A Multiple Refinement Approach in Abstraction Model Checking

Phan Nguyen, Thang Bui

A Multiple Refinement Approach in Abstraction Model Checking

Phan T.H. Nguyen and Thang H. Bui

Faculty of Computer Science & Engineering,
Ho Chi Minh City University of Technology, Vietnam
nguyenthihongphan@gmail.com, thang@cse.hcmut.edu.vn

Abstract. Abstraction in model checking is the most effective method to overcome the state explosion problem, the most serious problem in model checking when the size and the complexity of the system-under-check are increasing. Unfortunately, when the abstraction goes wrong, the answer must be validated with the concrete system, so it faces the state explosion problem again. Moreover, the techniques in checking the abstraction and in validating must not be obstructions in the checking process. Research recently has shown that, the way to abstract a model and the approach to use abstraction are the main concerns in abstraction model checking.
In this work, we report our study on both two questions: (1) a model analyzing method to find a way of abstraction effectively, and (2) an error refinement approach using multiple abstraction in symbolic model checking. The experimentation shows that the new approach has a great performance in checking both 'buggy' and 'correct' models.

1 Introduction

In model checking, a model is a specification of the under-the-check system for detecting violations of desired properties. When a violation is found, a counter-example is generated to help system engineers in allocating defects. Unfortunately, when the size and the complexity of system-under-check grows up, the state space will be exposed, the checking process is more intractable. To deal with the explosion, abstraction can be used to reduce the size of the model, and thus the effort. However, abstraction may introduce a 'fake' behavior that violates the specification, but that behavior is not present in the original system (false alarm).

When abstracting a system, domain abstraction [1], that reduces domains of the model, and predicate abstraction, that uses (pre-defined) predicates in reducing the model, are mostly employed. The research in [1] has discovered that in most system there are some *collar variables* that play an important role in representing the behavior of the system. Base on that idea, Qian et al. proposed that some 'weak' *system variables* that almost play no role in the system can be abstracted away [2, 3]. A simple variable dependency analysis [2] can be used to classified variables based on their impact to the system. Note

that, the research focused only on *directed model checking* [4]. The research in [5] proposed a propagation variable dependency analysis method to discover the impact of variables to each other. The experimentation has shown that, it outperforms the simple analysis method.

However, good abstract methods still introduce spurious counter-examples. Clarke et al. [6] states that refinement on a false alarm counter-example can eliminate it. Whenever a counter-example is found on the abstraction, it should be refined in the concrete system to validate the error. Therefore, abstraction refinement approach must deal with the state explosion on the concrete model when the abstraction is too bad to generate many 'fake' counter-examples.

Qian et al. [2] instead uses multiple abstraction and indicates that if there is no error on some abstraction, the system is verified without examine the original one. Of course, if all abstractions contain errors, the concrete system should be checked. When a defect is found on an abstraction, the next abstraction will be used. However, the multiple abstraction does not reduce the state space of the next search as in abstraction refinement [6].

In this report, we propose a new analysis method as an improvement of the former method [5] to analyze system variables so that weak ones are eliminated to make the model simpler. It takes the *convergence* into account to analyze the variable dependency in domain abstraction. Moreover, we applied the methods to symbolic model checking when states (and the transition relation) of the model can be represented as an expression of system variables, and then can be simply abstracted by domain abstraction.

Besides, a method that takes the advantages of the abstraction refinement [6] and multiple abstraction [2, 3] is also proposed. In other words, the new so-called multiple abstraction refinement (MAR) method is a combination of multiple abstraction and abstraction refinement. It currently *supports only for directed model checking*, a research direction that considers only invariant properties.

The rest of paper is constructed as follows: Section 2 is to provide some technical background, Section 3 is for the convergent propagation variable dependency analysis, Section 4 is for multiple abstraction refinement. The conclusion and future work is in the last section.

2 Technical Background and Related work

2.1 Symbolic model checking

Model checking is a problem to answer is a given model M satisfies the checking property.The problem has two related sub-problem: model representation and model checking approach. There are two main model representation: symbolic model checking, that represents the state space as symbolic expressions, and explicit state model checking, that represent states explicitly. When the state spaces in symbolic model checking are in expressions, In symbolic model checking, the state spaces are in symbolic expressions. one can capture more states than that of the explicit state model checking [7].

