AUTOMATICALLY INFERRING STRUCTURE CORRELATED VARIABLE SET FOR CONCURRENT ATOMICITY SAFETY

Long Pang[1], Xiaohong Su[2], Peijun Ma[3] and Lingling Zhao[4]

School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China

ABSTRACT

The atomicity of correlated variables is quite tedious and error prone for programmers to explicitly infer and specify in the multi-threaded program. Researchers have studied the automatic discovery of atomic set programmers intended, such as by frequent itemset mining and rules-based filter. However, due to the lack of inspection of inner structure, some implicit sematics independent variables intended by user are mistakenly classified to be correlated. In this paper, we present a novel simplification method for program dependency graph and the corresponding graph mining approach to detect the set of variables correlated by logical structures in the source code. This approach formulates the automatic inference of the correlated variables as mining frequent subgraph of the simplified data and control flow dependency graph. The presented simplified graph representation of program dependency is not only robust for coding style's varieties, but also essential to recognize the logical correlation. We implemented our method and compared it with previous methods on the open source programs' repositories. The experiment results show that our method has less false positive rate than previous methods in the development initial stage. It is concluded that our presented method can provide programmers in the development with the sufficient precise correlated variable set for checking atomicity.

KEYWORDS

Correlated variables, frequent subgraph mining, program dependency graph

1. INTRODUCTION

The multi-threaded program has become pervasive during the popularization of multi-core hardware. The shared memory can be simultaneously accessed by the parallel threads. The concurrent threads communicate implicitly the local value of variables by directly accessing the shared variables. While the concurrency dramatically improves the efficiency of execution, it is also the source of interleaving complexity. As a result, the concurrent programs are prone to bugs. It is more difficult to guarantee the safety than the single-threaded programming.

The atomicity [1] is one of the basic properties for the safe shared memory program. It is traditionally the terminology for the transaction in the field of database. In database, atomicity means that any interleaving execution of group of transaction can be mapped to a serial execution of the same group of transitions with the underlying logics equivalent. In shared memory concurrency programs, the atomicity guarantees that the atomic set of variables are consistent among any interleaving execution of concurrent threads. In other words, any interleaving execution of accessing atomic set of variables is semantically equivalent to a serial execution of accessing the same atomic set of variables. The atomicity is also called serialization. In multi-threaded programs, the atomicity violation will break the consistency of concurrent threads resulting in the unintended bugs. However there is no explicit syntax in programming language to declare the atomic set of variables. They are hidden inside programmers' intention.

In previous research work, some researchers focus on detection of single variable related concurrency safety bugs, such as data race. Dinning et al. [2] and Savage et al. [3] investigate the well-known lock-set data race detector. Netzer et al. [4] proposed the happen-before race detector. Miller et al. [5] formalized races and classified the atomicity as high level data race. There were various static or dynamic enhancements of data races [6-9]. However, the absence of single variable race is not sufficient to prevent the unexpected thread interleaving. Then researchers study detection of the violation of atomicity for multiple correlated variables. Flanagan et al. made use of type system and dynamic analysis to check the atomicity violation [10]. This method requires user to specify the atomic set of multiple variables by annotation. In recent research work, researches concentrate on the automatic inference of atomic variables set. Xu et al. [11] brought forward the region hypothesis to infer the computational unit as the atomic set. Lu et al. [12] proposed the automatic discovery of correlated variables by the frequent item mining.

