

Slicing criterion is specified as ‹*2*, *sum*›, then each statement after statement *sum = 0* is checked if it is affected by variable *sum*. So statement that is related to variable *i* such as *i = i + 1* will not be included in a slice since it is not affected by the variable *sum*.

E. *Program Dependency Graph*

Program dependency graph (PDG) is a graph representing the consecutive statements of the cascading dependency of the data values assignments in the program. It simplifies the static analysis of the program and typically used to perform the program slicing. It has been a method to identify the relevant entities of the program according to data and control dependencies [11].

Formally, a PDG is a tuple (*V*, *E*) where *V* is a set of vertices and *E* is a set of edges. The vertices of a PDG represent program statements, control predicates, and regions of the program. While, the edges of a PDG represent data or control dependencies between the vertices.

The PDG is related to program slicing since program slicing can be reduced to the graph reachability problem in which a program slice is a set of vertices that can be reached from some indicated vertex. Based on Fig. 2., PDG of the original program can be written in Fig. 4, which is based on [11].

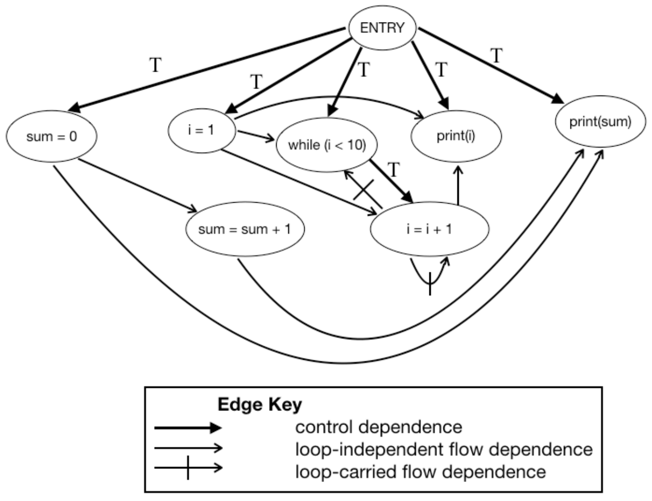


Fig. 4. An example of PDG of the original program based on Fig. 2. [11]

After the backward slicing with the slicing criterion ‹*8*, *i*›, a resulting smaller slice is illustrated by the PDG shown in Fig. 5, which is based on [11]. The unrelated vertices of the original PDG are elided according to the slicing criterion ‹*8*, *i*›. As you can see, the vertex of *sum = 0* and its cascading dependency vertices, carrying the variable sum, are left out.

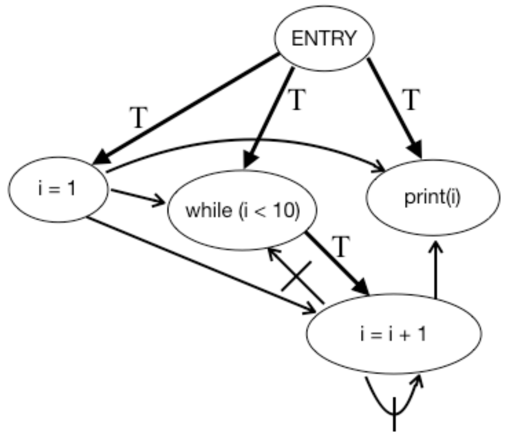


Fig. 5. An example of PDG of the sliced program based on Fig. 2. [11]

