Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Static Detection of Anomalies in Transactional Memory Programs

Bruno C. Teixeira — 26043

Lisboa
(2010)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Static Detection of Anomalies in Transactional Memory Programs

Bruno C. Teixeira — 26043

Orientador: Prof. Doutor João M. S. Lourenço

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do Grau de Mestre em Engenharia Informática.

Lisboa
(2010)

Para a avó Maria,
para o avô Joaquim,
e para a avó Lucília

Acknowledgements

"If I have seen a little further, it is by standing on the shoulders of Giants."

— Sir Isaac Newton

In spite of all the days and nights spent on developing this work, I would have not been able to produce it without the help of numerous people.

I definitely owe much to my advisor, João Lourenço, for always being supportive, and for always being able to pragmatically put the most complex problems in the most simple perspectives, which so much helped me in times of despair. His disregard for his own sleeping needs has many times enabled crucial advances in my research, and shows his dedication to science and to his students.

A big *thank you* goes out to Diogo Sousa, who provided much help in developing the tests for the approaches presented in this thesis. Even though he's an undergrad student at this time, I am by far outmatched by his tremendous technical skills, and he has a bright future ahead of him. Ricardo Dias, fellow student, also provided important help and companionship.

Eitan Farchi, from the IBM Haifa Research Lab played an irreplaceable role in the development of this thesis. The knowledge he shared and the hints he provided may have decided the direction in which this work was carried out. He also provided me with tests for my approaches be enabling access to the IBM concurrency testing repository.

I would definitely like to thank the partial support by Sun Microsystems and Sun Microsystems Portugal under the *Sun Worldwide Marketing Loaner Agreement #11497*, by the Centro de Informática e Tecnologias da Informação (CITI), and by the Fundação para a Ciência e Tecnologia (FCT/MCTES) in the Byzantium research project PTDC/EIA/74325/2006.

And, of course, because there is more to life than work (even though it didn't seem that way in last few months), I would like to thank a number of other people for helping preserve my sanity, and for sometimes righteously putting me on the wrong track.

I would like to thank Joana Pinto, without whose caring attentions this thesis would have been written in half the time, even though I would not know what to do with that time if I didn't have her in my life.

I would like to thank all my wonderfully cool family for all their support and motivation. I specially want to thank my parents, Armando and Alexandra, for being (incredibly) able to raise such a stubborn and incomprehensible creature as their son into such a reasonably apt

human being. And of course, my sister Sofia, whose naiveness (which always ends up being less than I perceive) always reminds me of myself and of where I came from.

And last, but definitely not the least, I would like to thank them, the guys, the buddies, my friends, who countless times, often with the company of a beer (though most times it was a cup of coffee), reminded me what the point of life is, and of what being a student is all about. This thesis is the culmination of a long path, and therefore I would like thank all the friends I have met that have helped shape my character, and getting where I did. In order to conceive this section in finite time, and out of respect to trees, I shall enumerate just a few, such as Danilo Manmohanlal, Pedro Bernardo, and all the other guys who also wrote their thesis at the same time as me and pulled an all-nighter right by my side, Maria Café who was also always there for me, João Not Gomes, Sofia Gomes, João Cita Martins, João Tok de Sousa, Luís Açoriano Nunes, Tiago Amorim, Arlindo Lima, Simão Mata, Hugo Aguiar, Emanuel Couto, and all the guys from my old school class.

Summary

Transactional Memory (TM) is an approach to concurrent programming based on the transactional semantics borrowed from database systems. In this paradigm, a transaction is a sequence of actions that may execute in a single logical instant, as though it was the only one being executed at that moment. Unlike concurrent systems based in locks, TM does not enforce that a single thread is performing the guarded operations. Instead, like in database systems, transactions execute concurrently, and the effects of a transaction are undone in case of a conflict, as though it never happened. The advantages of TM are an easier and less error-prone programming model, and a potential increase in scalability and performance.

In spite of these advantages, TM is still a young and immature technology, and has still to become an established programming model. It still lacks the paraphernalia of tools and standards which we have come to expect from a widely used programming paradigm. Testing and analysis techniques and algorithms for TM programs are also just starting to be addressed by the scientific community, making this a leading research work is many of these aspects.

This work is aimed at statically identifying possible runtime anomalies in TM programs. We addressed both low-level dataraces in TM programs, as well as high-level anomalies resulting from incorrect splitting of transactions.

We have defined and implemented an approach to detect low-level dataraces in TM programs by converting all the memory transactions into monitor protected critical regions, synchronized on a newly generated global lock. To validate the approach, we have applied our tool to a set of tests, adapted from the literature, that contain well documented errors.

We have also defined and implemented a new approach to static detection of high-level concurrency anomalies in TM programs. This new approach works by conservatively tracing transactions, and matching the interference between each consecutive pair of transactions against a set of defined anomaly patterns. Once again, the approach was validated with well documented tests adapted from the literature.

Keywords: Transactional Memory, Concurrent Programming, Concurrency Anomalies, Program Testing, Program Validation, Static Analysis, Program Transformation.

Sumário

A Memória Transaccional (MT) é uma abordagem à programação concorrente baseada na semântica transaccional importada dos sistemas de bases de dados. Neste paradigma, uma transacção é uma sequência de acções que podem ser executadas num instante lógico, como se fosse a única a ser executada nesse momento. Em contraste com sistemas de concorrência baseados em *locks*, a MT não requer que apenas um único fio de execução esteja a executar as operações sincronizadas. Em vez disso, tal como em sistemas de bases de dados, as transacções executam em paralelo, e os efeitos de uma transacção são anulados em caso de conflito, como se esta nunca tivesse tido lugar. A MT é um modelo de programação mais fácil e com menos tendência para erros, e oferece uma potencial melhoria na escalabilidade e no desempenho.

Apesar destas vantagens, a MT continua a ser uma tecnologia ainda jovem e imatura, e tem ainda de se afirmar como um modelo de programação estabelecido. A MT carece ainda da parafernália de ferramentas e padrões que se veio a esperar de um paradigma de programação generalizado. Técnicas e algoritmos para teste e análise de programas que utilizem MT também estão apenas agora a merecer a atenção da comunidade científica, fazendo deste um trabalho inovador em muitos destes aspectos.

A ambição deste trabalho prende-se com a detecção estática de possíveis anomalias em programas com MT, que possam surgir em tempo de execução. Irão ser abordados *dataraces* de baixo nível, bem como anomalias de alto nível resultantes de quebra incorrecta de transacções.

Foi definida e implementada uma abordagem para detectar *dataraces* de baixo nível em programas com MT através da conversão de todas as transacções de memória para regiões críticas protegidas por monitores, sincronizadas num novo *lock* global gerado. Para validar esta abordagem, a ferramenta foi aplicada a uma série de testes, adaptados da literatura, que contêm erros bem documentados.

Foi também definida e implementada uma nova abordagem para a detecção estática de anomalias de concorrência de alto nível em programas com MT. Esta nova abordagem funciona através da listagem pessimista de transacções, e da verificação da interferência entre cada par de transacções consecutivas em relação a padrões de anomalias pré-definidos. Uma vez mais, esta abordagem foi validada através de testes bem documentados adaptados da literatura.

Palavras-chave: Memória Transaccional, Programação Concorrente, Anomalias de Concorrência, Teste de Programas, Validação de Programas, Análise Estática, Transformação de Programas.

Contents

List of Figures

Listings

1

Introduction

1.1 Context

Concurrent programming has been an essential programming model in the last years. Even though the main paradigm in use has been that of sequential programming, concurrent programming has proved its advantages, and helped us better develop many programs, even on sequential processors.

However, its importance is about to be raised to a new level. As the trend in processors shifts from increasing the processing power of single-chip processors, to the increase in number of cores per processor, software development must adapt to this new environment. Future advances in technology will mean increased parallelism, instead of increased clock speed. Software development must therefore take advantage of this offered parallelism in order to evolve.