This paper proposes a dependency graph mining approach to automatically infer atomic set of multiple correlated variables from program source code. We find that not only the layout of variable access, but also the pattern of variable access contribute to the manifestation of the correlated variables intended by programmers. The traditional methods [11][12] take these two factors into account separately. Our aim is to integrate the two factors by dependency graph mining. We represent the potential logic correlated relations in the terms of the variable's dependency relation. We formulate the inference problem for atomic set of correlated variables into a subfigure mining problem on the program dependency graph. This paper makes the following contributions:

- Dependency formulation of the variable access layout and the variable access pattern for variables' logical correlation. These two factors are unified into dependency subgraph.
- Automatic inference algorithm for the correlated variables set. We propose the graph mining algorithm for dependency subgraph. It is used to exhibit the correlated variables for small scale of program base.
- Experiment results. To simulate the development process, we evaluate our method in the initial version of open source program in small scale. The results show that our approach improves the precision of correlated variables detected for atomicity checking.

2. CORRELATED VARIABLES FOR ATOMIC SET

The correlation among variables is intended by the programmers in mind. There is no explicit syntax to express this relationship. The atomic set of variables is needed to preserve the consistency through the arbitrary interleaving among the concurrent threads. The atomic set of variables must be some kinds of correlated variables. However, it is unnecessary to ensure the atomicity for all of the correlated variables. For example, the variables for synchronizing threads are correlated variables but not atomic set, such as two correlated flag variables in Peterson's algorithm [13]. Our aim is to identify the correlated variables for atomic set.

In shared-memory multi-threaded programs, the thread computation is synthesized by the communication of shared variable accesses such as write and read. The concurrency of thread execution is control flow centred. It is common for the programmers to suppose that some code segment will execute atomically as the single threaded program. But the concrete interleaving of concurrency thread may break these assumptions, which cause the atomicity violation bugs. It is inherently more difficult than single-threaded programming to debug. The correlated variables for atomic set are helpful for the programmers to comprehend the data centred concurrency and locate the cause of bugs in the development stage, even avoiding the potential bugs.

There is two directions to infer the atomic set. Xu et al. [11] made use of the read-of-write of shared variable dependency, local variable and control dependency as rules to divide the shared variables into computation unit as atomic set. It lacks of support from source code. On the other hand, Lu et al. [12] used the frequent item mining algorithm to detect the correlated variables in source code as atomic set. However, they only analysed the dependency but not used this important clue in the mining algorithm. Our work synthesizes these two factors to extract the desired atomic set for the initial stage of software development..

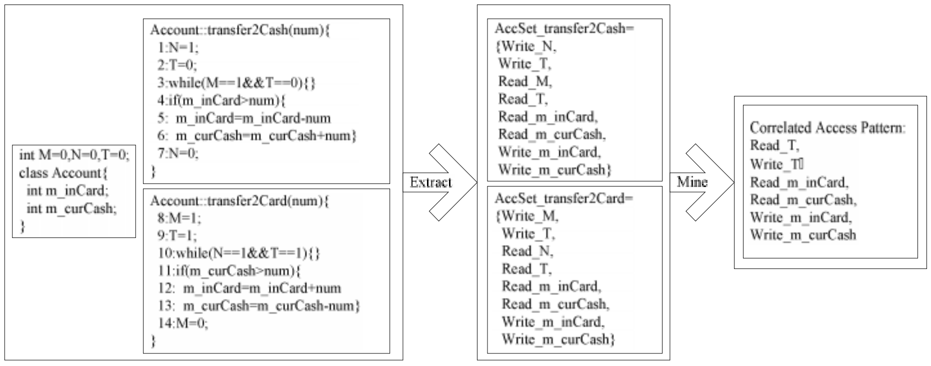
2.1. Semantics analysis of correlated variables

The scalar variable can specify one character of the fact. The vector of variables describes some complex properties of the target. In summary, the programmers are used to the following styles to model the objective world:

- Conditional Relation: a conditional variable describe a state of a content variable. The update of content variable is determined by the conditional variable. At the same time, the effect of the update is also reflected by the conditional variable. For example, consider the string object consisting of fields: length and characters array. The length variable is conditional guard for the characters array. These two variables are updated in the same basic block. The array of characters is control dependent on the length variable.
- Simple Parallel Relation: The objective fact has different aspects to be recorded in the program. These attributes should be updated atomically. For example, there are three variables to describe RGB (Red, Green, Blue) color. These variables are usually accessed in the same basic block.
- Complex Composite Relation: The correlated variables compose the complex data structure such as linked list, tree and graph. The variables may be read separately in the different basic block. The variables are updated in the same basic block.