Definition 1 (Transition systems). *A finite state transition system is a tuple* $M = (S, S_0, T)$*, where* S *is a finite set of states,* $S_0 \subseteq S$ *is a set of initial states and* $T \subseteq S \times S$ *is a transition relation.*

In symbolic model checking, a state $s_i \in S$ is represented as an expression over a non-empty set of system (or state) *variables* $X = (x_0, x_1, \ldots, x_n)$, where each variable x_i ranges over a finite domain D_i, $s_i = f(x_0, x_1, \ldots, x_n)$.

2.2 Abstraction

Abstraction is the concept using in model checking to make the model smaller. In general, if the abstract system is correct, so does the concrete system [8]. But there is no guarantee that there is an error in the concrete system if there is an error in the abstraction. For example, an abstraction of the concrete system [9] is depicted in Fig. 1, in which the error concrete state 10 is unreachable in

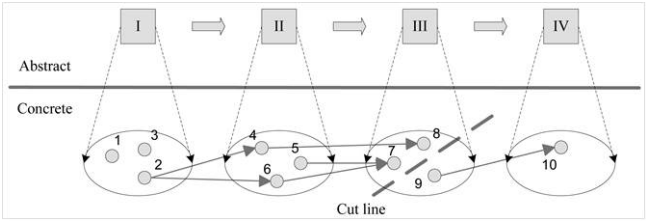


Fig. 1. An abstraction system [9]

the concrete system even though its abstract state IV is reachable from the (abstract) initial state.

Support that, there is a set of surjections $H = (h_1, h_2, \ldots, h_n)$, where each h_i maps a finite domain D_i to another domain $\hat{D}_i$ with $|\hat{D}_i| \leq |D_i|$. An domain abstraction of a transition system is defined as follows [2, 3].

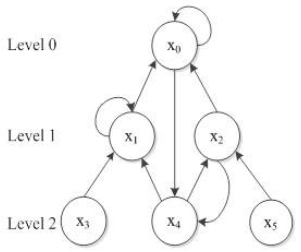
Definition 2 (Homomorphic abstraction). *A homomorphic abstraction of a transition system* M *is* $\hat{M} = (\hat{S}, \hat{S}_0, \hat{T})$*, where* $\hat{S} = H(S) = \{\hat{s}_i \mid \hat{s}_i = h_i(s)\}$*,* $\hat{S}_0 = H(S_0)$*, and* $\hat{T} = \{(\hat{s}_i, \hat{s}_j) \mid (s_i, s_j) \in T, \hat{s}_i = h_i(s_i), \hat{s}_j = h_j(s_j)\}$.

For example, if we want to omit a system variable x_k of a model in making an abstraction, we can simply abstract its domain to an empty domain, and apply the above definition. It has been proved that, the (homomorphic) abstraction preserves the checking properties of the concrete system [2, 3].

Based on that definition, the *restriction operation* can be defined as: let S_M is a set of states of a model M, $S_{\hat{M}}$ is a set of abstract states of an abstract model $\hat{M}$ of M, the restriction of S_M according to $S_{\hat{M}}$ is $\{s_i \in S_M \mid \hat{s}_i = h_i(s_i), \hat{s}_i \in S_{\hat{M}}\}$.

2.3 Variable Dependency Analysis

There are many way to abstract a model such as predicate abstraction [10], that uses predefined predicates to abstract the system, and lazy abstraction [11], that prunes branches of execution by abstracting and refines the pruning to allocate errors. Suppose that, the state space of a model in symbolic model checking can be defined over a set of system variables, Qian et al. [2] proposed that, pruning 'weak' system variables will set off the main features of the model. They have shown that, a simple variable dependency analysis can be used to find 'important' and 'weak' variables. The research in [5] suggested to use a propagation approach to the variable dependency analysis and achieved good results.



$$
\begin{aligned}
&E(x_0) = 1,\\
&E(x_1) = E(x_0)/3 = 0.33,\\
&E(x_2) = E(x_0)/3 = 0.33,\\
&E(x_3) = E(x_1)/3 = 0.11,\\
&E(x_4) = E(x_1)/3 + E(x_2)/2\\
&\qquad - 0.01/(1 + Lv(x_0))\\
&\qquad - 0.01/(1 + Lv(x_2)) = 0.26\\
&E(x_5) = E(x_2)/2 = 0.17.
\end{aligned}
$$

Fig. 2. A variable dependency graph [5] and its importance factors

Let $Prv(x_i)$ be the set of all variables that defines x_i in the relation transition, $Nxt(x_i)$ be the set of variables that are defined by x_i, $varOf(f)$ be the set of variables in the expression f and φ be the checking property of the model. The variable dependency graph is defined as follows.