But parallel programming is inherently more difficult than sequential programming. The fact that program steps are not executed in a predetermined order makes it very difficult or impossible to consider all possible thread interleavings, and thus assure that a program is correct. The main resulting problem are the so called *dataraces*, which occur when two threads simultaneously access the same shared data, leading to incorrect program behavior. In order to avoid these conflicts between threads, critical code regions must be declared. The standard approach used for synchronization nowadays is the use of *lock*-based mechanisms. Locks are items that must be acquired by a thread before it can proceed into a guarded code block. Two threads may not simultaneously execute code guarded by the same locks, i.e., if at least one lock is simultaneously guarding both code blocks. Another recurring problem is the *deadlock*, which occurs when two or more threads are simultaneously trying to acquire a lock that another one has already acquired. In practice, this means that they both are waiting for the other one to finish, and so neither of them ever does.

Transactional Memory [HM93] is a fairly recent approach to concurrent programming that presents a new programming model. It borrows the abstraction of *transaction* from database systems to provide for better abstraction and composability. Its goal is to provide a better programming model for concurrent applications, while potentially increasing performance of parallel programs on multi-processor systems. In a transactional memory system, critical accesses to shared memory are coordinated through transactions. A transaction is a sequence of instructions and memory accesses that must appear logically atomic. Through transactions, a programmer can state that a particular code block must be executed as though a single thread was running on the system.

In contrary to memory accesses guarded by locks, a transaction can be committed, aborted, rolled-back and retried. A transaction's effects are not immediate. Instead, after a transaction is done with its operations, it tries to commit. If it succeeds, the effects are made permanent and are visible by other threads. If a conflict is detected with another transaction, then one of them shall be aborted, and its effects rolled-back. The aborted transaction may be retried later, or give up. One desirable property of this execution model is that it allows real concurrent execution of mutual exclusive blocks. This optimistic synchronization allows two or more cores to execute blocks that access the same variables, instead of having all except one of them lock and wait for the active one to end. Transactional Memory was first proposed by using hardware support. However, approaches have been proposed that implement TM through software, without need for modifications on current hardware.

Although TM has promising advantages over lock-based synchronization, it is still possible to write an incorrect program [FQ03a]. Many problems are mitigated with the use of TM, but some are still present, in varied, often more subtle, forms. Because TM is such a young technology, a lot remains to be studied about the evolution of these problems into their transactional forms, and the seriousness they may still present.

1.2 Problem Statement

Transactional Memory (TM) is an emerging technology, and as such it still lacks the maturity and standards of other paradigms. In particular, there is a shortage of tools and mechanisms to analyze programs and detect errors, in the same amount of those available for non-concurrent programming, or even for other concurrent programming approaches.

Many problems that are present in legacy concurrent programs are still observed with TM. Such is true with dataraces. Although there has been a lot of study around datarace detection, few of these works consider the transactional scenario. There are not either many studies that address the possible application of established datarace detection approaches on programs with transactions.

We now present two classes of anomalies which may be observed in transactional programs. *Low-level dataraces* are the equivalent in TM to the classical dataraces observed in lock-based programs, and which have been the subject of intense study. *High-level anomalies* are more subtle issues, which may be present even in the absence of low-level dataraces.

Low-Level Dataraces

As further detailed in Section 2.1.4, even when using TM there is the risk that the code inside a transaction will be interleaved with non-transactional code. A system that allows this kind of interaction is said to provide *weak isolation* [BLM05, LR06]. In this scenario, we have the same pragmatic guarantees for thread interaction that we would have if not using synchronization at all. Therefore, it is easy to conceive a scenario where two threads have undesired concurrent accesses. An example is presented in Listing 1.1. This small program has been written in a dialect of Java, that has been extended with the `atomic` keyword. This language extension is quiet common in the literature because it provides a transactional environment which is succinct and easy to understand, and is actually used in many systems [HF03, ATLM+06]. The `atomic` keyword defines a code block with transactional semantics. As such, all instructions in this block are to be executed completely, or not executed at all, and without allowing any other transactions to see the effects of this transaction until it has successfully finished. In

Listing 1.1: Example of a datarace in weak isolation.

```
1  public class Example extends Thread {
2      private static float salary = 0.00; // shared state
3      public void run() {
4          // ...
5              atomic {
6                  float tmp = salary;
7                  tmp = tmp + 42.00;
8                  salary = tmp;
9              }
10         // ...
11     }
12
13     public static void main(String args[]) {
14         Thread t1 = new Example();
15         t1.start();
16         // ...
17             float tmp = salary;  // read shared state
18             tmp += 42.00;  // tmp might not reflect salary anymore
19             salary = tmp;  // parallel updates might be lost
20         // ...
21     }
22 }
```

this program, two threads will be increasing a shared value. The spawned thread, referenced by variable `t1`, runs in a transaction, and is therefore protected from interleavings with other transactions. However, because the critical code in the main thread is not transactional, it is possible that these threads will be scheduled to run in parallel. One of them may be preempted just after it has read the value from the counter. In this case, the other thread may then have the chance to read the salary, increment its value, and store it, without being interrupted. When the first thread resumes, it will then work on an outdated value. The result is that the counter will be update only one time, when two updates should have been observed.

High-Level Anomalies

We will now assume that the underlying TM system provides *strong isolation* [BLM05, LR06]. This means that code inside a transaction is guaranteed to run in full isolation, even from non-transactional code. In practice, this means that every operation outside an explicit transaction is automatically pushed inside a transaction of its own.

We will be reusing the previous example of a shared salary value. Because strong isolation is assumed this time, we have encapsulated all the pertinent instructions inside transactions, for illustrative purposes. This adaptation is shown in Listing 1.2.

Listing 1.2: Example of an anomaly in strong isolation.

```
1  public class Example extends Thread {
2      private static float salary = 0.00; // shared state
3      public void run() {
4          // ...
5              atomic {
6                  float tmp = salary;
7                  tmp = tmp + 42.00;
8                  salary = tmp;
9              }
10         // ...
11     }
12
13     public static void main(String args[]) {
14         Thread t1 = new Example();
15         t1.start();
16         // ...
17             float tmp;
18             atomic { tmp = salary; }  // read shared state
19             tmp += 42.00;  // tmp might not reflect salary anymore
20             atomic { salary = tmp; }  // parallel updates might be lost
21         // ...
22     }
23 }
```

We are assured by the TM system that the `atomic` block can assume that none of the operations in the `main` method will execute between two consecutive instructions of its own. However, the opposite is not true. Each single instruction in the `main` method is encapsulated inside a transaction, but the sequence of instructions as a whole is not. In this specific case, the programmer probably wanted to explicitly define a transaction containing the code pertinent to the counter increment.

1.3 Thesis Claim

We claim that it is possible to detect runtime anomalies in programs which make use of Transactional Memory, with reasonable precision, using a static approach. We furthermore claim that the detection of low-level dataraces in TM programs may be achieved by conveniently adapting available approaches for datatarace detection in lock-based programs. Our final claim is

that the detection of most high-level anomalies may be reduced to the evaluation of consecutive pairs of transactions only, as opposed to verify the whole set of transactions of a thread.

1.4 Approach

We have listed a series of issues that may possibly be observed in TM programs. As was said, many of these issues are derived from, or related to, similar issues that are present in either sequential, object-oriented, or non-transactional concurrent programs. Although TM is a recent technology, there is considerable study around detecting and correcting these issues on lock-based programs. Therefore, our approach will consist in assessing the possibility of applying these established techniques to the transactional scenario.

This work will perform anomaly detection through static analysis of Java source code.

The anomalies addressed by our work are divided into two major classes, and accordingly the approach will be divided into two major parts. Due to the tight relation between low-level dataraces in TM and in typical synchronization mechanisms, the detection of these anomalies in TM programs will be carried out by porting these solutions to our context.

The detection of high-level anomalies will also take in consideration previous work which addresses lock-based concurrent programs. However, rather than merely porting these approaches for usage in TM programs, we present a refined approach, which rivals with presently available solutions. We will refine previously established anomaly patterns, in order to produce more satisfiable results.

1.5 Contributions

As happens with all new development technologies, the use of TM will not be generalized immediately, and must be gradually integrated in the current software development context. The maturation and establishment of a new development technology is limited by the understanding and confidence of the community in this new technology. We consider that automatic solutions for the problems presented here are facilities that would accelerate the assessment of the community as to how profitable TM can be for software development in this age of multi-cores.

We expect to contribute to increase the efficiency of TM software development, aiding current TM programmers with tools that will help assure a correct and desirable program behavior. We hope as well to provide a more appealing development environment, enabling the software development community to start using this new promising paradigm, and understand and use the advantages that TM offers.