The Figure 1 shows the example of correlated variables mined by MUVI. [12]. However, the result contains the user-defined synchronization global variable accesses, such as Read_T and Write_T. The reason is that the access set does not discriminate the internal logical structure. We still address this problem in the following section.

The discussion is related to the meaning of correlation with respect to intention in programmer's mind. They are expressed by different behaviour during the access. The following will discuss the behaviours exhibited by the correlated variable access.



```
int M=0,N=0,T=0;
class Account{
  int m_inCard;
  int m_curCash;
}
```

```
Account::transfer2Cash(num){
  1:N=1;
  2:T=0;
  3:while(M==1&&T==0){}
  4:if(m_inCard>num){
  5:  m_inCard=m_inCard-num
  6:  m_curCash=m_curCash+num}
  7:N=0;
}
```

```
Account::transfer2Card(num){
  8:M=1;
  9:T=1;
  10:while(N==1&&T==1){}
  11:if(m_curCash>num){
  12:  m_inCard=m_inCard+num
  13:  m_curCash=m_curCash-num}
  14:M=0;
}
```

```
AccSet_transfer2Cash=
{Write_N,
 Write_T,
 Read_M,
 Read_T,
 Read_m_inCard,
 Read_m_curCash,
 Write_m_inCard,
 Write_m_curCash}
```

```
AccSet_transfer2Card=
{Write_M,
 Write_T,
 Read_N,
 Read_T,
 Read_m_inCard,
 Read_m_curCash,
 Write_m_inCard,
 Write_m_curCash}
```

```
Correlated Access Pattern:
Read_T,
Write_T
Read_m_inCard,
Read_m_curCash,
Write_m_inCard,
Write_m_curCash
```

Figure 1 Program Example and Correlated Pattern

2.2. Behavior analysis of correlated variable

Due to lack of syntactic description of correlated variables, the behaviour of access of variables is the direct approach to inference the correlation among them. In the imperative programming language, there are three basic structures: sequence, selection and loop. It is common for programmers to code the access of correlated variables in togetherness. How to define the togetherness is the critical criterion to identify the correlation of atomic set.

Lu et al. [12] used the absolute distance *MaxDistance* to filter out the un-togetherness access. The source code is flat representation of variable access behaviours. In the extreme condition, it is possible to dispatch the two accesses of the correlated variables into two positions across the *MaxDistance* distance. Xu et al. [11] utilized the data and control dependency on the execution traces to discriminate the atomic set. Because the initial stage of development can't generate the concrete execution trace, it is not adaptable for our need. The straightforward solution is to substitute the trace dependency graph (PDG) with the program dependency for source code. Although the dependency is a candidate of dimension for PDG to discriminate the correlated variables, it is too strict for inference of correlated variables in PDG. For example, the order of accessing simple parallel relation varies with the occurrences with the same correlated semantics. The above suggests that the dependency method and the mining method can be very much complementary to each other.

Our solution is to integrate the data dependency and the control dependency into the criterion for the togetherness of the correlated variables for atomic set. To relax the constraints on the simple parallel relation, our dependency graph consists of the data dependency and the control dependency excluding the control flow graph. The behaviour of correlated variables can be described by the following classical dependency definition.

- Control Dependency [14]: In CFG, the node X is control dependent on the node Y iff (1) there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and (2) X is not post-dominated by Y.
- Data Dependency[14]: In CFG, the node X is data dependent on the node Y iff (1) the node X and the node Y are both definition statement and (2) the value in node Y must be evaluated before the node X.

