出國報告（出國類別：參與國際會議）

抽象化字串驗證：SPIN/CAV 出國會議報告

String Abstractions for String Verification:

The report of attending the 18th International SPIN Workshop on Model Checking of Software and the 23rd International Conference on Computer Aided Verification

Yu, Fang (郁方)

服務機關：國立政治大學資訊管理系
姓名職稱：郁方, 助理教授
派赴國家：美國
出國期間：7/12-7/22, 2011
報告日期：August 10, 2011

國立政治大學發展國際一流大學及頂尖研究中心計畫 出國成果報告書（格式）

計畫編號[1]		執行單位[2]	資訊管理系
出國人員	郁方	出國日期	100 年 7 月 12 日至 100 年 7 月 22 日，共 11 日
出國地點[3]	2011 SPIN and CAV Conferences, Snowbird, Utah, United States.	出國經費[4]	NT 97,277

報告內容摘要(請以 200 字～300 字說明)

This report summarizes my presentation and selected invited talks, tutorials, regular paper presentations in the 18th SPIN Software Model Checking workshop and the 23rd International Conference on Computer Aided Verification. I conclude with my future research directions inspired by the talks and discussions among attendees.

（本文[5]）

Objective:

I am pleased to attend and present my recent work on string abstractions for string verification in the 18th SPIN workshop on model checking of software. This year, SPIN is co-located with the 23rd International Conference on Computer Aided Verification (CAV), which has been recognized as one of the top conferences in Computer Science. The program of CAV consists of 9 workshops, 4 tutorial talks, 3 invited speeches, 35 regular paper presentations and 20 tool short talks and demos. This joint event attracts many researchers in the verification and software engineering fields, altogether with impressive brainstorming.

It has been a great experience to attend SPIN/CAV this year. I can see that in the near future verification of parallel programs and verification of mobile applications would continuously be the focus in the next few years, and has potential to bring significant affects not only in research but also in industry and common life.

My talk on symbolic string analysis:

Verifying the properties of string manipulating programs is a crucial problem in computer security. String operations are used extensively within web applications to

[1] 單位出國案如有 1 案以上，計畫編號請以頂大計畫辦公室核給之單位計畫編號 + 「-XX（單位自編 2 位出國案序號）」型式為之。如僅有 1 案，則以頂大計畫單位編號為之即可。
[2] 執行單位係指頂大計畫單位編號對應之單位。
[3] 出國地點請寫前往之國家之大學、機關組織或會議名稱。
[4] 出國經費指的是實際核銷金額，單位以元計。
[5]頁數不限，但應含「目的」、「過程」、「心得及建議」。

manipulate user input, and their erroneous use is the most common cause of security vulnerabilities in web applications. Unfortunately, verifying string manipulating programs is an undecidable problem in general and any approximate string analysis technique has an inherent tension between efficiency and precision. In this paper we present a set of sound abstractions for strings and string operations that allow for both efficient and precise verification of string manipulating programs. Particularly, we are able to verify properties that involve implicit relations among string variables. We first describe an abstraction called regular abstraction which enables us to perform string analysis using multi-track automata as a symbolic representation. We then introduce two other abstractions---alphabet abstraction and relation abstraction---that can be used in combination to tune the analysis precision and efficiency. We show that these abstractions form an abstraction lattice that generalizes the string analysis techniques studied previously in isolation, such as size analysis or non-relational string analysis. Finally, we empirically evaluate the effectiveness of these abstraction techniques with respect to several benchmarks and an open source application, demonstrating that our techniques can improve the performance without loss of accuracy of the analysis when a suitable abstraction class is selected.

My talk was well received by the audiences. Several questions have been addressed, e.g., how we generate constants for alphabet abstraction (we did this by collecting constants appearing in the corresponding dependency graphs automatically), and what the differences compared to Prof. Dirk Beyer's recent work CPAchecker. CPAchecker is the reimplementation of BLAST which is for analyzing C/C++ programs. We focus on string manipulating programs. An interesting future work is to extend their abstraction framework to string analysis. This can be done by adding string abstractions to their abstraction lattice. I will discuss more details later in summary and suggestions.