1.6 Document Layout

Chapter 1 has introduced the transactional programming model, the problems it raises and how they may potentially be solved. We have taken a look at a few exemplary issues in con-

current and transactional programming, and how they may affect the quality of applications and of their development.

In Chapter 2 we present the scientific context related with the work described in this dissertation. We will address areas of research which must be discussed, since they will be used in defining the core work of this thesis.

Chapter 3 presents the detection of low-level dataraces in TM programs, its implementation resorting to a well known datarace detector for Java programs, and its validation by applying our detector to a set of well-known buggy programs from the literature. These programs were adapted from classical synchronization mechanisms to TM. The detection of high-level anomalies will in turn be addressed in Chapter 4, where we will show how we implemented a static conservative inference of the sequences of transactions that may be executed at runtime, and check for anomaly patterns between each pair of consecutive transactions.

After the presentation of our work, Chapter 5 summarizes it and presents the conclusions. Possible future work will also be presented.

2

Related Work

2.1 Transactional Memory

While parallelism has been hard to manage in general programming, database systems have for long been required to cope with multiple simultaneous queries, and to efficiently exploit concurrency. Users of a DBMS typically describe their operations without having to worry with this underlying concurrency: each user sees the system as if his transaction was the only one running on the system. The system takes these concurrent requests and tries to serialize them. If a conflict is detected, the DBMS takes appropriate action. It is therefore the natural attempt to try to apply these techniques to general programming.

Transactional Memory (TM) [HM93] is a new approach to manage concurrent access to data shared among different computational processes, that may be implemented either through software (Software Transactional Memory – STM) [ST95] or hardware (HTM) [HM93]. Like concurrent access management of database systems, TM rests on the essential concept of a *transaction*. A transaction is a sequence of actions that must be executed atomically and isolated from other transactions. While it is executing, a transaction makes tentative changes to data. After the changes have been made, the transaction attempts to *commit*, i.e., make the changes permanent. Because multiple transactions are potentially being executed concurrently, there is a chance that a conflict will occur resulting from parallel accesses to the same data, so action must be taken to avoid an inconsistent state on the data set. The system will *abort* one or more of these conflicting transactions, resulting in the aborted transactions' effects to be *rolled-back*. The transactions that were not aborted will have their effects successfully committed, and the aborted ones may be retried later, or dropped.

2.1.1 Properties of Transactions

A transaction should satisfy the following properties, commonly and collectively known as the *ACID* properties:

ATOMICITY A transaction's effects must be visible in its totality, or not at all. If any of the effects implied in a transaction can not be observed, then the transaction should abort completely. A failed transaction must not leave evidence of its execution.

CONSISTENCY A transaction takes the system from a consistent state to another consistent state. A transaction is required to leave the system in consistency, because transactions will assume they start executing in a consistent state.

ISOLATION A transaction executes with the perception that no other transactions are executing. Intermediate states of transactions must not be visible to each other, and a transaction must execute as if it was not interleaved with others.

DURABILITY Database systems are concerned with the additional property of durability. A transaction's effects must be permanent and durable over time, even in the event of a system crash. Transactional Memory generally cares only of accesses to transient memory, and so this property is generally dropped, characterizing TM transactions as satisfying the *ACI* properties.

The goal of TM is to provide a better programming model for concurrent applications, while potentially increasing performance of parallel programs on multi-processor systems. Lock-based synchronization methods are conservative and pessimistic, potentially making threads block in order to avoid conflicts that would rarely occur. This is even more undesirable with multiprocessors, where interacting threads may be running on different processors, because we do not take full advantage of available parallelism and computational power. Locks are also a hard-to-manage mechanism, and as such are error-prone and lead to confusing code. For each lock in a program, the programmer must keep track of its name, its purpose, and which threads may hold it at an arbitrary moment. This turns the simple task of declaring a critical section into a very error-prone process.

TM creates an abstraction of the underlying synchronization primitives, and lets the user state *what* code should be synchronized, instead of *how*. Transactions are also easier to compose with each other than locks are. If a transaction is defined inside another transaction, the system should still be able to figure out how to manage the interaction with other concurrent accesses. If a guarded code block is moved inside another guarded one, one must be careful in considering the implications of holding multiple locks in the inner block, having to review a larger part of the code than the one that is being modified. Another advantage of TM is that optimistic concurrency control is possible. While not all systems implement this feature, a TM system may allow multiple threads to access data concurrently, as most databases do. The system then analyzes the sequence of interactions between threads, and decides whether a conflict has occurred. If both threads finish their transactions without conflicting, a better overall utilization

of the system resources has been achieved achieved, compared to having one of the threads sleep and wait for a lock to be available.

2.1.2 Differences Between Memory and Databases

Although TM finds its roots on databases, the main memory of a computer system is a very different environment, and therefore techniques can not be immediately ported from one environment to the other. The following are some differences between database systems and transactional memory, that require special attention when adapting existing techniques:

1. Database systems typically manage data on disks, while we are managing data on main memory. Access latency for a hard disk, or other type of permanent storage, is various orders of magnitude higher than that of memory access. Any delay caused by synchronization or management overhead in TM will result in a much higher overhead than it would in a database system. In addition, an access to disk can be delegated to a dedicated I/O system. An access to main memory inhibits the CPU from performing any other task;

2. In contrary to DBMS's, transactional memory should not, in principle, have to worry about the *durability* property. Operations are made on volatile memory whose state need not be preserved after execution. This allows the simplification of a TM system implementation;

3. The interaction with DB systems is typically made through simple systems, using high-level languages. TM faces the challenge of penetrating a very wide environment, built on complex languages, large existing programs, established programing methodologies and libraries. TM must coexist with current trends and implementations. If TM is to replace present concurrency mechanisms, a transition must be supported.

4. Requests to a database system are only concerned with reading and writing of data, which are easy to reverse and remake. However, in a general programming environment there are many operations, such as I/O and network communications, that can not be undone or replayed so straight-forwardly.

2.1.3 HTM vs STM

There are many proposals to implement TM both as specific hardware with support for transactions, or by software, through the use of special libraries or language constructs. Both of them have different implications, and different weaknesses and strengths, which we discuss in this section.

HTM systems have better performance, as they are composed of specialized hardware that implies no overhead. They are independent of operating system, programming language and compiler technology. This means that, on one hand, programs can take advantage of TM more transparently, as program design is not entangled with transactions implementation. On the other hand, hardware components are blind to the semantics of program implementation, and

are therefore unable to perform possible optimizations. For example, STM systems can easily provide object-granularity (discussed in the following section), whereas HTMs only allow word granularity, observing memory as a regular sequence of equal memory units. On the contrary, implementing word or byte granularity in STMs is feasible, but at a cost of increased complexity and overhead. HTM systems are bounded by the fixed size of registers and caches, imposing limitations, e.g., on the maximum number of instructions on a transaction. Finally, STM can be implemented on existing systems without the need for modifications. It can be modified or tuned more easily than HTM, without commitment to a specific implementation. In conclusion, HTM offers better performance and transparency, while STM offers more flexibility.

This subject would not be fully discussed without a word about *Hybrid* TM systems, which are a combination of both hardware and software mechanisms. The motivation for these approaches arises naturally from the desire to combine the advantages of both HTM and STM. Some proposals simply take an existing HTM and STM systems and glue them together [Lie04, KCH+06]. Their typical mode of operation is to execute transactions in the HTM system, and in case of failure, gracefully and transparently delegate them to the software component. The main philosophy behind these systems is to take advantage of hardware for full performance in the common case, and use software to get around the problem of fixed size structures, only when required. Studies such as the ones present in [CCM+06, AAK+06] indicate that most transactions are compatible with the size of fixed hardware structures. A different approach is the one presented in [SMD+06], a hardware-accelerated STM system. This system provides an interface for the STM system to have fine control over hardware transactional mechanisms, namely which cache lines the transactional system should control. Hybrid systems are indeed a promising approach, and may yet come to have a similar impact as hybrid virtual memory mechanisms have today.

2.1.4 Design Considerations

As transactional memory is still an emerging technology, many aspects remain without consent on which are the best decisions. We present some problems that arise when designing TM systems. Some of them are open issues that can be resolved, but lack deeper study by the community, or need time for TM to mature in order to be able to understand which are the best decisions. Others have more than one way of being solved, each with their own advantages, creating a kind of taxonomy for TM systems.