The control dependency defines the boundary of the potential correlation and relaxes the strictness of the source code sequence. The data dependency links the possible correlated variable access by the propagation of related control dependency. For example, the conditional relation consists of two parts. The first part is the computation of the predicate of condition. The second part is the control decision part. The first part is linked with the second part by the data dependency. The if-then-else body of the second part is linked by the control dependency. The left two semantics of correlation is similar to the conditional correlation case. Therefore, the control and data dependency is sufficient to represent the semantics of correlation intended by programmers.

3. DEPENDENCY GRAPH MINING ALGORITHM

3.1. Overview

The aim of dependency graph mining is to inference more structure logics among variables than flat set of variables, without lose of the generality of correlation. The graph is the precise representation of the relationship among variables such as data dependency and control dependency. However, it is not practical to mine the program dependency graph (PDG) directly due to its tightness. We introduce the simplified PDG to preserve the generality of variable correlations. The Figure 2 shows the main framework. We will discuss the details in the following subsections.

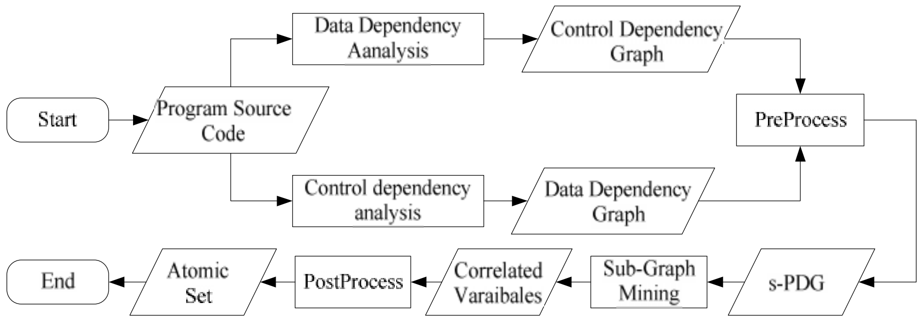


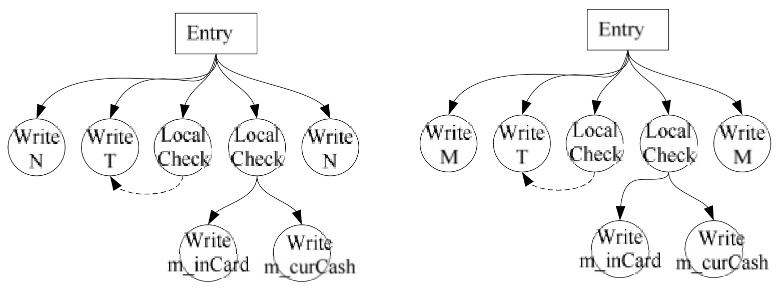
Figure 2 Sub-Graph Mining Based Atomic Set Inference Framework

3.2. Simplified program dependency graph

We present the simplified program dependency graph(s-PDG) to represent the data and control dependency. The s-PDG combines the control dependency and the data dependency graph excluding the control flow graph. The Figure 4 is the corresponding s-PDG of the Figure 1. The solid edge means the control dependency. The dashed edge means the data dependency. The nodes for variable access include: the global variable access, the object's fields variable access and the judgment related local variable access. We present the algorithm for simplifying PDG in Figure 3.

```
INPUT: ControlDependencyGraph cdg_in, DataDependencyGraph ddg_in
OUTPUT: SimplifedPDG sPDG
GenerateSPDG(cdg_in,ddg_in,sPDG)
{
HashTable to_visit, visited;
Node cur_node,              //for traverse
        unified_localnode;  //for simplify the PDG
 visited.insert(entry);
 to_visit.insert(entry);
while(to_visit.hasElement()){
cur_node = to_visit.pop();
switch(cur_node.type){
  case GlobalAccess:
   sPDG.addNode(cur_node);
  sPDG.addEdge(cur_node,cdg_in(cur_node),ddg_in(cur_node));
   break;
 case LocalAccess:
   unified_localnode = new LocalCheckNode();
   sPDG.addNode(localnode);
   sPDG.addEdge(unified_localnode,
   cdg_in(cur_node),ddg_in(cur_node));
   break;
};
foreach child in cur_node.children do
  if(!visited.has(child))
    to_visited.insert(child);
}
```

