Verification of Concurrent Software

B. Tech. Seminar Report

Submitted in partial fulfillment of the requirements
for the degree of

Bachelor of Technology

by

Prathmesh Prabhu
Roll No: 06005002

under the guidance of

Prof. Supratik Chakraborty

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Contents

Abstract

Increasing complexity and widespread use of concurrent programs coupled with the pervasion of software systems handling diverse costly, heavily loaded and safety critical equipment has led to the need for benchmarking multithreaded software systems and verification of their reliability. This seminar surveys some of the recent approaches to practical software verification.

1 Introduction

The importance of software verification was brought to the foreground by a number of instances of failures of costly and safety critical systems due to unforeseen and often trivial seeming bugs in the system software. Quite a few famous disasters that occurred as a result of software malfunctioning may be quoted that justify software verification as a central pursuit in academia as well as industry[**?**].
In November 1985, a 2-byte buffer overflow led to a discrepancy in the Bank of New York securities issue that involved mismanagement of securities worth 20 billion USD. A Russian mars probe was ordered in September 1988 to commit suicide by the ground control when they mistakenly transmitted one wrong character in a message that was in entirety some 20 pages long. A small bug in a switching station software upgrade by the AT&T led in 1990 to a malfunctioning of the entire New York CCS7 network till the upgrade was retracted and the company forced to apologize and compensate its clients. Besides such instances of costly failures, there have even been instances of loss of life as in the case of Therac-25 instruments where programming errors led to death of at least 3 people by overdose of radiation during treatment using these machines. It can thus be seen that there are enough examples justifying the need of robust and more reliable software systems as we ever increasingly rely on autonomous equipment in critical applications. Till date, manufacturers have relied on rigorous testing as the primary approach to produce reliable software, but it is an accepted fact that testing is scarcely good enough to guarantee quality. Over the years, it has been found that programmers on average make 50 errors every 1000 lines of code, and even after thorough testing, finished and well tested software has about 3 errors per 1000 lines lurking in the code on average. Considering the increasing complexity and size of code that today's software consists of, this directly implies a plethora of bugs inherent in the system despite the usual testing measures. Quoting Dijkstra: *"Testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence, never their absence"*. These observations clearly call for a more complete software verification protocol.
A recent development in the field of programming methodology (or programming art!) is the shift towards concurrent programming. Concurrent programming can be succinctly defined as *"the simultaneous execution of multiple interacting computational tasks"*. As the processors' capability curve seems to be flattening out at length, using multiple threads of execution is fast becoming an alternate means of increasing computational efficacy and speed. Besides multithreading, advent of multiple processor machines and distributed computing have added to the allure of concurrent programming. With more and more software employing heavily multithreaded programs, software verification also needs to adapt to the changing needs.
In this seminar, we study some of the more recent approaches towards treating this problem of verifying concurrent programs in perspective of the overall theory of program verification as developed over the last few decades.

Organization of the Report

This report discusses a few recent approaches to this problem of software verification in reference to concurrent programs. These approaches treat the issue of concurrency as the central theme of development while building on top of the traditional sequential program verification techniques, learning from the decades of experience in program verification that was mainly limited to sequential software.

To put things in perspective and to aid the understanding of the underlying concepts of program verification, we start with a review of the theoretical foundations. Section 2 explains the historical treatment of the problem of verification that lead to the conception of a new branch of computer science. It then goes on to treat how these concepts were extended to the world of parallel programs as early as the 70s and touches upon the different schools built around the way of reasoning about concurrent programs.

Section 3 discusses some recent and markedly different approaches to the said problem. A striking commonality of these treatments of the problems is the stress laid on finding practically implementable and efficient solutions to the problem, which stems from the fact that these verification softwares are very much in need today in the industry. We start off by discussing in detail an approach that deals with the problems raised by concurrency by statically type checking programs and declaring them to be type safe. We then move on to other more *testing* like approaches that involve model checking programs by various means. As the state space of a concurrent program is infinite even with finite program length and data, novel approaches to making these programs amenable to model checking are discussed.

Finally Section 4 concludes with a writer's commentary on the methods discussed and the insight gained through the exercise.

2 Putting it into perspective

Before venturing into the different ways in which concurrent programs may be verified, it is important to understand the meaning of program verification and also the theoretical framework on which all the approaches rest.
When we talk about *"verifying a program"* we essentially claim to be able to decisively show that the program has a few well laid down properties. In program verification, these properties are usually given in the form of assertions, or claims that certain of the program variables have the given values, or satisfy a given relationship at some moment in the execution of the program. Hence, one may claim the following: *If a given set of mathematical relationships possibly involving the program variables, holds before the execution of the program, and assuming that the hardware and other variables do not alter the execution of the program, a given set of mathematical relationships may be claimed to be always satisfied after the running of the program.*
Program verification essentially tries to either formally prove or otherwise guarantee such claims about programs under consideration. An important point to note here is the assumption that factors beyond the written code and the inputs given do not affect the execution of programs. This assumption is fundamental in order that program results are reproducible and is inherent in the theory and practice of formal verification. It is of course possible to question this very assumption and hence the purported effectiveness of verification[**?**]. This seminar, however, refrains from the discussion of this issue and moves forward with the assumption that program verification is indeed a topic worth the notice it has. Hoping for the best, we assume that no alpha particle will strike the machine running a software verified by one of the discussed techniques, flipping an important register & leading to utter failure of the whole system!