Definition 3 (Variable Dependency Graph). *A (variable) dependency graph is a directed graph $VDG = (V, E)$, where V is a set of variables as vertices, E is a set of edges $E = \{x_i \to x_j \mid x_j \in Nxt(x_i)\}$.*

We also defined a mapping Lv to map each variable to a 'layer'. It is defined as $Lv(x_j) = 0$ iff $x_j \in varOf(\varphi)$ and $Lv(x_j) = Lv(x_i) + 1$ if $x_i \in Prv(x_j)$.

In the case when x_i and x_j depend on each other, the propagation will be applied to calculate Lv. In the graph in Fig. 2, variable x_0 depends on itself and on x_4, x_2 and x_4 depends on each other.

Let $in(y)$ be the in-degree of the vertex y, the importance factor of a variable is defined as follows.

$$
E(x) = \begin{cases} 1 & Lv(x) = 0 \\ \displaystyle\sum_{y \in Nxt(x) - \{x\}} \frac{E(y)}{in(y)} - \alpha(x) & Lv(x) > 0 \end{cases} \tag{1}
$$

where $\alpha(x)$ is the *counter-effect* on x and is defined as

$$
\alpha(x) = \beta \sum_{y \in Prv,\ \mathrm{Lv(y)} < \mathrm{Lv(x)}_{(x)}} \frac{1}{1 + Lv(y)} \tag{2}
$$

and $\beta \in [0 \dots 1]$ is an adjustment, that controls how much the variables in a higher level affects a variable in a lower level. For example, let $\beta = 0.01$, the importance of variables in the graph in the left of Fig. 2 are given on the right of it.

2.4 Counter-example Guided Abstraction Refinement - CEGAR

In order to find out the truth about a counter-example in an abstraction, Clarke et al. [6] proposed a method so-call Counter-Example Guided Abstraction Refinement (CEGAR). It is to determine if a counter-example on an abstract model can be confirmed on the concrete system. The main idea is that the search is performed on an abstraction and any found counter-example will be refined and confirmed on the concrete system to report an error. If there is no error on the abstraction, the concrete system has been proved its correctness. In this case, the performance of the model checking depends on the quality of the abstraction and the refinement algorithm. Moreover, the method faces the non-strong connectivity inside abstract state (See Fig.4 in [2]), when an abstract state represents a set of non-strong connected concrete states. The abstract state III in Fig. 1 is an example of the the issue.

2.5 Multiple Abstraction

Another abstraction approach had also been proposed by Qian et al. in [2, 3]. It is a multiple abstraction strategy to reduce verification efforts. Based on an assumption that there are a series of abstractions $M_0 \preceq M1 \preceq \dots$ such that some states $s_i^1, s_i^2, \dots$ in M_{i+1} (more concrete) are abstracted into only one abstract state s_i in M_i (more abstract). The model checking searches through abstractions from $M_0, M_1, \dots$ for any violation (in those abstraction). It can stop if there is no error found in any M_i. Otherwise, the search continuing detects the error on the remanding abstractions and even at last on the concrete system. The proposed method treats all abstractions as separated models and in the sequential order. However, it can take into account the counter-example found on the previous search as a heuristic to guide the search on the next abstraction (or concrete model). Of course, the problem of non-strong connected abstract state mentioned above still exists when it may lead the next search into the wrong direction. In the worst case, the search must be executed on the concrete model to find the actual counter-example. Although its performance is very good [2], it can be improved by concentrating on (only) dangerous areas in the next abstraction.