Other talks: There are total 3 invited talks, 4 tutorials, 35 regular paper presentations and 15 tool demonstrations in CAV. Below I summarizes some of them.

<u>HAVOC</u>

In the CAV tutorial day, Dr. Shuvendu Lahiri from Microsoft presented SMT based predictable analysis. The tool Havoc is a property checker for C Program. The basic idea is translating C programs to Boogie-Memory models, to verification condition generation, to the inputs of satisifiability SMT solver (Z3). They focus on type safety, particularly support low-level data structure using new efficient SMT solvers. The problem comes from that type checking in c is lack of support for memory safe violation on lists.

The idea of combining SMT: Boolean Satisfiability solving and theory reasoning provides a powerful approach to combine decision procedures for theories [Nelson, Oppen'79], e.g., Z3, Mathset.

One way to deal with memory model for C is treating pointer is an integer, heap as a map and as a result, they have the following:

mem: int -> int // all values are integers,
alloc: int -> {Unallocated, allocated, freed}
base:int-> int //map base address of each pointer-> size from that address

Then they are able to translate C to Boogie (intermediate language, procedure) with simple semantics in integers. For type checking to assertion checking, the run time value corresponds to its compile time type.

Mem: addr-> value
Type: addr->type
HasType: value x type -> bool
forall a in addr, HasType(Mem(a), Type(a))
Here is the formula for modeling the heap:
t* -> HasType(v, Ptr(t)) <-> v==0 || v>0 && Match(v,t))

To deal with aliasing (field safety) refinement of type safety, we add field names and offsets. Briefly, the whole idea is to treat types as additional part of the state.

Regarding the precision of the presented approach, Shuvendu mentioned that the approach is precise to {P} S {Q}, S is a loop free program (Bound program) bounded quantification for interpreted sets. On the other hand, the problem including unsorted logic is undecidable. They enforce sorting restriction on \forall x \in S.F, where sort(x) should be in an less order, so that an incomplete axiomatization of predicate reachability analysis can quantify over lists, array, types.

In sum, synthesizing the quantifier invariants starting from a property is difficult for real code. Havoc presents techniques for new efficient logics on SMT for type checking of lists, arrays, etc. Here are some references: VCC, a functional memory checker for concurrent program, can be downloaded from http://rise4fun.com. For more details about how to deal with Lists: Nelson POPL'83, Strand: Madhusudan et al. POPL11, Trees: Wies et al, CADE11, and Separation Logic: O'Hearn, Reynolds, Yang CSL01.

HAMPI

Vijay Ganesh from CMU presents his recent string constraint solver: HAMPI. His work has been recognized by theACM distinguished paper award. The motivation is that the traditional SMT logics lacks of support for Strings. The key to support strings is how to do approximations, so that it is precise enough to verify interesting propertyes while coarse enough for us to conduct formal analyses.

A satisfiability solver (SAT) consists of the following key techniques: clause learning using conflict analysis, backjumping, variable selection heuristics, restarts. The advanced technique, Set module theory (SMT), provides combinations, under/over approximation of formulas, DPLL(T) solving Boolean formulas, bounding (sandbox) with an effort to combine decision procedures (but not giving ingredients).

The aim of this work is to provide a string solver using SMT, which is motivated by security errors (very difficult to define security errors), incomplete sanity checking.

The frmawork for string analysis is:

String Program, specification -Program reasoning tool-> string formulas--> HAMPI-->SAT, UNSAT

Theories of strings: HAMPI-based vulnerability detection: is there an assignment for the string expression matches the regular expression. You can write as many as attack vectors as you want to finding a string in an intersection of formal regular language. HAMPI takes a satisfying assignment by bounding (Scalability vs. Completeness). The trick is to find an implicit bound by variables:

Hampi->Normalizer, STP Encoder-STP solver-> STP DEcoder

Here are some recent work on string constraints: Hampi, Ardilla, Kudzu, Klee, STP+HAMPI exceed 100+ projects. The secret of HAMPI is eager for Boolean vectors and lazy for Arrays substitution. Another issue is how the bound is automatically derived: length, minimal length, --> enumeration. Also HAMPI encodes regular expression recursively and treats concatenation/kleen star to conjunction.