2.1 Formalization

The above stated idea of verifying assertions was formalized for programs by C.A.R Hoare[**?**][**?**] conceptualizing the field of program verification. Hoare logic is a formal logic whose syntax captures the semantics of a programming language which is then used to infer out *theorems* that are programs annotated with associated assertions. Hoare logic formulae are of the form *PQR* where P is an assertion that must hold before the execution of the program Q and R is an assertion that can then be proved to hold after the execution of Q provided that P was true before the execution. The proof technique relies on breaking down the given program into individual statements such that the overall assertions about program correctness rely on some properties (invariants, as for loops) that are maintained by these smaller pieces of code. Now, if these properties can be proved for the smaller code, the logical framework provides a way to combine these assertions and infer properties about the larger code block from the assertions proved for smaller parts of the program. In this way a proof for the whole program can be built.
Hoare logic was later shown to be applicable by Hoare himself[**?**] and others to generate proofs of parallel programs. As treated by Owicki & Gries[**?**], the formal system could handle parallel programs if augmented with the *cobegin* and *await* syntactic constructs. Programs running in parallel and using synchronization primitives could then be encoded in Hoare logic and proved to be correct by proving the correctness of independent programs (assuming they run independent of others) and then showing the independence of assertions in one of the programs to the execution of other programs. Owicki & Gries called this independence the property of being

interference-free. Besides assertions about specific program variables, the questions of termination of the program and freedom from deadlock can also be treated within this formal system.

2.2 The twin schools of reduction and refinement

As the program size and thread interactions grow, formal proofs of correctness tend to become cumbersome. In order to reason about properties of practical programs, some representational shortcut for program proving is essential. We find here the dichotomy of bottom-up and top-down reasoning in proof techniques.
Doeppner[**?**] suggested the technique of *refinement* to prove large concurrent programs. We begin by proving the correctness of the given program assuming the whole program to be atomic, i.e. we assume that the whole program takes place in one step and hence there is no question of interference due to other programs running concurrently with this program. Having proved the correctness thus, we *refine* our proof by degenerating the earlier assumption of atomic execution into parts of program that are assumed to be atomic while the program comprises of these one step parts. The essential step here is to be able to choose the refinement in such a way as to preserve the earlier proof of correctness or so that the earlier proof can be modified so as to accommodate the possible interleaving of programs due to this refinement - the idea of an expansion being *consistent.* By repeated applications of refinement, we aim to reach a point where the only steps assumed to be atomic are indeed executed as one step instructions by the underlying hardware, and hence our proof is now applicable to the executing program.
A complimentary approach is that of *reduction* as suggested by Lipton[**?**]. Here we begin by treating the given program P and assume certain statements to be uninterruptible in order to prove the desired properties for the program. Let R be such a statement. If it be the case that R is in reality interruptible, i.e. its execution may be interleaved with the execution of some instruction in some other program possibly affecting the effect of R, then our proof of correctness of P need not be correct(as our assumption was ill founded). But if it can be shown that the P has the same properties even if R was actually interruptible as otherwise, then our proof of correctness goes through. Let P/R is the program obtained from P by assuming R to be uninterruptible. Then, we want to prove:

```
P has a property S iff P/R has the same property.
```

All reductions P/R for which this statement holds are called *D-reductions.* Many important properties such as halting are preserved in D-reductions and these reductions are used extensively in the techniques described in this report.

D-reduction

Since D-reductions are used extensively by the techniques discussed below, let us delve a little deeper into the theory. Let programs P and Q be running in parallel and p_1 p_2 p_3 p_4 q_1 q_2 p_5 q_3 . . . be an execution trace where p_1 is the first instruction of P and so on. An instruction p_i of P is a *left mover* if in the execution trace, interchanging it with an instruction q_j preceding p_i in the original trace leaves the required properties of the program execution unchanged. An

instruction is defined to be *right mover* similarly. Now, reducing S$_1$,S$_2$,S$_3$,. . . ,S$_k$ to an uninterruptible form [S$_1$,S$_2$,S$_3$,. . . ,S$_k$] is a D-reduction if S$_1$,S$_2$,S$_3$,. . . ,S$_k$ consist of S$_1$,S$_2$,S$_3$,. . . ,S$_{i-1}$, S$_i$ and S$_{i+1}$,. . . ,S$_k$ such that S$_1$,S$_2$,S$_3$,. . . ,S$_{i-1}$ are right movers and S$_{i+1}$,. . . ,S$_k$ are left movers. Such a reduction preserves the proof of correctness of the given program.
Essentially, if some instructions can be translated to the right without changing the effect on required properties of execution and following them are instructions that can be translated left, then the whole code block can be assumed to be executed together, because to any execution where it is not the case, the above translations can be applied resulting in an execution that has all these instructions together and that preserves the required properties from the original execution. It may be noted that acquiring a lock is an example of a right mover action while releasing a lock is a left mover, a fact that is used extensively in practical proof methods. Equipped with this background into program proving and the extensions necessary using the dual approach of expansion or reduction, we may now venture into practically feasible methodologies to tackle the problem.