Granularity

Granularity is classified according to the smallest piece of memory that can be protected from conflicts. The two major classifications are *object*-based granularity and *block*-based granularity. Object granularity is concerned with the consistent state of an object, and detects conflicting accesses to the same object, or the same field of an object. Block granularity, or alternatively, word granularity, is concerned with conflicts that arise on a contiguous sequence of bytes or

memory units.

Direct/Deferred Update

Because changes made by transactions will potentially be rolled-back, it is not unnatural to delay these writings until the system can be sure they do not conflict with other transactions. This is called *deferred update*.

Deferred update can be achieved by modifying a local copy of an object, private to the transaction. This copy later replaces the target object, or is used to modify it atomically.

But one of the intentions of TM is to allow real concurrent access to data, based on the assumption that most transactions successfully end without conflicts. Thus, many systems have started to implement *direct update*, executing the tentative changes directly on the target object. This is more suitable to an optimistic approach.

Optimistic/Pessimistic Concurrency Control

Some TM systems implement a pessimistic concurrency control, much in the same way that lock-based mechanisms do. They try to deal with conflicts before they occur, before a conflict is even confirmed. If two threads try to access the same memory location or object, the system assumes a conflict and takes action. On the other hand, optimistic systems allow these transactions to change the object concurrently and leave the detection for later, if the conflict occurs. Pessimistic systems typically resort to some form of locking, and as such will incur in some overhead. Optimistic approaches eliminate this overhead, although they may be problematic in scenarios where many conflicts arise.

Conflict Detection Timing

Associated with the type of concurrency control is the time at which conflicts are detected. Systems are generally split into *early* conflict detection, and *late* conflict detection systems. Early detection is made before a transaction starts, or during its execution. Late detection is when a conflict is detected immediately before the data is committed. Invariably, conflicts are detected through *validation*, in which a transaction checks that the data that it will read and write (its read- and write- sets, respectively) is still consistent.

A transaction can be statically analyzed at compile time for variables that it will access. So a conflict can be detected even before a transaction starts if, for example, it will read a variable that is already marked as being written by another transaction.

Contention Policy

When a conflict is detected between two concurrent accesses to shared data, the TM system must take appropriate action to restore consistency. This is achieved by aborting one of the conflicting transactions, or optionally, if possible, by delaying it. The set of rules that decide which transaction from the two is to be aborted is called a *contention policy*.

There is a number of contention policies available. These policies can take into consideration the order in which the transactions started, for how long they have been running, the number of times they have been involved in a conflict, their size, and the number of objects they have opened. Naturally, other policies are mixtures of two or more of these, and many contention managers implement more than one policy. Through STM it is possible to define dynamic policies, changing the current policy according to the scenario, or letting the programmer define the policy to use.

Transaction Nesting

One of the strengths of TM is its ability to compose much better than guarded code blocks. Including a transaction inside another one, or calling a transactional routine from inside a transaction is straight-forward, and should be easily managed by the TM system. However, the semantics of this transaction nesting should be discussed. There are three established models of behavior for nested transactions: *flattened*, *closed*, and *open*.

If nesting is flattened, then the enclosing and nested transactions behave as though they were a single one. The effects of the inner transaction are not visible by other transactions (except the enclosing one) until the outer transaction has committed. If the inner transaction is aborted, the outer transaction will abort as well. Flattened nesting is conceptually simple and easy to implement. However, because only one transaction is actually in execution, it is not very useful. It is also not a very good semantic for programmers. Imagine defining a transaction that manipulates data, then calls a method `m`, then commits. If `m` contains another transaction, and if that transaction aborts, then our original, enclosing transaction will be aborted as well. This might not be desirable, specially considering that we may not even be aware of the transaction inside `m`.

Closed nested transactions also do not have their effects visible until the outer transaction commits, although this outer transaction can see its effects. However, if the inner transaction is aborted, the enclosing transaction will remain active. If the inner transaction is given up, execution will resume at the body of the outer transaction.

An open transaction has an opposite behavior. When a nested open transaction commits, its effects are immediately visible to all other transactions, even ones that are not surrounding it. Furthermore, even if an enclosing transaction later aborts, the nested transaction's effects will still persist.

While the use of open transactions is handy for making permanent changes that can not be rolled-back, it should be observed that this may go against the established constraints imposed by the ACI properties. An aborted transaction which encloses an open transaction could leave evidence of its execution. Once again, the option of implementing more than one type of nesting, or letting the user decide between them, is not discarded.

Exceptions Handling

The system's behavior must be defined in handling exceptions that are thrown from within transactions. If such an exception is not caught inside the transaction (and thus leaves its scope) it would be natural to make the transaction abort, and pass control to the closest exception handler. There is another alternative: the transaction could be terminated, committing its changes so far, but without proceeding, and treating the exception as in the previous case. In common exception-supporting languages such as Java, different exception classes could be created for each scenario, making the exception handling more flexible, and passing to the programmer the power and responsibility of deciding the behavior to adopt in the case of an exception.

Isolation Level

So far we have talked about interactions between different transactions, and assumed the property of isolation between them. However, the basic methodology for programming with transactions is to annotate the beginning and end of a transaction, encapsulating a small part of the code. So it is possible that an anomaly will be caused by interaction between transactional accesses, and accesses not protected by a transaction.

In database systems, all operations are defined inside of a transaction. Whenever a commit operation is requested, the system handles this, and immediately starts another transaction. This is called the transactions-all-the-time model. In this model, a program is a sequence of transactions, and not instructions. In this model, it is not possible for a data access to be outside of a transaction, and so we may confine ourselves to study the interactions between transactions. However, most TM systems do not follow this model, and are designed for sporadic transactions which must be explicitly declared.

If the code outside of a transaction is allowed to interfere with code inside of a transaction, then we say that the system offers *weak isolation*. If a transaction is guaranteed to be executed in isolation, even from code that is not on a transaction, then we say that there is *strong isolation*, which is equivalent to automatically encapsulate all accesses to shared data inside a transaction [LR06].

Unrevertible Operations

The behavior of transactions and the interaction between concurrent transactions are well defined for data manipulation. If a transaction commits, its changes to memory are made permanent; if it aborts, any updates that it performed are reverted to the previous values; if the transaction is retried, these changes could be made again, possibly succeeding or being reverted again.

However, in general programming one must also take into account operations that are not only concerned with data manipulation. Consider a transaction that writes a value to memory and then sends a confirmation to another system, through a network channel. If the transaction is aborted, we are unable to change the fact that the communication was already sent. Even if we can buffer disk writes and delay other types of I/O, and other case-specific solutions, there

are scenarios where this is not possible. Suppose a transaction that sends a prompt to a user and then receives input. If the prompt is buffered for the eventuality that it must be reverted, the user will not see the prompt, and thus will not provide the input, resulting in a technical deadlock.

There is no definite solution to this problem. Some systems drastically disallow such operations inside transactions, or allow only one single non-abortable transaction to perform I/O. Others relax their guarantees and confine themselves to manage only memory accesses. This may be a good solution if combined with a mechanism to automatically invoke specific handling code when a transaction commits or aborts, such as the one presented in [DLC08].

2.1.5 Discussion

Transactional Memory defines the context for this work. Even when we discuss areas that are not directly related to TM, or not specific to TM, we will attempt to define the relation to TM and emphasize possible applications in TM. Throughout this thesis we will address issues related to TM systems such as the ones described in this section. The work presented here will as generic as possible, but may address specific types of TM systems. We will clarify any assumptions made about the type of TM system addressed.

2.2 Program Analysis

Program analysis is the automatic analysis of the behavior of a program. It enables the verification or discovery of certain properties of programs. Since our goal is to detect anomalies in programs, program analysis is naturally related to this work. Throughout this section we will explore relevant features of program analysis, with emphasis on static analysis and its possible application to TM programs.

We will first establish the difference between static and dynamic analysis, in Section 2.2.1. We then survey two different types of static analyses, dataflow and control-flow analyses, in Section 2.2.2. Control Flow Graphs and Abstract Syntax Trees are two program representations used in static program analysis, and with particular relevance for this thesis. They are discussed in Section 2.2.3. Finally, we consider two prominent approaches to static analysis. Type systems are discussed in Section 2.2.4, and symbolic execution is explored in Section 2.2.5.