In the experiments, they deal with 1367 string constraints from Su [PLDI'07], and concolic executions for XSS (the result would be similar to taint analysis), and no replacement examples of Ardilla (Kazen et al. ISSTA'09).

Here are some related work: CFGAnalyzer, REX(MSR), DPRLE (U. Virinia) and Klee: symbolic execution engine with implicit spec, Static analysis: Su, Security testing: Ernst. An interesting project: NoTamper-parameter tamper detection, is based on HAMPI. The idea is to solve S1, S2, .. to C, E1, E2, ... to ~C by calling Hampi, and submit (S1, E1...) to the servers to see how the server responds. If they are the same, then ignore. Otherwise, the server differs from the tamper and mat contain potential vulnerability.

Finally, here are some complexity results: Undecidability of of Quantified word equations, Quine (1946), Undecidability of Quantified Word Equations with single abstraction, Durnev (1996), Decidability of QF theory of word Equations +RE (Makanin, 1977, Plandowski, 1996), Decidability of QF, QF word equation +length (?), QF word equation in solved form+length+RE (G. 2011). It is worth to note that the complexity to solve QF word equations+length constraints is open.

As for the future research direction, they will work on bound for completeness, which appears to be another interesting questions.

<u>Memory Safety</u>

Prof. Ranjit Jhala from University of California, San Diego, presents his recent work on memory safety. The title is using types for software verification. Specifically, he is working on combining several techniques of artificial intelligence, formal methods and counter example guided abstraction refinement to derive sound loop invariants automatically.

This work is motivated by the fact that quantifiers kill SMT solvers, and the key idea is to derive invariants without quantifiers by separating type and logic (so that logic is quantifier free). He gave an example using a famous algorithm: "Map-reduce"

```
map:: (e-> (k,v) list) -> e list -> (k,v) list
group:: (k, v) list -> (k, v list) table
reduce:: (v->v->v) -> (k, v list) table -> (k,v) table
```

Another example is on “K-means clustering.”

0. choose k centers arbitrary
1. (Map) Points to Nearest Center
2. (Group) Points by Center
3. (Reduce) Centroids into New Centers

Repeat till you feel happy (Convergency). The demo is on: demo:http://goto.ucsd.edu/~rjhala/liquid/demo/index2.php

His approach can also extends to collections i.e., structure, as well as genetic types. To make it easy checking software, they tried to avoid quantifiers by mapping factor invariants into liquid types based on SMT+predicate abstractions. In sum, they identify invariants that are independent of states, and use the functional language-ml to simplify the most behaviors of C programs.

After Rajit’s ‘talk, Andre Platzer from CMU presents how to conduct logic and computation verification of hybrid systems. Hybrid system analysis is important for many real time applications, such as flight control. He reduced verification of hybrid systems to image computation problems, and use differential dynamic logics to model semantics. He realized his idea in his tool Keymaera by hacking Key, a theorem prover for Java.

One important result is that dL calculas is a sound and complete axiomation of hybrid systems relative to differential equations. Based on this result, hybrid systems can be verified by recursive decomposition differential equations/parallel computations.

Finally he mentioned how to model stochastic hybrid system using logical foundations of cyber-physical systems.

Summary and Suggestions:

These four tutorials bring the brainstorm among the audiences. Tom Ball and Shaz Qadeer from Microsoft Research, Prof. Moshe Vardi from Rice University, Ed Clark from CMU, Rejeev Alur from U Penn, Kim Larson from Sweden, Alan Hu from University of British Columbia, Daniel Kronning from Oxford University, and many excellent researchers sit together, asked and answered sharp questions. As a junior faculty, I am very lucky to have this chance to attend this joint event. I cannot stop but keep taking notes in every minute.