3 Practical Approaches to Parallel Program Verification

We have so far discussed theoretical concepts involved in proving parallel programs. It is necessary to note that these techniques are not applied directly for proving large software systems. Coming up with proofs for programs is a fairly involved exercise as can be realized by trying to prove even a slightly non trivial program using the basic methods. For the purpose of proving software systems practically, we need autonomous systems that can reason and/or verify program correctness. We discuss some such approaches shortly. All these methods rely heavily on the theory developed so far but do not directly apply them to get formal proofs.
Different methods to concurrent program proving can be said to loosely confirm, among others, to one of following three approaches.

- The deductive approach follows the theory developed most closely in trying to come up with static pre-runtime proofs of the correctness of programs guarantying their safe execution. The first of the approaches discussed builds a type system that provides compile time guarantee of program correctness and falls into this category.
- Another approach to program verification that resembles "testing" is model checking, where we attempt to exhaustively reason out all possible execution paths of a parallel program, trying out all possible interleavings and ensuring that the provided properties hold for all these executions. For reasons given, an exhaustive search is not possible for parallel programs and different methods handle this problem of state space explosion in different ways.
- Lastly, a third strain of verification proposes a pre-emptive approach called Transactional Memory. This stream of verification techniques relies on providing an abstract memory model which provides ways to define interruption free functions and variables that can be assumed to work correctly even in the presence of other processes. The onus of maintaining the atomicity and other properties is relegated to the implementation of the TM and is handled using well developed methods and insights learned from database transactions either in software (Software Transactional Memory) or hardware (Hardware Transactional Memory). We do not discuss these methods in this report.

3.1 Type safety for concurrency

A typical type system aims at formalizing a given set of constraints on the values that a variable can take and the way that variable is treated in the context of the program. A similar idea can be extended to cover concurrency ,i.e., develop a type system that ensures certain properties relating to concurrency for type safe programs. The properties we treat here are race freedom and atomicity

Race Freedom vs. Atomicity

It is important in our current context to distinguish between a method (function) that is free from races and one that is atomic. To be free of race conditions means that two or more processes executing at the same time may not access any shared variable at the same time. Let A and

B be two processes that share a common variable V. Then, read$_A$(V) and write$_B$(V) occurring at the same time leads to a race. Races are unacceptable because the value as read by A in this case can be a corrupted value as B might be halfway in its write execution when A read the value. Atomicity on the other hand means that any execution of process A can be assumed to have happened in one single step and the actual interleaving of process A with B does not change the effects of the execution of A. It is interesting to note that although atomicity and race freedom often occur together in programs, neither is implied by the other as shown by the example given in the figure below. The program in *fig 1a* is race free but is not atomic as *cur* may become outdated by the time *bal* is updated, leading to incorrect update to balance, while that in *fig 1b* is atomic despite having a possible race condition in the function read.

```
int read(){
 synchronized this {return
 bal;}
}
int add(int val){
 int cur = read();
 synchronized this{
  bal = bal + cur;
 };
}
```

fig 1a.

```
int read(){
 return bal;
}
int add(int val){
 synchronized this{
  int cur = read();
  bal = bal + cur;
 };
}
```

fig 1b.

3.1.1 Type System for Race Freedom[?]

Concurrent access control is handled in Java by the use of locks that must be held before accessing the guarded shared variables. As only one thread at a time can hold a given lock, this ensures that concurrent updates do not occur. A type system that verifies that correct locking is observed does the dual job of giving a way of formally specifying as well as checking locking discipline in programs.

The Annotation:In order to specify what locks must be held at what point in the program, all variables at declaration also declare what locks guard them. This is done using the keyword *gaurded_by*. A variable thus declared must then be accessed in the program only if the required locks are held by the thread. Similarly, a method declares the locks that it requires to be held whenever a call to it is made using the keyword *requires*.

```
field ::= [final]_opt type field_name guarded_by lock = expression
meth ::= type method_name argument* requires lock_set { expressions }
defn ::= class class_name<ghost_var*> body
```

In order to provide flexibility with respect to these annotations, classes can be defined with *ghost variables*. Ghost variables are locks passed as parameters during instantiation. The correct locks are then translated to the gaurded_by and requires clauses. Besides, every class has its own lock that we refer to by *this.* Finally, some classes can be defined to be thread local in order to ease working with sequential parts of the code. We talk about these classes in some detail later.

The Type system: Having annotated the code thus, it can now be checked by type inferencing whether each of the lock constraints declared are indeed followed in the code on field accesses. The core of this system is a set of rules of the type

$$P \; ; \; E \; ; \; ls \vdash e \; : \; t$$

Where P is the program under consideration, E is the environment that provides the types for free variables in e and *ls* is the set of locks currently held in program execution. Then, this rules states that e can be inferred to be a valid expression of type t.
Let us look at an example of a type rule in this system that also highlights an important implementation issue. The rule type checks a reference to a field fd of a class c in the program P.