2.6 Other Direction on Reducing Model Checking Effort

There are some other directions on reducing model checking effort such as Partial Order Reduction [12], SAT-solving [13]. The latter is known as Bound Model Checking - BMC. In this work, we only focus on abstraction approach to reduce the size of the state space. We may try to apply the idea in this work to some other directions, including BMC in the future.

3 Convergent Variable Dependency Analysis

This section is to propose the convergent variable dependency analysis and some experimentation.

3.1 The Method - VRK

In this section, we try to add the convergent approach to the propagation approach mentioned in Section 2.3. In this approach, it captures the counter-effect from all related (or connected) variables. The idea of the convergent is actually came from the Page Rank algorithm [14], that had been used in the Google search engine to calculate the degree of related web pages for each web page.

Let d be the damping factor to the convergent (default value is 0.85), the formula to calculate the importance factors of a variable dependency graph $VDG = (V, E, Lv)$ is as follows.

$$PE(x) = \frac{1-d}{|V|} + d\left(\sum_{y \in Prv(x)} \frac{PE(y)}{|Nxt(y)|}\right) \tag{3}$$

$$E(x) = PE(x) + (Max(Lv) - Lv(x))/Max(Lv) \tag{4}$$

Let try to apply the new formula to the graph in Fig. 2.

(+) Apply equation (3):

$PE(x_0) = 0.16243$, $PE(x_1) = 0.17687$,
$PE(x_2) = 0.1017$, $PE(x_3) = 0.025$,
$PE(x_4) = 0.13348$, $PE(x_5) = 0.025$.

(+) Apply equation (4):

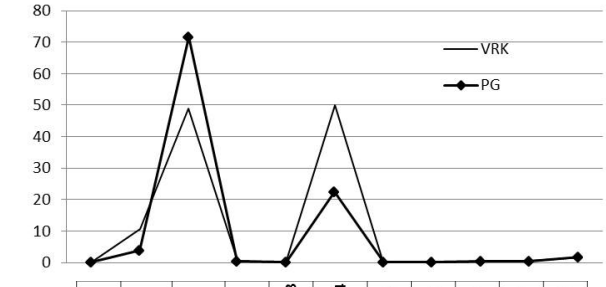
$E(x_0) = 1.16243$, $E(x_1) = 0.67687$,
$E(x_2) = 0.6017$, $E(x_3) = 0.025$,
$E(x_4) = 0.13348$, $E(x_5) = 0.025$.

Discussion: (1) It easy to see that the relationship among variables in this case is different to that of the proposed method in [5] (See Fig. 2). We captured more about the *inter-effect* between any two variables. (2) Especially, the variable's level on the graph still has its effects to the variable important degree similar to the method in [5]. Convergent variable dependency analysis simply has focused on both sides.

3.2 Experimentation

To compare our new proposed variable analysis in abstraction with previous work [2, 3, 5], we re-use models from those works. They are *peterson*, *leader-election*, *needham* protocols and *sender-receiver* communication system. They are actually 'buggy' models.

The variable analysis algorithm is made into NuSMV (http://nusmv.fbk.eu), a famous symbolic model checking tool. We also re-use the guided search (A*)



	leader-3	leader-4	leader-5	needham	peterson-3	peterson-4	sender-05	sender-10	sender-20	sender-30	sender-50
VRK	0.076	10.644	48.929	0.535	0.050	50.075	0.040	0.080	0.340	0.470	1.740
PG	0.085	3.890	71.615	0.515	0.035	22.515	0.030	0.080	0.330	0.480	1.750

Fig. 3. Experimentation on A* using **VRK** and **PG** variable analysis methods

algorithm proposed in [2, 3] to compare the abstractions made from our new approach and the previous ones. When [5] has shown that the propagation analysis is generally better than the simple analysis proposed in [2, 3], we will only compare the new algorithm named **VRK** with the one in [5] named **PG** in this experiment. The experiment is taken on a Core 2 Duo 2.93 GHz with 2 GB of RAM computer running Ubuntu 13.10. The results are the average of running time in seconds. The experimentation is in Fig. 3.

In this experiment, 40% of the least important variables ('weak' variables) are abstracted away to make the abstraction. It is easy to see that, the searching time of A* using abstraction generated from **VRK** is generally smaller than that of from **PG**. The chart on the figure also shows that, the strength of running time of A* using **VRK** is smoother than that of **PG**.