I have several chances to talk with Rajeeve Alur, asking my recent research problem on Strategy Interaction Logics (joint work with Prof. Farn Wang, NTU). His insight opinion on whether Alternating Temoral Logic is expressive enough for the interaction of strategies directs us the next level of this research. I have also asked for the potential future collaboration and chances to visit his lab next year. His lab in UPenn is recognized as one of the toppest verification lab in the whole world. I believe the experience will be rather helpful both for research and teaching.

Industry talks:

Andy Chou, the cocounder of Coverity Inc, one of the largest static analysis tool company, gave the invited talk: static analysis tools in industry: notes from the front line, to share his past six years experiences on applying research results to real products to solving industry scale program analyses. He starts from static analysis overview, with an aim to do bug-findling and fixing.

Coverity has more than 190 employees. Their tool has analyzed 3-5 billion lines of code, detecting memory leak, deadlocks, race conditions, etc. (Here one problem comes to my mind: how about string defects in web application.) The advantage is the low false alarm rate (typically <20 \% out of the box) of their tools. Several techniques have been implemented including interprocedural analysis with bottom up function summarization. path sensitivity with false path planning, and staged analyses, i.e., cheaper analyses are run before more expensive ones, and parallel, incremental analysis (used to verify android kernal, ~700KLOC).

On the other hand, the things that they don't have: pointer alias analysis, heap structure analysis, and complex string analysis. This is very interesting since all these three topics are the current research focus of programing analysis. I have noticed more than three papers related to these three aspects accepted in CAV. In industry, the major defects are integer overflows, buffer overflows (unknown size to fixed size buffer, using tainted analysis).

Later, Andy shifted the topic to its business model. The big question for a static analysis company is why the clients are willing to provide their source codes for you to analyze. Their business model is as follows: The first stage is trial process: customer build (download trial software, analysis, crypt result). Then the second stage is vetting: results meeting: leave. Then wait, wait until the clients come to see you to explore more vulnerabilities that are encrypted. This is very smart. Their current customers are around 80\% in embedded systems, where people write in C/C++. On the other hand, there is no glory in fixing bugs. You need to motivate your developers and engineers.

At this point, David asked the question why not using binary analysis? The thing is that you may detect bugs but don't know how to fix them in binary analysis. Another question is what the impact is for multi-core changes? Andy answered that it is not really affecting the current structure. For most engineers, their first order is to keep things simple.

Finally, Andy showed some secret statistics that are used in the internal company. One shows that new languages do get adopted in the past years. According to their data, the most languages that are used to develop software in order as follows: Java(-), C(-), C++(-), C#(increase), PHP(down), Objective C(increase significantly), ..., Java Script. This indicates what languages shall we target on. More users, more bugs, and more the need of static analysis.

Regular Paper Presentations:

Code Synthesis

Prof. Doron Peled: presents his work on synthesis of distributed control through knowledge accumulation. The aim is to synthesize a system using formal methods. Their approach consists of the following stages: from a CTL/LTL specification, to synthesis, then to a system. However, it has been shown that synthesize concurrent programs is a undecidable problem. To tackle the problem, they focus on discovering additional specifications to the given system, as automatic generation of controllers to enforce a sound system. This is then the decidable alternative (synchronizing actions as needed by the analysis). Further more, they distribute the analysis by making decisions distributed: local states with overlap. The knowledge is calculated based on the states of original code alternative architecture. They can check knowledge of a process about other processes having enough knowledge. This work will also be published in ATVA11.

Real Time Systems

Real time constraints are everywhere. One can analyze rela time systems using ILP modulo theories (IMT). In CAV, they present the tool Synthia: synthesis of real time systems. They first perform model checking on timed automata, then synthesize the component to satisfy the property, and finally do simplification and optimization to build the final sound system. As we have mentioned in the previous talk, the problem is undecidable in general. We need some means to develop either sound or complete techniques. They adopt abstractions. They convert input models to symbolic product automaton to explicit abstraction automaton, so that they can apply zone based analysis with approximations. If it is too coarse to prove the property, i.e., raising false alarms, they apply abstraction refinement. They show the experiments against CSMA/C for 3 processes.