[exp ref]

$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E \vdash ([final]_{opt}\ t\ fd\ guarded\_by\ l \ = \ e') \in c \\ P; E \vdash [e/this]l \in ls \\ P; E \vdash [e/this]t \end{array}}{P; E;\ ls \vdash e.fd\ ? \ : \ [e/this]t}$$

This rule checks that e is a well typed expression of type c where c is a class define in the program. It then checks that *fd* is a field in class c of type e guarded by l. Next, it checks that the lock l is held at this point in the program. To do this, it replaces all instances e in the expression for l by *this*, to get to a from in which the locks would have been defined inside c and then checks for the membership of l in *ls*. Finally, it checks that t is a well defined type. If all these checks go through, then the type checking for the inferred expression also goes through.
A Type safe program thus guarantees that the proper locks are held at all accesses to variables and no race conditions occur. One last note worth making is the introduction of thread local classes in the type system. These are classes that are not shared by threads and hence the annotational effort can be spared for such classes. So as to maintain the race freedom of programs, a clear distinction is necessary between the shared and thread local classes. Shared classes are classes that have a shared super class and have only sharable fields. A thread local class on the other hand has thread local fields and may inherit from either type of classes. A thread local class instance being treated as an instance of a shared super class can not be downcast to a thread local type in a share method.

3.1.2 Type System for Atomicity

The type system developed by Flannagan and Qadeer[?] for Atomicity builds upon the type system just described. In addition to the guard annotations, we now specify atomicity properties to be satisfied by the procedures in the program and then verify that these atomicity conditions are indeed observed by the implementation. For this, we first develop a framework for atomicity types of program code. Based on the theory of D-reductions by Lipton every sub-program (including individual instructions, procedures and the program itself) can be said to have any of the following atomicities:

- Const: The evaluation of the piece of code does not depend on or change any state.
- Mover: As defined by Lipton, the evaluation is both a left and a right mover.
- Atomic: The whole evaluation can be assumed to be happening atomically.
- Cmpd: The evaluation is a sequence of steps that are disjoint and interleaving with other processes may change the effects of this evaluation.
- Error: The evaluation violates some locking principle of the program. This is the atomicity we are trying to eliminate from our program.

These atomicities follow the sub typing relationship

```
Const <:  Mover <:  Atomic <:  Cmpd <:  Error
```

For example, if a statement is of type *Cosnt*, then it definitely is also *Mover*, since it can be translated in either direction without having any effect on the result of this or other threads. Once the atomicities of individual program statements is known, the atomicities for larger blocks of code can be inferred. For example, a code block with all instructions of atomicity *Mover* is itself a *Mover* (since all instructions may translate left as well as right, and hence the whole code block can translate as well) while a code block of *Atomic* instructions is *Cmpd* (since another process interleaving between two atomic sub-programs can affect the execution of the overall program) etc. The process of thus combining atomicities of consecutive code blocks to get the overall atomicity is called *sequential composition* and is denoted by the operator ;

```
mover ; atomic = atomic etc.
```

Again, the non deterministic choice between executing two statements of choice α_1 and α_2 is

$$\alpha_1 \sqcup \alpha_2$$

called the join of α_1 and α_2

These atomicities are the base types in this type system. To account for conditional statements where either one of the two execution paths may be taken, conditional atomicities have to be introduced.

$$l\ ?\ T_1\ :\ T_2$$

means that if the lock l is then type is T_1 else T_2
Our annotated program now consists of these atomicity tags added to the ones discussed above. Hence, every method must be declared to have one of these atomicities.

The Type Checker

The ideas presented above for a developing a type system extend directly to the added task of inferring atomicity types for the statements and code blocks. The primitive instructions are assigned either of the basic atomicities. The atomicities for the procedures are inferred using the rules of the kind shown before:

[exp while]

$$\frac{\begin{array}{c} P; E \vdash e_1 : int\ \&\ a_1 \\ P; E \vdash e_2 : t\ \&\ a_2 \end{array}}{P; E \vdash while e_1 e_2 : int\ \&\ (a_1; (a_2; a_1)*)}$$

states that while e_1 e_2 can be inferred from the conditions given and if a_1 and a_2 are the atomicities inferred for the statements above then the atomicity of the while loop is given by (a_1; (a_2; a_1)*).
The inference rules developed inculcate both intuition behind the basic atomicity types for primitive statements and the composition of these atomicities in the program.