Note that, a other configuration may results differently with that experimentation. The experimentation of abstracting away 90% of 'weak' variables is in Fig. 4. In this case, they are almost identical. We believe that, the topology of the

	leader-3	leader-4	leader-5
VRK	0.041	3.811	61.413
PG	0.047	3.808	61.407

Fig. 4. A* with 90% of weak variables abstracted away

dependency graphs (of those models) may effect the calculation on convergent. For example, after analyzing, some first variables of the most important ones are the same in both approaches, thus the removing of them makes similar abstrac-

tions in both case and the checking will have the same effort. Therefore, the **PG** algorithm that has lest analysis computation effort, may take advantages. The similar issue has also been reported in [15].

Discussion: The VRK method in general has more computation effort than the PG method when it takes into account the relationship on all related variables in the system. This effort sometime effective, but in some other cases, it gives no advantages to the whole checking process. But when the development strength of the running time on the abstraction using VRK is smooth, it is likely better than other methods in general. We suggest to use VRK in general.

4 Multiple Abstraction Refinement - MAR

In this section, we present our new proposed approach named **MAR** that takes advantages from both CEGAR [6] and multiple abstraction methods [2, 3].

4.1 MAR algorithm

Suppose that there is a series of abstractions $M_0 \preceq M1 \preceq \ldots$ as in Section 2.5. The search is started in the M_0. If there is a violation (of the checking property), a series of counter-example refinements will be performed on the rest of abstraction M_i. If the counter-example is confirmed on all abstractions and then on the concrete system S, it is the actual counter-example. Otherwise, the search will continue on M_0 for another error. If there is no (or no more) counter-example, the concrete system S is proved to satisfy the checking property.

The flows of the algorithm are in Fig. 5. On the right, it is to show how

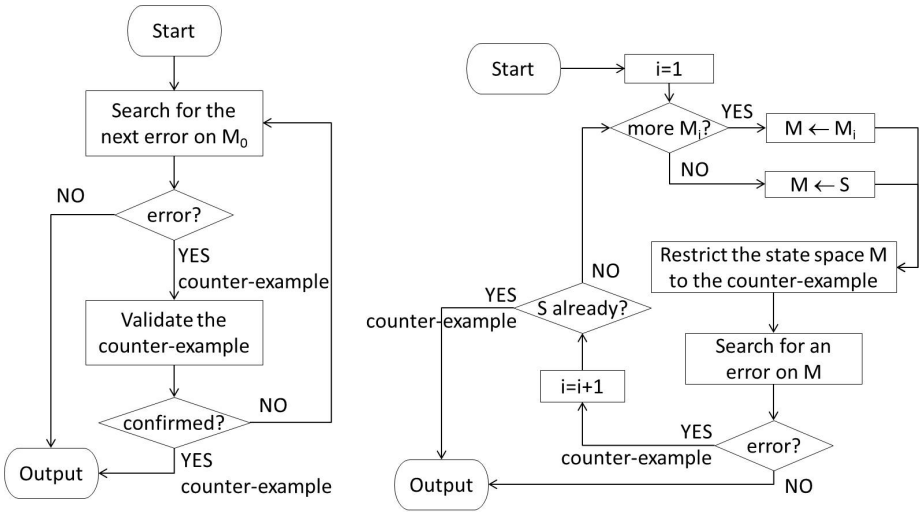


Fig. 5. The MAR algorithm

to refine error on all abstraction M_i, and even on the concrete system S. The restriction operation is applied on the state space of the checking model based on the counter-example of the previous (more abstract) model. The difference to that of CEGAR is that instead of refining the error on the concrete system S immediately when an error is found on the abstraction M_0, it refines the error on the 'more concrete' abstraction M_1 (and then on M_2 and so on). In the worst case, the concrete system will be used to search. General speaking, it delays the refinement on the concrete system as long as possible. So, if there is a false alarm, this algorithm may have a chance to find out the truth without searching on the concrete 'huge' state space. In fact, MAR and CEGAR have the same idea when we only have only one abstraction M_0 such that they will refine the counter-example on the concrete state space immediately when it is found on the abstraction. Technically, they are not the same when CEGAR refines error on the concrete system using some complicated algorithms, and our MAR just search on the restricted area made from the (abstract) counter-example on the concrete state space. In this work, we assume that our MAR with one abstraction is CEGAR.