Model Checking + Theorem Proving

Model checking demonstrates the technique of (symbolically and automatially) exhaustive search on all reachable states and possible behaviors to check the correctness of systems; while theorem proving shows the ability to deduct rigid theories with axioms and proofs to assert system properties, occasionally human interactions involved. It then poses an attract direction to combine both techniques. In this talk, they combine verification: model, implementation with mathematical theory: algorithm, implementation. They put two famous tools together: formal verification VCC with Isabella/HOL. The contribution would be connecting a dirty code verifier to an interactive roof assistant. The analysis stage is:

Semantics C, Memory Model VCC, Assertion Language VCC -> Graph with pure math manipulation -> Second order logic, which maps to the implementation: VCC-> Concrete property->Abstract Property----->Isabella-> Theory

Hybrid Systems

Prof. Sriram Shankarayamana presents how to discretize Hybrid Systems using relation abstractions. Among numerous finite state abstractions, they use relationalization of flows in infinite state transition systems. They use R(x,x') with/without time, connection with positive invariants, and BMC/k-induction for relation. The strongest relation is to collect all pairs of states, from all trajectory:

R \superset {(x,x')| x<x', x~>x'}

The next problem is how to find a good way to compute relational abstractions. They propose positive invariants or ODEs based on the fact that flows starting inside s will remain inside s. (Nagumo's Viability Theorem (1942)). For relation abstraction, the reflexibity: R(x,y) holds. We can derive positive invariance for the associated systems using widening and narrowing techniques. For optimal control approaches, they compute eigen values for affine systems. They also apply loop acceleration, function summarization, transition invariants [Podelski+Rybalchenko]. Their tool is based on Linear Templates+Fixedpoint Solver (Polyhedral Solver using PPL library) + box invariants. They showed the effectiveness of their approach against NAV benchmarks (hard hybrid system).

In sum, they propose relationalization of Ordinary Differential Equations. However, abstraction can be coarse. Their current work consists of refinement: CEGAR approach and trace partitioning, and reduction from liveness to safety.

Timed Automata

Georges Morbe presented fully symbolic model checking for timed automata. UPPALL, a famous existing timed model checker, is semi-symbolic model checking, i.e., explicit location representations. The only fully symbolic model checker for timed automata is RED by Prof. Farn Wang using BDD-like data structures CRDs. I was involved in the RED project a few years ago. It is good to see another symbolic model checker presented in CAV.

They introduce Finite State Machines with Time, and conduct backward reachability analysis on top of them. The advantages of their approach include: symbolic interleaving traversal of discrete states, and parallelism of the components. To symbolically model configurations, they use LinAIGs: And-Inverter Graphs extended with linear in-equations. They use FSMT parallelized interleaving behavior, add boolean inputs to the guards, and allow parallel behaviors. Compared to RED, FSMT outperforms in most standard benchmarks, such as fisher and fddi.

Another Industry Talk:

Formal in Industry

Dr. Vigyan Singhal, Osaki Company, gave a keynote speech on "Deploying Formal in a Simulation World." He starts from the comparison – "Formal in Academia": CTL model checking (linear by system, PSPACE-complete by design), and "Formal in EDA Company": Property synthesis: PSL and SVA. We use to find as many bugs as you did, but we seldom ask the question: when will we find the last bug?

In Silicon Valley, verification is still the largest problem. There are several large verification companies that have been successfully inducing formal techniques to the industry, e.g., Gate-level formal (verification 95\%) Chryslais->Cadence, RTL formal (simulation 90\%). IBM-> Jasper, Mentor.

While using formal tools in industry, it is important to keep that single user should not learn simulation and verification together. The tradeoffs in design flow include schedule, scope, resource. Formal has to be more cost-effective than the alternative to get adopted. It is then important to have an abstraction of design: localization, datapath, memory, sequence, counter, floating pulse. The speaker shows an example: PCIe Transaction Layer. In sum, to be a successful verification company, you need provide a unique methodology, highest coverage, fastest time to market.