Figure 3 Algorithm for Computing Simplified PDG



(a) s-PDG for Account::transfer2Cash (b) s-PDG for Account::transfer2Card

Figure 4 s-PDG example

3.3. Preprocess

During preprocessing, we build the simplified program dependency graph by collection the following behaviors of accessing variables:

- ***Access of global variable:*** the global variables are shared among the concurrency threads. In programmer's mind, some shared variables cluster together to synchronize and communicate the information among threads. The global variables are the candidates for the atomic set.
- ***Access of field variable:*** the field variables are organized under different parent types. For programmers, the field variables of the same type have the internal connectivity among them. However, the connectivity does not need to be atomic. The filed variables may become shared by its parent type variable. The correlations among field variables are also candidates for the atomic set.
- ***Local variable predicate:*** in the compiler intermediate language, the variables used in predicate is firstly fetched into the logical register and then computed as expressions. The branches are all control dependent on the final result. To improve the robustness, we separate the local predicate into two parts: its data dependency and its virtual node for judgment. It is a balance between the accuracy and the granularity of the correlation variables.

3.4. Frequent sub-graph mining and example

To extract the atomic set, we adopt the gSpan [15] to mine the s-PDG. The s-PDG abstracts the predicate for the judgment and loop statement. The directed s-PDG classifies the variable access into hierarchies with respect to the control dependency. The data dependency is the skeleton of the effect of target variable writes. Due to the set property of graph's successor relation, the nodes of parent node represent all possible sequence of the variable accesses. Meanwhile, the data and control dependency discriminates the logics of variables correlation avoiding blurring the programmer's intention.

The criterion for designing s-PDG is to balance the generalization of variable access pattern and the specialization of correlated variables. The Figure 5 shows the result of frequent sub-graph of s-PDG in Figure 4, which is mined by gSpan [15].

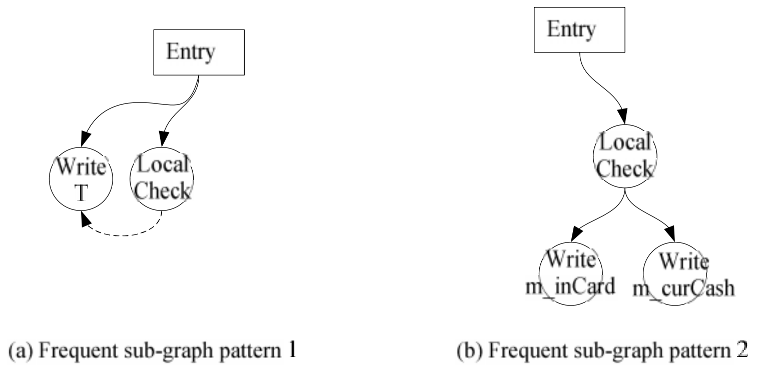


Figure 5 Result of Frequent Sub-Graph Mining

The Figure 5(a) is an instance of data dependency. There is only one shared variable candidate. Therefore, it is not the candidate for the atomic set. In the following subsection, we will discuss the details of post-processing. The Figure 5(b) is a candidate of correlated variables, *m_inCard* and *m_curCash*. Comparing to Figure 1, our method's result has excluded the shared global variable T for the user-defined synchronization.

3.5. The Generation and pruning of correlated variables

While the frequent sub-graph is the candidate for the atomic set, we need post-processing to filter out the unnecessary correlated variables for atomic set. The intuition is to get most correlated variables among the other candidates.