The MAR algorithm also differs to the multiple abstraction proposed in [2, 3] when it uses the abstractions only for validating an error found on the previous abstraction. At every iteration, instead of searching the state space of M_i for a (new) error as in multiple abstraction method, it restricts the state space of M_i to the area that comes from the previous counter-example. The idea is that, when the counter-example found from the previous search is small, the restriction will help the algorithm to reduces the verification effort.

Remark 1. The MAR algorithm is sound and complete.

It is easy to prove the remark. When the algorithm returns no error, based on the algorithm, if there is no error on the abstraction M_0, or if there are some errors on M_0 but were rejected in the validation step, the concrete system has no error. When the algorithm returns a counter-example, it is confirmed on the concrete system, hence it is a counter-example.

4.2 Experimental Results on Multiple Abstraction Refinement

In this section, we compare our algorithm (MAR) to the that of [2, 3] (named SAGA) using the variable dependency analysis VRK. To compare to CEGAR, we assume that our algorithm acts similar to CEGAR if there is only one abstraction M_0. We named that algorithm is MAR-1 in this experiment. The benchmark we use in this section is provided on the main page of NuSMV. They are named *brp*, *dme*, *gigamax*, *msi_wtra*, *periodic*. All of them are correct models, so we can expect that only a few level of abstraction is enough to find the truth. The environment is the same as stated in Section 3.2.

In Fig. 6, we compare the algorithms. The abstractions are built by removing 80%, 60%, 40% and 20% of 'weak' variables, respectively. It is easy to see that, they almost have the same performance, when (in many cases) only the first abstraction is searched and no error ever returned.

	brp	dme1	dme2	gigamax	msi_wtrans	periodic
MAR	0.310	0.040	0.113	0.017	0.072	0.010
SAGA	0.333	0.041	0.112	0.026	0.073	0.010
MAR-1	0.265	0.040	0.113	0.017	0.072	0.010

Fig. 6. MAR vs. SAGA vs. MAR-1 on correct systems

We also re-use the benchmark in Section 3.2. When they are 'buggy' systems,

	leader-3	leader-4	leader-5	needham	peterson-3	peterson-4	peterson-5	sender-05	sender-10	sender-20	sender-30	sender-50
MAR	0.050	7.515	84.810	1.285	0.030	4.070	222.900	0.805	9.200	79.835	179.775	timeout
SAGA	0.085	79.570	timeout	0.160	0.065	6.765	timeout	0.050	0.115	0.530	0.940	7.110
MAR-1	0.040	7.690	81.920	1.310	0.040	4.090	251.870	0.790	9.420	80.070	178.840	timeout

Fig. 7. MAR vs. SAGA vs. MAR-1 on 'buggy' systems

the search in M_0 will return some error and the validation step will search on all the abstractions and the concrete system. The experimentation results in Fig. 7 confirms that the proposed method outperform both multiple-abstraction search (SAGA [2, 3]) and single-abstraction refinement (MAR-1 as CEGAR). Only one exception that SAGA is extremely good on *sender* systems. [15] mentioned that this is a special case when the topology of the state space is skew to some buggy locations and thus the guided search algorithms like SAGA are the best to be used.

To not be bias to the variable analysis methods, we repeated out experiments using random abstraction, in which the importances of system variables are assigned randomly. The experimentation results in Fig. 8 shown that the MAR

	leader-3	leader-4	leader-5	needham	peterson-3	peterson-4	peterson-5	sender-05	sender-10	sender-20	sender-30
MAR	0.050	7.780	82.130	1.310	0.020	4.410	224.200	0.810	9.670	79.890	180.860
SAGA	0.090	8.710	82.300	2.250	0.030	4.960	timeout	5.430	189.220	timeout	timeout
MAR-1	0.050	7.790	82.050	1.310	0.030	4.080	224.400	0.810	9.900	79.680	181.880

Fig. 8. MAR vs. SAGA vs. MAR-1 on 'buggy' systems using random abstraction

is still the best.