2.2.1 Static vs Dynamic Program Analysis

Static analysis works by analyzing the *specification* of a program. While most static analyses take the program in its source-code form, many take as input an already compiled program in binary form. Static analysis is made before program execution. Therefore, as an example, it may be capable of finding errors in programs before they occur. A static program analysis considers all possible inputs and program execution traces, including ones that may never occur. So a property or error may be reported even if it may never have observable results in practice, originating false positives. Static analysis works by applying specific algorithms to

input programs. Most times these algorithms actually work on graphs, and for this there are some well-known program-representing graphs that are used often, such as the Control Flow Graph (CFG) [All70, dCLSL01].

Dynamic analysis, on the other hand, targets the *behavior* of the subject program. This kind of analysis is based on a specific execution of the program, and is performed while the program is running. This means an added overhead, but a much simpler implementation, and results that are the reflection of real program executions. In order for the results of dynamic analysis to be valid, the input variety and test size must be large enough, and reflect real scenarios. Dynamic analysis works by logging a series of events at runtime, relevant to the analysis in question, such as memory accesses to a specific memory area. These events are evaluated either during or after the program execution. The relationships between these events, as well as the timely ordering between them might also be important for the analysis. The events may be generated by a modified version of the target code, sometimes adding instructions at specific places surrounding the event-triggering instructions, in a process called *instrumentation*. Another approach is to run the program on a controlled environment that observes the behavior in a way transparent to the program, such as some form of virtual machine.

The approach taken by this thesis will be based on static analyses, and throughout this document we will give special emphasis to it.

2.2.2 Control-flow and Data-flow Analysis

Data-flow analysis is a static program analysis that aims at determining the possible values of memory locations in each program step. Language compilers generally perform some sort of data-flow analysis, in order to make data-related optimizations, such as recycling memory locations mapped to variables that will no longer be used.

Control-flow analysis, on the other hand, is concerned with establishing flow relationships between the *basic blocks* [All70] of a program, i.e., where can the control of the program go from a specific point, or where may it have come from. Control flow is the order in which the operations (instruction, functional calls, and control flow statements) on a program are executed. Control flow analysis is used by compilers to perform a number of optimizations, such as the elimination of unreachable code.

In the remainder of this section we discuss other categorizations of static analysis related to control-flow and data-flow analyses.

Types of Data-flow Analyses

Data-flow analysis is actually a very broad class of analyses, with many specifications. Some of the more relevant for this thesis are now presented.

Reaching Definitions Determines which variable assignments may reach a given point in the program, i.e., which assignments were made to the variable that were not overwritten until the current point in program.

Liveness Analysis Determines which variables are live at a given program step. A variable is live if it will not be written before it is read again, i.e., it is live if it does not contain a value that will be needed in the future. Not to be confused with reaching definitions.

Pointer Analysis Determines the possible variable, or memory location, that a pointer points to. It may be also called *points-to* analysis.

Alias Analysis This is a specific case of pointer analysis. Its goal is to determine if two pointers point to the same location, in which case they are said to be aliased.

Escape Analysis Another specific case of pointer analysis. Escape analysis is concerned with the escaping of memory references from subroutines or threads. This happens when, for example, a variable or object is declared inside a subroutine, and then that routine returns a pointer to that memory location. We say the pointer has *escaped* because we can no longer determine where in the program the pointer will be used.

Happens-in-Parallel A static analysis which is specific to concurrent programs. Happens-in-parallel analysis is concerned with defining which statements in a program are executed concurrently by different threads. The most common variant is the may-happen-in-parallel analysis, which is commonly used in datarace detection approaches.

2.2.3 Program Representation for Static Analysis

Any static analysis must be carried out by first interpreting the program. This input program is transformed into a structure which is more suitable for analysis than either the source-code or the binary form. We present two such structures which will be of special importance in our work.

Control Flow Graph (CFG)

The CFG is the basis for a number of analysis procedures, such as the one in [dCLSL01]. The CFG summarizes all the possible executions of a program by showing the possible traversal sequences between its statements.

In a CFG, a *basic block* is a sequence of instructions without any flow alteration such as branching or loops. In this kind of graph, the nodes represent statements or basic blocks, and the edges are the possible control transfers that may happen in the program.

Control-flow analysis is used to discover both the basic blocks and the edges of a program, thus providing an analysis of the *behavior* of the program. Data-flow analysis may be used in order to analyze the *values* of the program, by determining the possible states of each node.

While in simple sequential languages the flow graph can be derived almost directly, complexity grows when dealing with certain language facilities, such as exception raising. This is why a CFG is more complex and powerful than a simple flow diagram.

Abstract Syntax Tree (AST)

Although more often used in program translation or compilation, an Abstract Syntax Tree (AST) [Jon03] is also a frequent starting point for static program analysis. An AST is another graph representation of a program, which is concerned with source code structure. Unlike the CFG, which details the behavior of a program, the AST shows the relation between different statements in a program. Since it is a syntactic representation, and a direct mapping to points in the source code, it is suitable for easy program transformations. Even though it is not as adequate for program analysis, since it does not feature context information, it is often the starting point for building more complex representations, such as a CFG.

2.2.4 Type and Effect Systems

Type systems allow to statically assure that certain operations are only performed on data that corresponds to a required type [FQ03b]. The types of values may be either declared or inferred during the analysis. They are implemented in most common-use languages through annotations that define the data type of each value to use. A type-checker observes the program before compilation to assure that certain runtime errors derived from incorrect interaction between data types do not occur, rejecting programs that are not correct according to the type system. A type system defines the available types, the type resulting from an interaction between two specific types, the allowed combinations, and as such, the possible type-errors. A type system works by trying to apply a type to a program.

Effect systems specify the effects that result from an operation [FQ03b]. Effect systems are typically extensions of type systems, in which case they are collectively called a type-and-effect system. These specify not only what kind of values there are in the program, but also what happens to those values, maintaining the same idea of mechanically applying rules to check whether the program is correct with respect to the constraints of the system.

Type-and-effect systems may be used to statically check a program in respect to almost any correctness criterion, including those related to concurrency. We will see in Section 2.4 how some solutions use type-and-effect systems to guarantee the absence of dataraces and deadlocks.

2.2.5 Symbolic Execution

One major drawback in dynamic program analysis is the difficulty in providing guarantees for all possible program inputs. Symbolic execution [Kin76] is a static analysis technique that allows us to have some knowledge of variable values in each point in the program. Instead of assigning literal values to variables, the flow of a program is followed by providing inferred *symbolic* values, which represent classes of values of the variables. A simple example is the inference of the value of an integer variable. Certain mathematical properties may be used to eliminate certain values. If two positive integers are multiplied, then the result is known to be positive as well. This guarantee may be used to determine control flow of certain sections in the program.

2.2.6 Discussion

Program analysis is a very wide subject, with many ramifications that are out of the scope of this work. We have presented a few general concepts, some of which are of particular interest to us, and will be referred later in this dissertation. The concepts introduced in the following sections and in the remainder of this document will be defined by using the definitions presented in this section. In Section 2.3 we will see how some tools are used to provide program analysis for varied purposes, and in Section 2.4 we will see in which ways datarace detection tools make use of program analysis. Later, in Chapters 3 and 4, many concepts presented here will also be invoked in order to explain the approaches taken in detecting anomalies. Although the subjects we have addressed are not directly related to TM, they may be possibly applied to transactional programs.

2.3 Program Transformation and Analysis Tools

Our goal is to scan TM programs for spots that may cause an anomaly. For this, we must build tools that parse a subject program and find pieces of code that match a set of rules. In this section we consider systems that attempt to simplify the process of defining analyses and transformations of programs.

These transformation systems vary in many aspects. While some of them are confined to a single specific language (or a small set of languages), others have a language-definition component, typically through the usage of a language-definition-language. The approach for defining transformations may be through the use of a transformation language, or imperative manipulation of an abstract program representation. Particularly, each system generally falls into one of two categories; we shall call them *rewriting systems* and *transformation frameworks*. These are very different types of systems, whose differences are relevant to the scope of this thesis. However, because they have coincident goals, we present and compare them together.