Subgraph isomorphism and subgraph support: for labelled graph, there is a label function l mapping a vertex or an edge to a label. A graph *g* is subgraph of another subgraph *g'* if there is a subgraph isomorphism from g to g' denoted as $g \subseteq g'$. For two labelled graphs g and g', a subgraph isomorphism is a injective function f: V(g)->V(g'), s.t. , (1) $\forall v \in V(g),$; and (2) $\forall (u,v) \in E(g)$, $l(v) = l'(f(v))$ and $(f(u), f(v)) \in E(g')$, $l(u,v) = l'(f(u), f(v))$. F is also called an embedding of *g* in *g'*. Given a graph dataset D={G_1,G_2,…,G_n} and a subgraph *g*, the supporting graph set of *g* is $D_g = \{G \mid g \subseteq G, G \in D\}$. Then the frequency is |Dg|/|D|. The frequency of the mined subgraph of correlated variables is greater than user defined *MinSupport*.

Variable Support and Correlation Confidence: the support of variable is the number of frequent graphs containing the variable. The confidence of correlation is the conditional probability of the composed variable support. The correlation confidence is a vector of its composed variables. We compute the mean of all elements in this vector. The user defined *MinConfidence* is used to filter out the correlation with too low mean.

Other pruning: as encountered in Figure 5(a), there are some single global correlations of data dependency. These frequent subgraphs are filtered out.

4. Experiment

We implemented a user-defined pass in LLVM[16] to extract the needed static information from source code, such as data dependency, control dependency and variable access. The outputs of extraction pass are Acc_Set and s-PDG respectively. We re-implemented the access pattern mining in MUVI[12] by the FPClose[17]. We implemented our method based on the gSpan[15]. We selected the PARSEC[18] as our benchmark. The Table 1 shows the result.

Table 1 Experiment result table

Program Name	Lines of Code	MUVI			Our Method		
		Correlation	False Positive	Time(s)	Correlation	False Positive	Time(s)
Stream Cluster	2333	167	21%	0.45	106	3%	23.5
Freqmine	2710	184	20%	0.65	117	2%	43.5
Facesim	1966	192	27%	0.41	124	1%	20.1
Bodytrack	11169	221	18%	2.32	137	5%	230.4

Although we detect the less correlation, the result indicates that our method is much more accurate than MUVI. Our method is based on the graph mining. While we improve the generalization by introducing abstracted s-PDG, the representation of graph is stricter than the set used in MUVI. In the future, we will focus on how to improve the generalization to detect more correlation with the same precision reserved.

5. Related works

5.1. Correlated variables generation and pruning

There are two strategies to inference the atomic set. The first approach is heuristic inference rules. Xu et al. [11] made use of true dependency (read-write of shared variable) to divide the dynamic PDG. It is direct and easy to get the atomic set. This rule is suitable for the mature programmer who strictly obeys the requirement of the programming styles. It is lack of support from other occurrences of related variables accessed. For the atomicity composition of library usage, Liu et al[19] inference the atomic set by two program dependencies. One is the accessed object’s field clustered by synchronized blocks. The other dependency is the fields transitively reachable by the field’s reference. To improve the efficiency, the thread-local objects are also filtered out by escape analysis. While this approach combines the support from other occurrences of the related variables, it is possible to lose the atomic set across the boundary of synchronization blocks.

The second approach is the data mining. There are many application of data mining in the field of software engineering [20]. MUVI [12] is the first representative of mining based inferring the correlated atomic set. This approach is the inspiration of our method. Due to the generalization of variable access in function, the logical structures are lost after mining. Therefore, to solve the problems of previous approaches, we integrate the heuristic rules into the abstraction of graph mining target.

5.2. Detection of concurrency bugs

The goal of our automatic inference approach of atomic set is to detect the concurrency bugs. We also give a overview of related work on this field. There are three kinds of method: static analysis, dynamic testing and the combination of the former two methods. Miller et al. [5] formalized races and classified the atomicity as high level data race. There were various static or dynamic enhancements of data races [6-9]. Flanagan et al. made use of type system and dynamic analysis to check the atomicity violation [10].