3.1.3 Type Inference

In the approach discussed above, the onus of annotating a program in order to be able to argue about its type safety lies largely on the programmer. An intriguing next step in developing the suggested type system would be to be able to infer consistent atomicity types for the program procedures given an unannotated / partially annotated program[**?**].
We approach this problem by extracting a system of constraints from the program that should be satisfied by any assignments to the unknown atomicities. We then solve this system of constraints, i.e., we find an assignment of atomicities to the unknown methods such that all the constraints are satisfied, proposing a simple algorithm with this aim.
To first come up with the necessary constraints, we extend the type system developed so far a little further. To accept methods with unknown atomicities, we introduce *atomicity variables* that are placeholders for atomicity types inferred from the analysis. Our new type inference

system now consists of rules of the form:

$$P; E \vdash e : t \; d \; \overline{C}$$

where $\overline{C}$ is the set of constraints inferred in the type inference for the statement. For example, let

$$\frac{\begin{matrix} A \\ B \end{matrix}}{D}$$

i.e. expression D be inferred from the expressions A and B. and if this inference requires the atomicity type α_A of A to be subtype of the atomicity of B α_B, then D can be inferred with the constraint $\alpha_A \sqsubset \alpha_B$. The constraints thus obtained accumulate during the constraint generation phase. Once the whole program has been type checked, the generated constraints are solved to get a satisfying assignment.
Due to the form in which the constraints are generated and propagated in the type rules, all obtained constraints are of the form $\alpha_A \sqsubset$ d, where α_A is an expression containing atomicity variables and d is a *closed* (known) atomicity. A *satisfying assignment* is one that satisfies all these subtyping constraints in the constraint set $\overline{C}$. To find a satisfying assignment, we start by assigning the lowest possible assignment of atomicity *Const* to all variables and then expand the assignments incrementally by the least possible amount till we reach a fix point.

The methods discussed above have been implemented for the full Java language and tested on non trivial programs. The results were encouraging with many known and unknown violations detections in the tested Java libraries[**?**][**?**].

3.2 Model Checking

The second set of methods we discuss are based on model checking the given program. Model checking is traditionally done by capturing the relevant properties from the given program in the form of an abstract model drawn from the program and then exhaustively reasoning about the required properties on this model.

Typical model checkers for sequential programs exploit the fact that primitive data types in most languages are of finite range. This coupled with finite execution paths of the program text means that there are only finitely many states that the program can be in (where a state is defined as a particular combination of the program execution and the current data). This reduces model checking to an exhaustive check over a finite (albeit large) state space. The moment we introduce recursive function definitions, as found in most common languages today, the state space becomes unbounded. In parallel programs, programs do not have a bound on the number of threads that may be running at a given moment, and any possible combination of states for each of these many threads forms one state of the whole system, exploding the state space further. Even with a finite bound on the number of thread spawned at any given time, the interleaving of these threads significantly increases the complexity of model checking.

The first two approaches discussed below reduce the problem of modeling concurrent programs to that of modeling a derived sequential program along with some extra work to ensure that this modeling is consistent with the original concurrent program. The last approach actually model concurrent programs laying stress on ways to optimize search and cover the maximum state space possible.

3.2.1 Method View Consistency

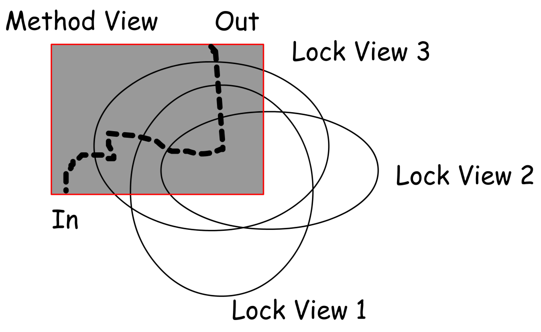
The first approach we consider is a model checking algorithm that runs on an abstract model derived from the compile time internal representation of the Java program. To tackle the concurrency issue, we define method consistency and check that our model has this particular property.

The following model checking algorithm developed by Praun & Gross[**?**] is intended to verify a given property for Object Oriented Programs written in Java keeping in view the issues raised by multiple threads running in parallel. The algorithm discussed here is *neither sound nor complete*, i.e., it is possible that it misses out some instances of concurrency violations while it is also possible that some of the alarms raised are false alarms.
The algorithm runs on HSG: The abstract model we choose for our algorithm is drawn from the *Heap Shape Graph (HSG)* that the Java memory model creates during compile time. The HSG has static nodes corresponding to each of the classes defined in the program and during runtime, any instantiation of a class is carried out by copying out an instance from the model in HSG. For our analysis, we work directly on the HSG and assume that any two instances created for a class are one and the same and two threads working on data structures in these two instances actually interfere. Thus, our analysis is on the conservative side assuming that maximum possible interference takes place. It is found that this conservative approach does not really increase the number of false alarms significantly since most threads that work on shared data do work on the same class instance in typical programs.
Method Consistency: The modeling of concurrent access to methods is handled in this approach through the idea of method consistency. First a couple of definitions are called for.

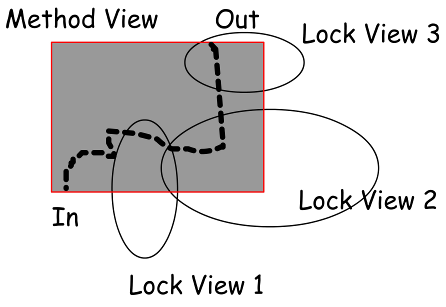
Lock View is a set of <variable, access> pairs that model the variables accessed and the type of access by a lock t at runtime. Access may be read(r) of update(u). The set of lock views of thread t is specified as $L_t = l_1, l_2, l_3, \ldots l_k$.
Method View is a set of <variable, access> pairs that model the variables accessed during the method call at runtime. Here both read and write access are added to the set and the set of Method views of a thread t is $M_t = m_1, m_2, m_3, \ldots m_k$.