On the rest of this section we analyze this difference closer, as well as other forms of categorization. We then take a look at some of these solutions, list their features, and make a comparison between them.

2.3.1 Rewriting Engines vs Compiler Frameworks

We present two typical characterizations of transformations systems. These are mere guidelines, in the sense that a system that fits on one of these classifications does not necessarily present all of the related attributes.

Rewriting systems follow a declarative approach. They are easier and quicker to use, and can be used for any language. This is because their functioning is based on transformation languages, often divided into a *language-definition language* and a *substitution language*. A language-definition language defines a grammar, enumerating terminal symbols and tokens, much in the way of Backus-Naur Form (BNF). Substitution languages define the output for each matched token, optionally according to free symbols that are to be matched with attributes of the matched

token. This is, however, very limited compared with the context-dependent transformation available with program analysis tools presented next. Many rewriting systems work by examples, i.e., "find all `x=x+1` and replace them with `x++`". Examples of such systems include TXL [Cor06], ASF+SDF Meta-environment [vHKO02], and Stratego/XT [Vis04].

Compiler Frameworks are more complex and powerful. Instead of making simple text transformations on input code, they manipulate some form of abstract representation of the target program, such as an Abstract Syntax Tree (AST). This representation is typically just referred to as the Intermediate Representation (IR). They are composed of front-, back-, and mid-ends. The front-end parses the input and generates an instance of the IR, which is passed to the mid-end. The mid-end then manipulates the IR and applies all required transformations on the program structure. This modified IR is then passed on to the back-end, which unparses it, i.e., generates the output code from it. This architecture around IRs is very good to handle programs written in a specific language, but makes it hard to adapt the system to a new one. These frameworks make a more semantical interpretation of the whole input program (which may span more than one file), and so this logic is spread throughout the framework. Although some provide facilities that allow them to be extended to new languages, this extension is not trivial and can not be accomplished without modifying at least one component of the architecture. In contrast to the declarative environment used in rewrite systems, compiler frameworks take a more imperative approach. They provide libraries for representing and manipulating IR instances, often implemented in the same language they manipulate. Another difference from rewriters is that, because the whole program is handled on a static representation, a lot more context information from any part of the program can be used in the transformation or analysis. Note that representing a large part of the program in IR, or all of it, makes compiler frameworks a lot heavier than simple rewrite engines. Examples of such frameworks include ROSE [SQ03], LLVM [LA04], DMS-SRT [DMS04] and Polyglot [NCM03].

In conclusion, rewriting is very simple and quick to use, with light systems. They are great for quickly testing language extensions and direct translation between languages.

Compiler frameworks are less flexible, consisting of more complex systems. They do provide additional power by supplying the user with specific and extensive API. for IR manipulation. This method can be a lot heavier: it is not cheap to handle an IR representing a program with millions of lines, compared with simple pattern detection and substitution. However, it does allow more complex program optimizations and analyses.

2.3.2 Examples of Analysis and Transformation Tools

We now present some of the most well-known systems, and try to understand how they relate.

ROSE The ROSE Compiler Infrastructure [SQ03] was developed at the Lawrence Livermore National Laboratory by Daniel Quinlan. It is still under development but seems to be stable. Because it is open-sourced and supports addition of new front-ends and back-ends, it could support additional languages, but out-of-the-box it is only designed for use with C, C++ and Fortran. ROSE allows attribute evaluation and IR manipulation through

a specific C++ API. It supports out-of-the-box graphical representation of programs, and high-level analyses such as control-flow and data-flow.

LLVM The Low Level Virtual Machine [LA04] (LLVM), is a compiler infrastructure designed for program optimization throughout the whole lifecycle of applications, i.e., at compile-time, execution time, installation time, and offline between executions. It was first specified on the Master Thesis of Chris Lattner [Lat02]. Parts of the project are still not fully (or not at all) implemented, such as profile-adaptable optimizations. It does have an extensive documentation. LLVM seems more flexible than ROSE when it comes to adding new language support, since it works on a specific low-level assembly-like language, performing all analyses and transformations on this representation. It provides an extensive API.for IR manipulation, including classes for pointer- and aliasing-analyses.

DMS-SRT The Design Maintenance System – Software Reengineering Toolkit [DMS04] from Semantic Designs Inc. is a commercial set of tools for performing analysis and transformation tasks on industrial-scale applications. It has been developed since the decade of 1990's and has been used on a number of critical real-world scenarios, the most cited of which being the migration of the flight control system of Boeing [DMS05]. An interesting fact about DMS is that it was used to generate a significant part of its own system. It allows to manipulate IR instances through an API.for various languages, and attribute analysis.

Polyglot The Polyglot framework [NCM03] is an extensible compiler framework for Java. It is maintained by the Computer Science department at Cornell University, whose first technical report dates to 2002. There is an available JavaDoc API. that is commented but does not seem very elaborate. There is not much more documentation apart from that.

Polyglot was designed to allow the creation of compilers for extensions of Java, through pre-compile transformation of these language extensions. The usage is through an imperative AST-manipulating API. also in Java. There are examples of type-checkers that do not need to perform any program transformation, and instead simply analyze the program and abort compilation in case of error. This is a good example of the application of this framework to perform static program analysis, through its program representation and manipulation interface.

TXL The Turing eXtension Language [Cor06] (TXL), is a transformation language that has been developed for more than 20 years. It has some quiet good documentation, featuring easy to understand introductory courses, and extensive, detailed reference documents for language definition and function lists. It allows fast processing of even very large applications. It has been used on a correction of the Year 2000 bug on a COBOL system comprising thousands of millions of lines of code. It defines the target language and the transformations to apply, through context-independent rewriting rules. There are ready-to-use grammars available for most common languages.

ASF+SDF The ASF+SDF Meta-environment [vHKO02] is a toolset that combines the Algebraic Specification Formalism (ASF) and the Syntax Definition Formalism (SDF). It is currently under active development, although it seems more focused on academic usage with little or no examples on production scenarios. Just like TXL and Stratego, it is better suited to simple text-based program transformations than deep semantic analysis of programs. SDF is used to specify the target language in a way similar to BNF, so it allows to handle any language. Transformations are defined by the ASF component, through rewrite rules that define the language semantics. It does not directly support context-dependent analysis such as data-flow, although it is featured enough to permit type-checking of programs and document-generators. The Meta-environment can handle large specifications, in the order of dozens of thousands of rules and millions of terms.

Stratego/XT [Vis04] is a language and toolset for program transformation that follows the same lines as TXL, but with more emphasis on the separation of concerns, in order to allow more reusability. Development does not seem very active, and it has slowly evolved since the decade of 1990. It uses the SDF language to define the target language. The transformations are made through rewrite rules, and the body of the rules can be specified on the target language. There does not seem to be much concern about performance or experimental results. There are not many practical applications outside the academic environment.

The characterization of these systems is summarized in Table 2.1. As discussed earlier, we observe a major difference in features between systems identified as *rewriters* and *compiler frameworks*.

If the goal is the implementation of a language extension or translation between languages, then a proved system like TXL allows to achieve that very quickly.

If the task is more complex then a robust compiler framework may be required. If this is the case, the most limiting factor in choosing a single system will be the target language, which may not be possible to handle in all systems. In particular, the best compiler framework for the specific task may not be prepared to work with our target language out-of-the-box, and a work-around will be required, if available. For example, although ROSE has powerful features, it is unable to work immediately with Java. If there is no satisfiable workaround then it may be preferable to have extra work with another system, which is not so well prepared for the type of transformation or analysis task that we intend, but is capable of working with our target language. DMS is a very particular system in this area. Because it is a commercial system, it is specifically targeted at industrial tools. It is able to handle projects that are written in more than one source language, or with embedded languages.

2.3.3 The Polyglot Compiler Framework

Polyglot [NCM03] is an extensible compiler framework for the conception of extensions to the Java language. Polyglot will be used in Chapters 3 and 4 as the basis for the anomaly detection tools.