More Interesting Talks (that are related to my research):

Parallel Programming

Another hot topic in CAV is parallel programming. Prof. Vikram Adve from UIUC gave a keynote speech on "Parallel programming should be and can be deterministic by default." He starts from LLVM, string analysis, and client parallelism. According to a recent survey on the priorities: HPC (expert parallel programmers) performance, \$5B -> time to market \$100B, he pointe out that multi-threaded programming is hard and in fact, developers tend to develop deterministic parallel programs. A deterministic property is that "fixed input gives fixed output." However, the current parallelism paradigm is non-deterministic, e.g., thread libraries in C, Java, C#, or HPC languages such as OpenMP, It is hence interesting to promote a deterministic mechanism that aims at sequential reasoning, parallel performance model.

There are several benefits: no data races, deadlocks, subtle memory models, easier debugging, using sequential-like tools, easier testing and verification, using sequential tools, incremental parallel tuning, long-term maintenance. Still there are several myths: high run time overhead, incompatible with low-level parallel code, and cannot mix with non-deterministic code. For a parallel program, if you want to test a program, you can test it with CHESS, which ignore data races and ignore compiler non determinism but test as many schedules as possible. On the other hand, for a deterministic parallel program, we can test with standard sequential tool. The non-deterministic behavior was hidden with commutative updates to concurrent data structures so that the final results are still deterministic.

In sum, parallel programming should be deterministic by default. Non-deterministic algorithms should be: explicit, data-race free, strongly atomic, isolated from deterministic ones. They achieve programs that are parallel, statically typed, deterministic by default by using a novel region-based type and effect system, and successfully enforce safe uses of parallel frameworks. In the future direction, they work on mixing deterministic +non deterministic code with an aim to get strongest guarantees.

If we take a look at recent research, we will find that largely focused on arbitrary multi-threaded code framework internals, to work on local properties of parallel framework instead, model checking is needed!

Finally, the tool DPJ today is base on the foundation of deterministic execution guarantee through simple compile-time type checking. It implements data structure alias analysis. An ongoing work is ease of adoption (efficient implementation of atomic blocks).

Malware Analysis

Dr. Domagoj Babic from UC Berkeley presented "Malware analysis with Tree automata inference." It has been dramatically increasingly in recent years malware such as malicious unwanted software for stealing, spying, spamming, etc. including viruses, worms, botnets, trojans, rootkits, spyware, adware.

In this work, they first presented the current solution to detect malware: identification, classification (15-30 min for manual per example), to signature extraction, and then perform signature detection. The syntax based signature detection is brittle and easy to circumvent by code obfuscation. In this work, they shift their focus on how the malware operates, instead of what it does.

According to McAfee Threats report, Malware has six times increased (from 10000 in 2007 to 60000 in 2010) in the past four years. We need new malware samples per day! It also affects the effectiveness of signature based detection: 20\% to 60\% (refine after 7 days). On the other hand, for most applications, souces are not available. In most cases, we need be able to analyze binaries instead, which is easier to be obfuscated, pieces of code encrypted, and often can't even be disassembled!

These challenges come together make static analysis very difficult or impossible. To perform behavior detection, the idea is to identify sequences of executed operating system calls, and get an under approximation of the behaviors. To achieve this goal, they draw system call dependency graphs that traces program executions, log system calls, and track how parameter propagate, and finally compute graphs.

To learn behavior patterns of malware, they propose "tree automata" approach. From the difference of malware graphs and goodware graphs, they generate an automaton that accepts the malware but reject the good one. However, the inference problem of regular tree languages is NP-complete. They use tree languages defined by a set of patterns to scale their approach. They under approximate the beahviors using k-TSS, so that s.length> k, l . s the same as l' . s, to ensure the properties of k-TSS language inductive, positive examples, linear time for inference.

They collect 2631 malware samples in 48 families and show that their approach can increase distinguish rate and classify malwares better. One can download their tool from www.domagoj.info.

My Suggestions (Future Research Direction):

It has been a great experience to attend SPIN/CAV this year. I can see that in the near future verification of parallel programs and verification of mobile applications would continuously be the focus in the next few years, and has potential to bring significant affects not only in research but also in industry and common life. After attending these talks, I was inspired to work on verification of parallel programs. I summarize the idea below. This is my ongoing research project.