Discussion: When we model check a system in practice, we actually do not known it is correct or not. The abstraction method MAR is a good choice when it works similarly to CEGAR or SAGA in correct systems and outperforms them in buggy systems. Moreover, it also works well on arbitrary abstraction such as random abstraction.

5 Future Work and Conclusions

In this paper, a new convergent variable analysis method has been proposed to enrich abstraction technique in model checking. It employees the idea of the convergent computation using in the Page Rank algorithm [14] to calculate the importance of system variables in domain abstraction. The less important variables can be omitted to make abstractions to reduce the state space. The experimentation shows that the new analysis generally gives a better solution in abstraction than some algorithm proposed in previous work. The study on when to use the new analysis and when to use the others is not carried out in this work and should be in the near future.

A new model checking based on counter-example refinement and multiple abstraction has also been proposed. It takes advantages from both methods such that it uses multiple abstraction to confirm a counter-example found on the finest abstraction and the idea of counter-example refinement to refine the counter-example. The modification is instead of refining the error on the concrete system, the new method validates it in the abstractions first. The modification allows the checking process works around the abstractions which are smaller than the concrete system and hence reduces the effort. The experimental results confirmed our assumption. Of course, our algorithm still faces the same problem with CEGAR and multiple abstraction algorithms that if there are errors, it still has to refer to the concrete system in generating a real counter-example.

Nevertheless, when we only support for directed model checking, we need to extend our approach to support other checking direction. It should be in our future work.

Acknowledgment

This research was supported by Vietnam National University - Ho Chi Minh City (Ho Chi Minh City, Vietnam) under the grant number C2013-20-07.

References

1. Menzies, T., Owen, D., Richardson, J.: The strangest thing about software. IEEE Computer **40**(1) (2007) 54–60
2. Qian, K., Nymeyer, A.: Abstraction-based model checking using heuristical refinement. In: Proc. the 2^{nd} Int. Symp. on Automated Technology for Verification and Analysis (ATVA'04). Volume 3299 of LNCS., Springer-Verlag (2004) 165–178

3. Qian, K., Nymeyer, A., Susanto, S.: Experiments in multiple abstraction heuristics in symbolic verification. In: Proc. the 6^{th} Sym. on Abstraction, Reformulation and Approximation (SARA'05). Volume 3607 of LNAI. (July 2005) 290–304
4. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Directed explicit model checking with HSF–SPIN. In: Proc. the 8^{th} Int. SPIN Workshop on Model Checking of Software (SPIN'01). Volume 2057 of LNCS., Springer (May 2001) 57–79
5. Bui, T.H., Dang, P.B.: Yet another variable dependency analysis for abstraction guided model checking. In: Proc. SEATUC 2012. (Mar 2012)
6. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. the 12^{th} Int. Conf. on Computer Aided Verification (CAV'00). Volume 1855 of LNCS., Springer-Verlag (2000) 154–169
7. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Proc. the 5^{th} Annual IEEE Symp. on Logic in Computer Science, Washington, D.C., IEEE Computer Society Press (1990) 1–33
8. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. Proc. the 19^{th} ACM SIGPLAN-SIGACT Sym. on Principles of Programming Languages (1992) 343–354
9. He, F., Song, X., Hung, W.N., Gu, M., Sun, J.: Integrating evolutionary computation with abstraction refinement for model checking. IEEE Transactions on Computers **59**(1) (2010) 116–126
10. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proc. the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation, ACM Press (2001) 203–213
11. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. the 29^{th} SIGPLAN-SIGACT Symp. on Principles of Programming Languages, ACM (2002) 58–70
12. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Proc. the 6^{th} Int. Conf. on Computer Aided Verification (CAV'94). Volume 818 of LNCS., London, UK, Springer-Verlag (1994) 377–390
13. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. In Strichman, O., Biere, A., eds.: Electronic Notes in Theoretical Computer Science. Volume 89., Elsevier (2004)
14. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Seventh International World-Wide Web Conference (WWW 1998). (1998)
15. Bui, T.H., Nymeyer, A.: Heuristic sensitivity in guided random-walk based model checking. In: Proc. the 7^{th} IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM'09), IEEE Computer Society (Nov 2009) 125–134