Table 2.1: Comparison of Transformation Systems

	ROSE	LLVM	DMS	Polyglot	TXL	ASF+SDF	Stratego
Type	Compiler Framework	Compiler Framework	Compiler Framework	Compiler Framework	Rewriter	Rewriter	Rewriter
Stable / Mature	√	√	√	√	√	√	
Active Development	√	√	√			√	√
Documentation	Extensive	Extensive	Unknown	Poor	Good	Extensive	Sufficient
Availability	Free, Open-Source	Free, Open-Source	Commercial	Free, Open-Source	Free	Free, Open-Source	Free, Open-Source
Support	Mailing List	Mailing List	Commercial				Mailing List, IRC
Support	Mailing List	Mailing List	Commercial	Mailing List	Forum	Mailing List	Mailing List, IRC
Real-world Applications	√		√		√		
Language	C, C++, Fortran	Many, Front--end dependant	Many, Extensible	Java (extensions)	Any	Any	Any
Transformations	C++	C++	Parlanse	Java	Rewriting	Rewriting	Rewriting
Static Analysis Support	√	√					
Full Program Evaluation	√	√	√	√			
Large Inputs	√		√		√		
Lifecycle		√					

Polyglot is an AST infrastructure with several nodes to handle the full range of Java constructs. A Polyglot extension is defined as a set of passes, which iteratively traverse and transform this AST, eventually outputting the tree as new Java code which will be compiled. So polyglot may be used either for extensions to Java, translation from Java to other languages, or simply analyzing a Java program.

The first phase in defining a language extension is to define the new syntax. Rather than defining the language syntax completely, it is possible to define the syntactic *differences* between Java and the new language. This is achieved through the CUP parser generator for Java [HFP+], and the Polyglot Parser Generator [BM], an extension to CUP. The next step is to define the semantic aspects of the new language. For this, new AST nodes must be defined, as well as the new type system. Finally, the translation or analysis procedure must be defined by implementing and providing the necessary AST traversals.

As an example of a language extension which shall be used in our work, consider the extension to Java with `atomic` blocks presented in [Dia09]. The syntax is merely extended with these new atomic blocks, which are similar to other compound blocks, such as the inner bodies of *loop* statements. The semantic extension is also achieved by creating a new AST node, and some other nodes that enable the representation of fields known to be accessed atomically. Finally, the language translation is performed by AST passes which transform these atomic blocks into proper calls to a specific TM library.

2.3.4 Discussion

We have explored systems for program analysis and transformation, in order to carry out the anomaly detection approaches presented in Chapters 3 and 4. Since we will perform static analysis of anomalies, the usage of such a tool is vital. Full implementation of these approaches from scratch would be infeasible. From the set of available alternatives, we have chosen Polyglot, for fitting well into our context, for its specificity to Java, and because of the availability of previous work with this framework [Dia09]. We will also see in the following section how available anomaly detection tools, and JChord [NAW06] in particular, use these systems in order to achieve their goals.

2.4 Datarace Detection Methods and Tools

Program testing is a very wide research area. The detection of dataraces has itself been the target of intense study, dating back nearly as far as the appearance of concurrent programming. In this section we address this subject by discussing related work on datarace definition and detection, in order to assess which methodologies may be applied to the context of TM programs.

2.4.1 Dataraces

Concurrent programming means that more than one instruction might be executed in parallel, potentially on different processors. While on sequential programs one may safely assume that the program state on a specific point is derived from the preceding instructions, on concurrent programs the program state may have changed between two consecutive statements, as a result of thread interleavings. For this reason, synchronization is required between concurrent threads to avoid situations where a thread sees an inconsistent program state, because another one has overwritten a shared variable.

As an example, consider the program in Listing 2.1. In this example, two threads run in parallel, and each of them independently increments a shared variable a number of times. Suppose thread `t1` reads the value *41* from the variable, and a context-switch happens before `t1` writes the updated value. Thread `t2` may run, read the value *41* as well, and update `counter` to value *42*. If thread `t1` runs immediately after, it will still consider the old value *41*, and update `counter` to *42* as well. As we can see, there has been a *lost update*, since the program behaves as if one of the updates had never taken place. Because the code was run two times, the shared counter should have been incremented twice, but because of concurrency between the two threads, the second increment was based on an inconsistent state. We call these events *dataraces*.

Datarace Definition and Conditions

A datarace happens when one thread writes a variable, at the same logical instant that another thread is also accessing that variable, either reading or writing it. More formally, a datarace

Listing 2.1: Example of a Datarace

```
1  public class Example extends Thread {
2
3      private static int counter = 0; // counter, shared state
4
5      public void run() {
6          for (int i = 0 ; i < 50 ; i++) {
7              int n = counter;
8              n = n + 1;
9              counter = n;
10         }
11     }
12
13     public static void main(String args[]) {
14         Thread t1 = new Example();
15         Thread t2 = new Example();
16         t1.start();
17         t2.start();
18         // we want counter to be~100
19     }
20 }
```

occurs when:

1. There are two different accesses to the same memory location, and at least one of them is a write;
2. The two accesses are performed by different threads;
3. The two accesses are not guarded by at least one common lock; and
4. There is no order enforcement between these two accesses, for example by guaranteeing that the two threads are not running at the same time, by analyzing the order of thread *start* and *wait* instructions.

These are the formal *datarace conditions* that are the basis of current literature on the subject [dCLSL01, CLL+02].

Causes and Consequences

A datarace is almost always a programming error, although programs with dataraces may be correct. One notorious example is the one presented in [BLM05], in which a correct program with such a benign datarace actually exhibits a deadlock after it has been blindly (automatically) "corrected".

The damage caused by a datarace is associated with undetermined program behavior. This means that a program that appears to be correct and contain a datarace that has not yet been triggered, because a specific pair of scheduling and input, that will exhibit the datarace, has not yet been tried. A program may be run multiple times with correct behavior on one environment, and then show a datarace when run on a different system or environment. Therefore

dataraces are difficult to detect, and difficult to debug. Algorithms and tools that find dataraces, either statically or dynamically, are very valuable to concurrent program developers, and have become the target of intense research over the last decades. In the following sections we explore some of them.

2.4.2 Static vs Dynamic Detection

Like most testing and debugging procedures, the detection of dataraces can be achieved either statically, dynamically, or using a combination of both.

Static Detection

Static solutions do not evaluate an execution. Instead, they analyze the *specification* of a program. Because static detectors evaluate a program without (or before) running it, they are also referred in the literature as *ahead-of-time* (AOT) approaches [Raz06, CLL+02, OC03]. Static analysis does not have access to runtime information, and so can not make a complete and precise detection. Static datarace detectors make a conservative estimate, and as such they have the drawback of an excessive amount of false positives, reporting events that are unrealistic and may never happen in a real execution. The program specification to analyze is typically source code, but not necessarily. Many tools are based on preprocessing code annotations. There is also the possibility of using type systems to express synchronization relationships. In this case, a program that has passed a type-checker has some guarantees of correctness, with respect to dataraces. These approaches have the disadvantage of forcing programmers to annotate the code, since they they are unable to verify unannotated programs.

Dynamic Detection

Dynamic detection is made based on a specific real execution of a program. For this execution, memory accesses and synchronization operations are logged with an explicit timely order established between them. The algorithm analyzes this log and determines *if* and *when* a datarace occurred. Because the analysis is based on a feasible execution, the false positive rate is much lower than that of static analyzers. However, the amount of false negatives is expected to be higher, especially in situations where the test cases do not cover all scenarios.

Dynamic detection approaches may be categorized as either *on-the-fly* (OTF), during program execution, or *post-mortem* (PM), after execution has stopped. A dynamic detector works by analyzing a sequence of events generated during the execution. For this reason, a PM and an OTF detector that use the same algorithm are expected to always be able to find the same set of races: any information available at runtime can be dumped along with the event log, and any reasoning on the events can also be done in runtime, at the expense of increased execution time. Therefore, it is expected that an algorithm used in a PM detection tool can also be easily adapted to be run after execution, and vice-versa. The choice on whether to develop an on-the-fly or post-mortem tool is based on the compromise between (1) the absence of overhead, and (2) the advantage of having a datarace detected and reported immediately.

Hybrid Detection (Static and Dynamic)

Most tools actually use some form of hybrid approach, featuring dynamic and static analyses. Hybrid tools attempt to take the advantages of both methods: a static algorithm first finds spots where an anomaly *may* happen, and the dynamic component observes where during execution a race *does happen*. The static analyzer generates a set of statements that may be involved in a race. A statement that is not present in this set is guaranteed not to be involved in a datarace, and does not need to be observed. The remaining statements are suspected to be involved in races, and are checked at runtime by the dynamic analyzer.