Symbolic Consistency Checking of Parallel Programs

Parallel programing is a kind of design to integrate computability of processes and has shown great success in many blazing computing architectures, such as cloud computing and GPGPU (General-Purpose computation on GPU). Although integrating computability of processes can usually enhance total performance, writing correct parallel programs is more difficult than doing so for sequential programs.

One of the challenges comes from the nature of concurrent execution of a parallel program by different threads. Consistency is one fundamental property that a well-developed multi-threaded program shall satisfy, i.e., any of its parallel computation shall produce the same result as its sequential version despite of its execution orders among threads. This property ensures that the result of a parallel computation is deterministic and is consistent with its sequential version.

Open Multi-Processing (OpenMP) is an application programming interface (API) for multi-threaded programming. It is proposed by OpenMP Architecture Review Board in 1997 and now it is supported by many commercial compilers, for example, Sun Studio, Intel Parallel Studio, and Visual C++. It is also supported by GNU Compiler Collection (GCC) since version 4.2. Programmers can parallelize their codes to multi-core systems or superscalar computers by OpenMP. OpenMP provides an easy and incremental way to write parallel programs.

The well-structured OpenMP constructs and well-defined semantics of OpenMP directives make compiler analyses more effective on OpenMP programs than on loosely structured parallel programs that are solely based on runtime libraries, such as MPI and Pthreads. While OpenMP provides a common useful interface and parallel computing has been widely adopted, it is essential to have a formal approach and an automatic tool to check the consistency of multi-threaded programs.

Our work can help programmers to check consistency of their multi-core systems and generate counter examples for inconsistent ones. In this research direction, we will propose a symbolic approach for consistency checking of multi-threaded programs.

Our approach consists of two phases: race detection and symbolic witness search. Races on shared variables are the main cause that corrupts the consistency of parallel computing. When threads can access a shared variable in different orders, the output of the program that depends on the values of these variables may differ in different concurrent executions and make parallel programs inconsistent.

Our race detection is based on constraint solving where we consider parallel conditions, path conditions and race conditions to generate a sound race constraint of a multi-threaded program and solve the constraints (as Presburger formulas). We show that the program is race free (and hence, is consistent) if none of its race constraints is satisfiable.

This consistency is based on the assumption that programming languages should by default guarantee an interleaving-based semantics (sequential consistency) for data-race-free programs. For programs that have any race constraint satisfiable, we generate the corresponding truth assignments to characterize potential races during concurrent executions.

The second phase of our approach is searching inconsistent witness guided by races. This is achieved by symbolic model checking and simulation techniques. To model and analyze a multi-threaded system, we need a modeling language with diverse features, including machine-instruction-level concurrency, message passing among threads, read/write operations on shared variables, and etc.

We can adopt a version of extended finite-state machines with thread synchronizations as our modeling language. We convert programs to symbolic models and perform symbolic simulation on them. With our simulator, we can record the error traces, repeat the traces, backtrack in trace execution, observe intermediate values of variables in trace execution, and even check whether all reachable states have been explored.

We can perform symbolic simulations guided by the result of race constraint solving. The truth assignments of a satisfiable race constraint, represented as a set of values on variables, identify target variables on which races can happen. We symbolically explore execution traces that access target variable(s) in different orders and check whether the races render the output of the program. The search process terminates when (1) a witness has been found, i.e., there are at least two executions yield different outputs, or (2) all reachable states have been explored. In (1), we conclude that the program is inconsistent and generate two sequential versions of the parallel program that yield different outputs as a counter example. In (2), we conclude that the program is consistent (races are benign).

Finally, we would like to realize the ideas and develop the tool {\it Pathg}, an end-to-end automatic tool that checks the consistency of multi-threaded programs written in C with OpenMP directives. Pathg incorporates Omega library to constraint solving and Red symbolic simulator to guided witness search.

建議事項參採情形[6]	出國人建議		單位主管覆核		
	建議採行	建議研議	同意立即採行	納入研議	不採行
1.					
2.					
3.					

[6]出國參加學術會議、發表論文者，此欄位可不必填寫。

出國人簽名：	胡方	日期：	
連絡人：		分機：	77453