III. Related work

Many approaches for reducing state space has been investigated. Partial order reduction, based on research of [12], is a technique for reducing size of the state space by constructing a smaller state space that is searched by the verification algorithms. Their experiment was based on safety and liveness properties of parameterized systems, which produced excellent results as huge reduction of states, transitions, memory and computing time because the local semantic accelerations are strong enough to get new results that match or beat old results in the context of regular model checking. Divide-and-Conquer is another state space reduction technique. It decomposes properties of the system into properties of its components, then checks each component separately. It is necessary to incorporate knowledge of the context for expectation of each component to operate correctly. An example for state space reduction by divide-and-conquer technique is [13], which proposed an automatic method for systematically extracting subparts, then applying divide-and-conquer approach for computation subparts efficiently. Another work by [14] proposed agglomeration for state space reduction for model checking concurrent C programs. The agglomeration predicate needed to be defined which took as argument a C statement and returned true or false depending on whether the statement could be agglomerated or not. Also, the predicate was checked if it was safe or unsafe. It was called unsafe as a statement which might contain the global memory agglomerated. Agglomerating actions could reduce the number of states in control flow graph, for example two consecutive statements are $x = x +1$ and $x = x +2$, the agglomeration technique could combine two statements by one action which is the result as well as of executing the statement $x = (x+1) + 2$. The statement with a *return* statement followed could be agglomerated into one action as well, for example two consecutive statements such as $x = x + 1$ and *return*. Once the states in control flow graph was reduced, the resulting state space from model checking with this reduction techniques, is reduced accordingly. The results from experimental of agglomeration include 5 programs: Zunebug, Lamport's Bakery and Peterson algorithms, Dekker mutual exclusion algorithm and Dining philosopher problem, which could reduce huge number of states and also time for model checking as the ratio between the state space generated without reduction and the one after applying the reduction technique was approximately 90 to 99. The research of [15] which was about state space reduction in Agent Verification, applied program slicing to eliminate details of the program that were not related to the analysis in hand. The approach from [16] was chosen due to similarity of the programs. Furthermore, the executable slices did not need to be generated. Experimental results were shown as the comparison of the memory usage between traditional SPIN and the slicing approach—traditional SPIN used 606 MB while the slicing approach used only 407 MB. Moreover, the comparison of computing time between them showed that traditional SPIN used 86 seconds while slicing approach used 64 seconds. On the other hand, the study about program slicing by [17] applied static backward slicing to reduce the cost of property checking. The experimental results compared running time for checking the two properties using the first models from original program, from sliced program, and finally from sliced program with abstraction. With slicing approach, it took the time less than one second.

IV. Our slicing methodology

In this section, we describe our slicing methodology for Promela code. An example of Promela code of automated teller machine (ATM) is given and shown in Fig. 6. The Promela code briefly specifies the states of a ATM system using label names, including welcome idle state (*s_welcome*), enter pin state (*s_enter_pin*), main menu state (*s_main_menu*), deposit state (*s_deposit*), withdraw state (*s_withdraw*), and try again state (*s_try_again*).

The ATM system is normally in the welcome idle state and will respond to the events of depositing and withdrawing cash. The customer needs to enter the valid pin number and the amount of money to be checked with his/her current account balance. A sample of the data invariants of this ATM system could be specified in LTL property such as $\Box(\neg(account\_balance < 0))$, meaning that the value of *account_balance* should not less than zero at all times. According to the mentioned LTL property, we select the slicing criteria from the variables found, which is a pair ‹*location*, *account_balance*› where location is the last line number of the Promela code.

Our proposed slicing method is based on [14]. A slicing criterion is produced from extraction of variables in propositional temporal formula for data invariant checking, which can create a slice that contains relevant statements. After that a data invariant checking is processed based on each slice. The details of the activities are explained below.

```
active proctype exampleBankMachine()
{
…
…

s_welcome :
        state = S_Welcome;
        pin = 0;
        printf("state welcome");
        …
        goto s_enter_pin;

s_enter_pin :
        state = S_Enter_Pin;
        printf("state enter pin");
        if
                :: pin correct …
                :: pin incorrect …
        fi;

s_try_agagin :
        state = S_Try_Again;
        printf("state try again");
        …

s_main_menu :
        state = S_Main_Menu;
        printf("state main menu");
        …

s_deposit :
        state = S_Deposit;
        printf("state deposit");
        account_balance += deposit_amount
        …

s_withdraw :
        state = S_Withdraw;
        printf("state main menu");
        account_balance
        if
                :: account_balance < withdraw_amount
                …
        fi;
…
}
```