Two views are said to overlap if their intersection is non empty, i.e. some variable is accessed in the same way in these two views. A set of V of views form a chain with respect to a view v' if their intersections with v' form a collection of increasing sets, i.e. :

$$u \in V \wedge v \in V \longrightarrow u \cap V \subset v \cap V \vee v \cap V \subset u \cap V$$



(a) Method Consistent: intersections form a chain



(b) Not Consistent: An execution can go through several disjoint locks

Figure 1: Method Consistency: The square is a Method View and the Ovals are the intersecting Lock views. A chain formed ensures that stale values are not re-written. The dotted line represents a method call and the different values accessed.

A program is said to be method consistent if, for all the methods in the program, the overlapping lock views form a chain.
What this means is that given any method of the program, for any variable access within the method, the locks held must be nested. This idea is made clear by the picture above. Method consistency correctly captures a very common reason for violation of atomicity where a variable is read in a method under one lock and then written back under another lock. The possibility of a *stale* value of the variable being written back leads to inconsistency. Method consistency ensures that such stale values will be caught during model checking.
Model Checking: As described above, the model checker does an exhaustive search on the HSG model of the Java program. During this search, the model checker assumes the program to be sequential and then aims to guarantee atomicity by checking for method consistency in relevant methods. To begin with, methods from different classes can not interfere because in OOP the only data members visible are those within the classes. Even within a class, methods that are not synchronized are pruned from the search. The sets of relevant methods in each class are then checked for method consistency by evaluating the view overlap of all locks with all methods and checking for the chain property.
Thus, this model checker aims at verifying the given program through an exhaustive model search. To contain the state space explosion seen with concurrent programs, the model checker verifies the program assuming that it is the only program running (that is, running in sequential mode). The model checker then guarantees that errors due to concurrency do not arise by checking relevant methods to be method consistent. This is a simple and effective approach to model checking, but it is only an approximation to verification and can not guarantee good results.

3.2.2 KISS - Keep It Simple and Sequential[?]

This approach with the vogue name works by reducing the given concurrent program to a sequential one that models in some way the runtime behavior of the original program. This new

sequential program can then be model checked by any existing model checker (here SLAM) for errors that capture errors arising from concurrency issues as well as other sequential program errors. It is important to note that the sequential program obtained does not model all possible executions of the original concurrent program but only a small subset of the executions. It is hoped that any problems arising from concurrency issues will manifest themselves in the restricted form of concurrency modeled by this approach.
Program transformation: We model the execution of the concurrent program by introducing a non deterministic scheduler that may do one of the following at any point in the execution of the original program:

1. Start a new thread evoked by the concurrent program asynchronously in the past.
2. Terminate an existing thread.

To implement this non deterministic scheduler, instrumentation code is added after every line of code in the original program. The new program obtained has a single stack (sequential program). This single stack stores the context of all the thread that are currently "active", i.e. have been started by the scheduler some time in the past but have not terminated yet, in contiguous blocks. If the "scheduler" decides to terminate the currently running process, it sets a local variable "raise" to true and returns at that point. The variable raise leads to an immediate return by all functions down the call stack of the current process, there by causing an "immediate" return by the process. The process below the current process in the call stack then continues its execution from the point where the terminated process was started.
In order to schedule a new process and in order to do this activity in a bounded fashion, we introduce a global array *ts* that keeps count of all scheduled processes. Whenever the concurrent program evokes a function asynchronously, the function is added to *ts* if *ts* is not already full. The scheduler may now decide to evoke this function from *ts* any time in the future non deterministically, meaning that another context is created on top of the stack and the new call stack resides here. If *ts* happens to be full, then the asynchronous call is converted to a synchronous call and a new thread is created immediately.

The scheduler and race detection code: When we say that the scheduler decides to start a scheduled thread, we mean that the a part of the instrumentation code added in between the lines of the original program makes this choice. This scheduler is implemented on an abstract level by a function *schedule()* that is called after every line. It goes through the *ts* array and non deterministically decides to run any of the threads. Code displaying such non deterministic behavior can be efficiently checked by sequential model checkers like SLAM. Also, instrumentation code contains a choice between RAISE and NULL where the former leads to immediate termination of the running thread while the later does nothing. Finally, this instrumentation code must also contain code that checks that all the required assertions at a point are met and that no race condition exists. A possible way to check race condition is using functions like:

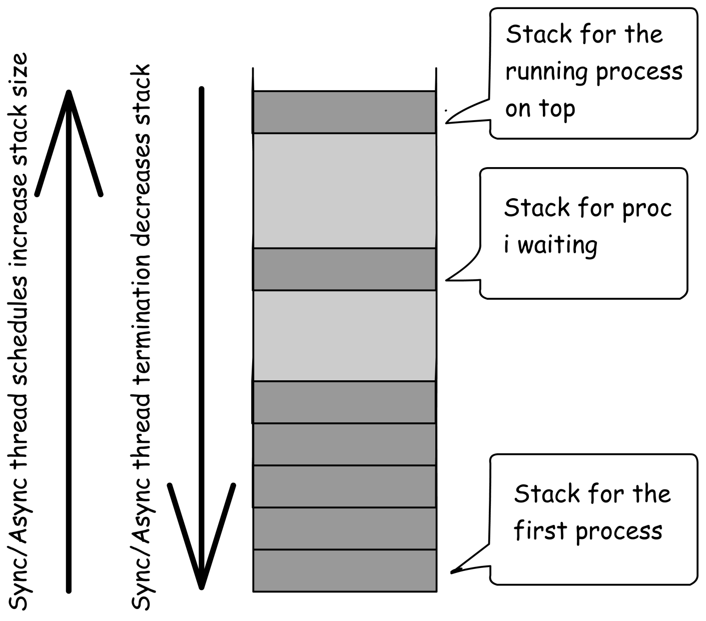

Sync/Async thread schedules increase stack size
Sync/Async thread termination decreases stack
Stack for the running process on top
Stack for proc i waiting
Stack for the first process

Figure 2: A pictorial view of the singel stack program. Different 'active processes' reside in contiguous blocks on the stack

check$_r$(x){
 if (x == &r){
  assert (¬(written(r));
  read(r);
 }
}
	check$_w$(x){
	 if (x == &r){
	  assert (¬read(r) && ¬written(r));
	  written(r);
	 }
	}

These functions check that beyond aliasing, no accessed variable has been written/read since the last step execution of the current process.
Finally, it is instructive to argue about the state space covered by this approach. Clearly, it does not exhaustively trying out all possible execution paths of the concurrent program. We note that the only source of concurrency modeling in the resulting program are through the synchronous or asynchronous calls executed by the scheduler code and the end of program terminations or RAISE terminations. Hence, any thread that starts executing once goes into the stack and many more threads may then execute preemptively before the termination of this thread, but once this thread terminates, it never restarts. This is precisely the types of interleaving that this procedure models: A stack based interleaving where every new thread is added to a stack and popped out on normal or preemptive termination. Any number of threads may be stacked on top of this thread during this time, giving rise to heavy concurrency even in this restricted domain. Also note that the length of the array *ts* determines the number of possible outstanding threads at any time, hence acting as a knob on the extent of concurrency exhibited by the model.

3.2.3 Iterative Context Bounding

We now move on to actually stepping through a model that captures all possible execution paths of a concurrent program. The approach developed by Musuvathi and Qadeer[**?**] simulates the

execution of the program itself, without abstracting away from the source code while following a systematic protocol to be able to guarantee the correctness of program to some measurable metric. In order to tackle the problem of infinite state space in model checking, the idea of *depth-bounding* is often used. Depth-bound model checkers guarantee the correctness of systems for the first m number of steps, m being the depth of the search. This idea was developed for and is very effective for modeling message queues where any protocol violation that may happen is likely to occur within the first few messages exchanged as long as all possible orderings of this exchange are captured. The same idea when extended to concurrent program verification, though, gives unsatisfactory results because programming errors may be situated anywhere in the program code and exhaustively verifying the first few lines of code does not guarantee program correctness on the average.
The complimentary idea that applies well here is that of *context bounding.* Context bound is an upper limit on the number of context switches that occur in the execution of the program. We define a context switch as a preemptive replacement of a running thread by some other thread (it does not include the change of context when a thread yields or blocks on a call). Then, a context bounded search tries out all possible executions of a program with no more than m context switches where m is the search depth. Iterative Context Bounded search starts with a bound of zero, i.e. a program execution without any context switches and verifies that program is correct when assumed to run independently. Then, it iteratively increases the bound, exhausting out all possibilities of the current bound before moving to a higher one. This process should ideally go on till infinity to cover all possible execution paths but practically only a small number of context switches are modeled and required to cover significant part of the state space.
Let us pay attention to a few important attributes of the search explained above. First, with a context bound of zero, the modeling guarantees the correctness of program running independently. Even with this trivial bound, the results obtained are meaningful!. Secondly, Context Bounded search gives us a very intuitive metric on the extent of program correctness assured at the end of modeling. A search up to bound m means that no errors occur up to a maximum of m context switches during the execution of program, irrespective of how and where these switches occur, since the state space for each bound is covered exhaustively. If it can be shown that modeling software execution with a bounded number of context switches faithfully simulates the actual execution of programs, then this modeling can prove to be a very efficient way of verifying correctness under parallelism, as is found to be the case. It was found that most common bugs in concurrent software manifest themselves in as few as 3 context switches, and if up to 8 context switches are allowed then almost all bugs can be found. Of course this search is not exhaustive and not all bugs can be detected by this approach as the number of switches has to be kept small to contain the state space. The number of states visited can be shown to equal

$$^{nk}C_c.(nb+c)!$$

where n is the number of threads, k is the length of program code for each thread, b is the number of non preemptive context switches and c is the number of preemptions. n and b are typically small and as long as c is kept small this bound is a polynomial bound on the length of code. Hence, keeping the number of preemptions small reduces the normally exponential bound to a polynomial bound on the number of states.