6. CONCLUSION

In this paper, we propose the s-PDG representation for variable correlation and a graph-mining based automatic inference algorithm for atomic set. The s-PDG makes a balance between the precise description of correlated variables' logical structures and the generalization of the correlation pattern. We also investigate the effect of our method on detecting atomic set. The experimental result indicates that our approach has lower false positive percentage than the set-based mining approach for atomic set recognition.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments and suggestions. This research was funded by National Natural Science Foundation of China under Grand No. 61173021.

REFERENCES

[1] C. Flanagan and S. Qadeer. (2003) "A Type and Effect System for Atomicity", Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, June.

[2] A. Dinning and E. Schonberg. (1991) "Detecting access anomalies in programs with critical sections", ACM/ONR Workshop on Parallel and Distributed Debugging (AOWPDD).

[3] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. (1997) "Eraser: A dynamic data race detector for multithreaded programs". ACM TOCS.

[4] R. H. B. Netzer and B. P. Miller. (1991) "Improving the accuracy of data race detection", In PPoPP.

[5] R. H. B. Netzer and B. P. Miller. (1992) "What are Race Conditions?:Some Issues and Formalizations", ACM Letters on Programming Languages and Systems, 1(1):74–88.

[6] D. Engler and K. Ashcraft. (2003) "RacerX: Effective, static detection of race conditions and deadlocks". In SOSP.

[7] M. Naik, A. Aiken, and J. Whaley. (2006) "Effective static race detection for java". In PLDI.

[8] Y. Yu, T. Rodeheffer, and W. Chen. (2005) "Racetrack: Effcient detection of data race conditions via adaptive tracking", In SOSP.

[9] J.D. Choi. (2002) "Effcient and precise datarace detection for multithreaded object-oriented programs", In PLDI.

[10] C. Flanagan and S. N. Freund. (2004) "Atomizer: a dynamic atomicity checker for multithreaded programs", In Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 256–267.

[11] M. Xu, R. Bodik, and M. D. Hill. (2005) A serializability violation detector for shared-memory server programs. In PLDI.

[12] S. Lu, S. Park, C.F. Hu, et al. (2007) "MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs", 21st ACM Symposium on Operating Systems Principles, SOSP.

[13] G. L. Peterson.(1981) "Myths About the Mutual Exclusion Problem", Information Processing Letters 12(3), 115–116.

[14] J. Ferrante, K. J. Ottenstein, J. D. Warren.(1987) "The program dependence graph and its use in optimization", ACM Transactions on Programming Languages and Systems (TOPLAS), 9(3): 319-349.

[15] X. Yan, J. Han. (2002) "gSpan: Graph-Based Substructure Pattern Mining" , by Proc. 2002 of Int. Conf. on Data Mining (ICDM'02).

[16] C. Lattner, V. Adve. (2004) "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar..

[17] G. Grahne and J. Zhu. (2003) "Efficiently using prefix-trees in mining frequent itemsets", In Proceeding of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03), Nov.

[18] C. Bienia, S. Kumar, J. P. Singh, K. Li. (2008) "The PARSEC benchmark suite: characterization and architectural implications", In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, October.

[19] P. Liu, J. Dolby, C.Zhang. (2013) "Finding Incorrect Compositions of Atomicity", FSE.

[20] Q. Taylor, C. Giraud-Carrier, C.D. Knutson. (2010) "Applications of data mining in software engineering", International Journal of Data Analysis Techniques and Strategies, 2(3): 243-257.

Authors

PANG Long born in 1984. PhD candidate in the School of Computer Science and Technology of Harbin Institute of Technology.His current research interests include alias analysis, concurrency nalysis,program analysis and software engineering.