Fig. 6. An example of automated teller machine in Promela source code

A. Identify slicing criterion

A slicing criterion is produced from the propositional temporal formula for data invariant checking, which might contain variables, constant values and operators. The variables are extracted to provide a slicing criterion for processing program slicing in the next step while the constant values and the operators are extracted to interpret the logical expression based on the value of variable for each state space whether it is true or false. A slicing criterion is a pair ‹*i*, *V*›, which *i* represents line number and *V* represents variable. This step produces slicing criterion as ‹*end*, *INV*›, which *end* is the last line number of proctype in Promela and *INV* is a variable from propositional temporal formula for data invariant checking. For example, we introduce propositional temporal formula for data invariant checking as $\Box(\neg(account\_balance < 0))$, an invariant formula that describes safety property as balance in bank account must never be less than zero in any situation. The variable *account_balance* and constant value 0 are accordingly extracted from the formula. Moreover, the logical operator NOT must be extracted in order to apply to the logical *account_balance* < *0* to be *account_balance* >= *0*. Given 100 is the last line number of proctype of this example. The slicing criterion for propositional temporal formula for data invariant checking $\Box(\neg(account\_balance < 0))$ is a pair ‹*100*, *account_balance*›.

B. Examine variable dependencies from PDG

PDG is created based on the original program, the graph shows dependencies between variables, statements and control flow. We have specified variable in slicing criterion from previous step, which can perform traversing vertices in PDG. The traversed vertices are marked and they will be included in a slice. Once the traversing process is complete, we will have the information which vertices are whether marked or not. The example slicing criterion from previous step is ‹*100*, *account_balance*›, which can perform traversing in PDG for checking if any vertices have dependencies to the variable *account_balance*. We found that the group of statements in the label *s_enter_pin* and *s_try_again* are not dependent to *account_balance*. On the other hand, the vertices that have dependency to *account_balance* are group of statements in the label *s_withdraw* and *s_deposit*.

C. Create a slice

This step produces a slice from PDG in previous step. The vertices that are not marked meaning that they are not related to variable in propositional temporal formula, and they are elided. A slice is created from the original program and it will contain only statements that are related to slicing criterion and also their dependencies statements. A sliced PDG according to the slicing criterion $\Box(\neg(account\_balance < 0))$ is shown in Fig. 7.

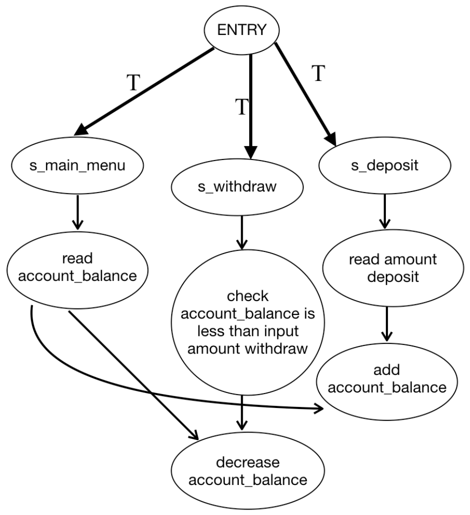


Fig. 7. A sliced PDG according to the slicing criterion $\Box(\neg(account\_balance < 0))$

D. Verify data invariant checking of a program slice

This study compares state space of data invariant checking between a slice and the original program using SPIN. So after a slice is created, it will be executed to verify data invariant in order to ensure the data invariant has been preserved. Data invariant must be exactly the same as the result from the original program's data invariant. In this paper, we focus on the Promela code with any size of one proctype and the propositional temporal formula, denoted as *A operator B*, where *A* and *B* are observable variables in Promela code, and *operator* ∈ {==, <, <=, >, >=, !=}. Propositional temporal formula can be single or multiple atomic proposition such as $\Box(\neg(account\_balance < 0)$

$\wedge \neg$*(account_name == null))*, given *account_name, account_balance* are the name of account owner and total account balance respectively.

V. Evaluation

The example Promela source code for automated teller chine is developed for the evaluation. The flow of the process is similar to simple automated teller machine. Start from putting the correct *PIN*, in case the *PIN* is incorrect then the state traverses to try again and getting *PIN* until it is correct. After that the process is about account managing such as deposition, withdrawal or transferring. This example, the data invariant checking on *account_balance* can produce a slice that has smaller size than the original program because program slicing approach can remove some statements such as statements about variable *pin* in state for entering pin, which can produce smaller state space than the original program as well. The total number of state space from data invariant checking of the original automated teller machine is 95, but after applying our slicing methodology, the total number of state space is reduced to 28. However, the different of state space on data invariant checking from our approach and the original program depends on how many irrelevant statements that the original program contains. The more irrelevant statements in the original program, the more state space can be reduced from our approach. The experiment results are based on source code from [18]. The results are shown in Table I, where #Proc denoted the process number and #State space denoted the numbers of state space.