An optimization in the search is possible by limiting the points at which the said context switch occurs. Initially, we assumed that a context switch can occur at any point in the program but it can be shown that it is enough to schedule context switches preceding synchronization operations in the program provided that race detection is done otherwise through other methods. Specifically, it can be shown that,
A race free terminating execution of a program α is equivelant to another race free terminating execution β where all context switches occur before synchronised actions with no more context switches than the original execution α.
In view of this result, the total number of states that need to be visited decreases further giving us a fairly efficient approach to modeling large concurrent systems.
As the approach described above relies heavily on the claim that a small number of context switches effectively model real program concurrency problems, it is mandatory to validate these results with actual implementation to program verification. The initial results stated in [**?**] are encouraging. An algorithm developed on this philosophy was tested with a number of concurrent software systems and it was found to detect a number of known and previously unknown bugs within a small context bound of 2 or 3. Also, it was found that almost all of the state space is covered (in terms of the number of distinct states visited) with a bound of 8 to 11. In initial testing, this method was found to work remarkably better then other contemporary methods of *depth bound search* and *depth first search.*
The algorithm discussed above does seem to cover large tracts of the state space effectively. It is imperative though for the reader to keep in mind that it merely provides a systematic approach to program testing. Like other model checking methods it does not guarantee program correctness but provides us with a metric that objectively ensures correct execution of programs in a large number of cases and weeds out most common bugs effectively.

4 Conclusion

Programming Language Principles is a field of computer science where one sees theory and practical implementation come close. We notice a similar situation in the course of this seminar. Beginning with the motivation behind program verification that for some stems entirely from the lure of finding structures in the different programming languages we use, and their effect on the correctness or effectiveness of the exam while for others is a simply he need of the day, to the approaches that have developed to tackle this problem, we find a series of magnificent insights from theory being brought out to a practically implementable level.
We see that the earliest work in the field was largly theoretical in nature. This earlier work helped in clearly defining the framework on which the subject builds and exploring the implications and limitations of the newly conceptualized ideas. This early work provided a robust base for the subject to build on.
It was soon realized that the formal concepts need to be diluted in order to build meaningful verification systems. The methods of Owick & Gries[**?**] never got uesd to prove any non trivial software systems. This formal methods were found to be too cumbersome for direct application. The first strain of the approaches we discussed simulated the original ideas the closest. The way these ideas were assimilated to come up with a type system to statically type check concurrent programs is amazing. Formalizing all necessary notions for *concurrent safe* programs, such type systems have opened a new window for system verification softwares, and also for a new programming paradigm that pays due heed to ensuring well written programs in parallel environments. It provides a means not only to verify programs but to switch to programming techniques that reduce inherent errors.
The other fork of verification software we saw are the more *usable* and *current* verification approaches, that systemize testing by modelling and extend the most intuitive approach to program safety a step further. Because these modelling software closely model the way most programmers are used to testing their programs, they go down well in the programming community. By providing a plethora of well implemented software verifiers, these model checking programs have catered to the current need of the industry. It is nice to note that all these approaches proived very intuitive and effective methods to control state space explosion, being simple (and hence!) efficient at the same time.
Finally, it is apt to note that the field of concurrent software verification is still in its youth. A lot of active research is going on and a lot of work is being done in the two fields discussed here as well as transactional memory. One hopes that along with tools to verify programs that make software systems safer and more reliable, the insights gained will help POPL to come up with language models tailored to the prallel environment and also the programmers to inculcate better coding habits leading to cleaner code : an utopian goal of a lot of CS research!

Acknowledgements

I would like to thank my guide, Prof. Supratik Chakraborty for introducing me to this wonderful topic and for his guidance therein. The experience has been enthralling.

References

[1] Peter H. Roosen-Runge. Why do we need software verification tools?

[2] Anthony Hall. Software verification and software engineering - a practitioner's perspective. *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE.*

[3] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), October 1969.

[4] C.A.R. Hoare. Proof of a program: Find. *Communications of the ACM*, 14(1), January 1971.

[5] C.A.R. Hoare. Parallel programming: An axiomatic approach. *Computer Languages*, 1:151–160, 1975.

[6] Susan Owicki and David Gries. An axiomatic approach for parallel programs i. *Acta Informatica*, 6:319–340, 1975.

[7] Thomas W Doeppner. Parallel program correctness through refinement. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 155–169, 1977.

[8] Richar J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18, 1975.

[9] Cormac Flannagan and Stephen N. Freund. Type based race detection for java. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, 2000.

[10] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 338–349, 2003.

[11] Stephen N. Freund Cormac Flannagan and Marina Lifshin. Type inference for atomicity. *Proceedings of the TLDI'05: 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 47–58, 2005.

[12] Christoph Von Praun and Thomas Gross. Static detection of atomicity violations in object oriented programs. *Journal of Object Technology*, 3:103–122, 2004.

[13] Shaz Qadeer and Dinghao Wu. Keep it simple and sequential. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 14–24, 2004.

[14] Madan Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 446–455, 2007.