VI. Conclusion

Program	#Proc	With our slicing methodology	Without our slicing methodology
		#State space	#State space
ATM	-	28	95
bakery	2	321	797
dining philosophers	5	96	967
peterson	3	2511	3112

Table I: Comparing model checking results with and without our slicing methodology

In this paper we propose an alternative of compositional verification of data invariants in Promela source code using a slicing technique. The data invariants, as the LTL safety property, are written in terms of propositional temporal formula which are used to specifying the slicing criteria. The variables found in a propositional term of the propositional temporal formula would be considered as a slicing criterion, used to perform a program slicing of Promela code. Each small resulting slice of Promela code would be verified to satisfy each propositional term at last to conclude its validity of the data invariants. We propose a backward slicing from the last line number of the Promela code. We provide the obvious evidence of the compositional verification of the smaller set of state spaces from the Promela slices. However, we need to pursue more experiments on the huge Promela codes in our future work.

References

[1] G. J. Holzmann and D. Peled and M. Yannakakis. On nested depth-first search. In 2nd International SPIN workshop on Model Checking of Software, pages 23–32. AMS Press, 1996.

[2] C. Baier, J. Katoen, “Principles of Model Checking (Representation and Mind Series)”, pages 20–230, 2008.

[3] J. C. Fernandez, M. Bozga, and L. Ghirvu. State space reduction based on live variables analysis. Journal of Science of Computer Programming (SCP), 47(2-3):203–220, 2003

[4] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. ACM Transactions on Programming Languages and Systems, 16(5):1512–1542, 1994

[5] Ben-Ari M. (2012) Propositional Logic: Formulas, Models, Tableaux. In: Mathematical Logic for Computer Science, pages 7–16. Springer, London

[6] H. Pospesel, “Introduction to Logic: Propositional Logic.”, 1979.

[7] C. Artho, P. Ölveczky, “Formal techniques for Safety-Critical Systems,” 4th International Workshop, Paris, France, November 6-7, 2015.

[8] S. Edelkamp, A. Lomuscio, “Model Checking and Artificial Intelligence”, 4th Workshop, MoChArt IV, August 29, 2006.

[9] R.Ettinger, “Refactoring via Program Slicing and Sliding,” 2007 IEEE International Conference on Software Maintenance, Paris, 2007, pp 505-506.

[10] M. Weiser. Program Slicing. IEEE Transactions on Software Engineering, 10(4):352–357, 1984. 7, 17, 18, 126

[11] Jian-Liang Chen, Feng-Jian Wang and Yung-Lin Chen, "An object-oriented dependency graph for program slicing," Proceedings. Technology of Object-Oriented Languages. TOOLS 24 (Cat. No.97TB100240), Beijing, 1997, pp. 121-130.

[12] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled, “State Space Reduction using Partial Order Techniques,” International Journal on Software Tools for Technology Transfer, November 1999, Volume 2, Issue 3, pp 279-287

[13] Bengt Jonsson, Ahmed Rezine, and Mayank Saksena, “A Divide-and-Conquer Strategy for Regular Model Checking”, 2007

[14] A. Methni, B. Ben Hedia, M. Lemerre, S. Haddad and K. Barkaoui, “State Space Reduction Strategies for Model Checking Concurrent C Programs,” 9th Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'15), 2015, Vol. 1431, pp.65-76.

[15] R. H. Bordini, M. Fisher, W. Visser and M. Wooldridge, “State-Space Reduction Techniques in Agent Verification, Autonomous Agents and Multiagent Systems”, 2004.

[16] J. Zhao, J. Cheng, and K. Ushijima. Literal dependence net and its use in concurrent logic programming environment. In Proc. Workshop on Parallel Logic Programming, held with FGCS’94, ICOT, Tokyo, pages 127–141, 1994.

[17] K. Gallagher and D. Binkley, “Program Slicing”, Frontiers of Software Maintenance, 2008.

[18] Fumiyoshi Kobayashi. Bakery Algorithm in Promela, 2008. URL http://www.ueda.info.waseda.ac.jp/~kobayashi/Promela/benchmark/index.html (Accessed 2017-21-12).