Though we shall analyze tools that fit on all categories, our final goal is to develop a static tool for anomaly detection in a transactional context. The emphasis will be on analyzing static approaches, or understanding how a dynamic solution can be adapted to a static scenario.

2.4.3 Datarace Detection Mechanisms

Before we move on to describe the detection approaches, we revisit a couple of useful concepts that are recurring in the philosophy of the described algorithms. We first show what instrumentation is, how it varies and how it can be used. It is used by dynamic detectors. Then we see some program representing graphs, that are mostly used by static tools for dataflow analysis.

Instrumentation

The idea behind instrumentation is that of controlled execution. Dynamic tools should evaluate the subject program as an external observer, without perturbing the normal functioning, except by possibly altering schedulings, which are not supposed to be deterministic in either way. Examples of instrumentation-based approaches include [OC03, QDLT09];

Instrumentation is performed by adding specialized instructions to a program, that generate a log of events occurred. These instructions should not do anything else, in order not to semantically alter the program. These additional instructions generate a stream of events that is to be analyzed through a dynamic datarace detection algorithm, either PM (save a log for later) or OTF (analyze events as soon as they are available). Instrumentation may be done in different stages: at compile-time, by using a specialized compiler that generates the instrumentation instructions; before compilation, by pre-processing source code and passing it to a standard compiler; or immediately before execution, by instrumenting the program in its binary form.

When third-party code is used, such as the template library or operating-system code, some detectors instrument it as well. This has the difficulties of having to instrument a much larger code base than that of the single program, or not having access to the program code in the required form. For example, if instrumentation is performed at compilation time, and source-code for the base library of the language is not available, then it might not be possible to instrument it. On the other hand, instrumenting the template library, or widely used code, has been used to discover subtle errors that are still present in spite of the wide-spread usage [QDLT09].

Graphs

It is typical of program analysis, specially of static and dataflow analysis, to use graphs to represent the subject program and make the analysis through a graph traversal algorithm. It is intuitive to represent each statement by a node and the transitions between the statements. This way both the states (values – dataflow) and the transitions (control – control flow) are represented. This is known as the Control Flow Graph (CFG) [All70].

Special kinds of graphs are required to analyze a concurrent program. We will see in Section 2.4.4 how Dinning and Schonberg use the Partial Order Execution Graph (POEG) [DS91]. This graph simply captures the happens-before relation [Lam78] between concurrent accesses, so it has a high occurrence of false negatives. Another approach is that of the Interthread Control Flow Graph (ICFG) [dCLSL01]. This graph does not have the problems of the POEG, but it also has a different purpose. The ICFG is intended to statically infer sound information about the execution ordering of certain operations of concurrent threads. The Interthread Call Graph (ICG) [dCLSL01] is an abstraction of the ICFG, intended to represent only the interactions between threads, and drop information about relations between statements in the same thread. It is better suited for a large program analysis because of its lighter weight.

2.4.4 Datarace Detection Approaches

We will now see the main approaches used in datarace detection. Many solutions implement some form of the lockset algorithm, since it is easy to implement and produces good results. However, it is suitable only for lock-based applications and has a high rate of false positives. There are both static and dynamic implementations. On the other hand, happens-before solutions only make sense in dynamic detectors, but may be applied to other forms of thread synchronization. This approach has no false positives but has a high rate of false negatives. Most dynamic detectors use a hybrid approach combining these two in order to achieve good precision. Dataflow based detections are especially good for static analysis and work by analyzing a graph representation of the program. Finally, and also for static detection, there are annotation based solutions, such as type systems, which have good properties but require the previous modification of the target program.

Lockset

Consistent locking is a method in which the same lock or set of locks is held whenever a shared variable is accessed. This practice of associating a lock with each shared variable is very common and natural, but also error-prone. Many solutions intended only for lock-based programs analyze the presence of this practice through an approach that has become known as *Lockset*.

The Lockset algorithm is quite simple. The first time a shared variable is accessed, the set of locks currently held by the running thread is recorded, which becomes the *lockset* associated with that variable. Every time the variable is read or written after that, its lockset becomes the interception between its *current* lockset and the set of locks held by the thread making the

access. If at any point the lockset becomes the empty set, then there is no lock consistently associated with the variable, and a datarace may potentially occur.

Note that only the unprotected variable is reported. There is no indication of which accesses are not correctly protected, although in some cases it would be possible to infer the associated lock, and the accesses that do not follow consistent locking. Some optimizations to the algorithm take into consideration read-only variables, or initialization accesses during which the variable is not yet shared.

Warlock [Ste93] was one of the first approaches to use lockset, although this name does not appear on the paper. It is an experimental tool developed by Nicholas Sterling from SunSoft Warlock is a static analysis tool for C programs. Lock_Lint [loc] is another tool from Sun, based on Warlock and available with Sun Studio. However, it is likely that the lockset algorithm was invented by Dinning and Schonberg [DS91], in order to solve many issues with their happens-before approach (see following section). Ironically, it was quickly understood that this workaround could stand an algorithm on its own, and many tools appeared that dropped the happens-before approach and started to analyze locking policies only. Examples include RacerX [EA03] and, more recently, Relay [VJL07]. These two are actually based on Eraser [SBN+97], a dynamic tool which was one of the earliest approaches, and probably the first to coin the name Lockset. In the paper, a detailed description of the algorithm is provided. Finally, other notable approaches that use lockset are RaceTrack [YRC05], MulticoreSDK [QDLT09] and [CLL+02, OC03], which, like Dinning and Schonberg [DS91] use a hybrid approach with happens-before relations.

The application of the lockset algorithm in the context of Transactional Memory is not absurd, though it does not seem as useful as in a lock-based scenario. Remember that the usage of TM is equivalent to using one single global lock that is acquired whenever a transaction begins. So if the algorithm was run it would be expected to report all variables that are sometimes accessed transactionally, and other times outside of a transaction. Variables that are shared but never protected are also in risk of race conditions, and lockset tools are expected to report this potential anomaly. So the algorithm could find potential anomalies, both in weak and strong isolation [LR06], derived from accesses that are not (or are incorrectly) protected.

Happened-Before

This approach is based on Lamport's work on partial ordering [Lam78]. Unlike lockset, algorithms based on happened-before relations (HBR) only make sense in dynamic analysis. During program execution, the program's memory accesses, as well as synchronization and thread manipulation mechanisms are logged and recorded in the order they are executed. It is known that the statements of a single thread are executed sequentially, so a happens-before relation is established between them. When a *signal* from a thread (such as a lock release) is *received* by another (acquiring the lock it was waiting on), a happens-before relation is also established. Note that this relation is established for this single execution, not meaning that it will hold for all runs. In fact, the threads could have been the scheduled in the opposite way, reversing the

established HBR, and exposing a previously unseen datarace (or hiding a datarace that had been detected).

This is the cause of the high rate of false negatives of this approach. Imagine that thread `T1` updates variable `x` and then acquires lock `L1` to write `y`, then releases `L1`. Thread `T2` acquires `L1` *after* `T1` releases it (establishing an HBR), releases it too, and *then* updates `x`. The algorithm will establish an ordering between all these statements and not accuse the datarace that could happen (and would be detected) if `T2` had been scheduled to run before `T1`.

In spite of the high rate of false negatives originated from the problem stated above, note that there are no false positives. If two concurrent accesses are logged to the same variable, with absolutely no synchronization between them (no HBR) then there are no access guarantees, and the conditions for a datarace are fulfilled.

Early approaches were based on this technique, but recent ones never use HBR alone. Instead, they integrate it in a hybrid solution, combining HBR and Lockset as will be seen in Section 2.4.4. Consider Dinning and Schonberg [DS91] who designed a HBR solution and were among the first to describe lockset, with the purpose of improving their tool.

This technique is only available in a dynamic context, which does not fit our final goals. It is possible to infer HB relations that are known to be held at execution time (such as when a thread starts or is waited on), but this is a different technique that is related to dataflow analysis. We will see later in this section how [dCLSL01] does this using flow graphs. Happens-before relations are known to work with any synchronization mechanism and not only locks. For this reason it is expected that an ordering of events in a TM scenario could help detect concurrency and atomicity anomalies, depending on the concrete definition of such anomalies.

Hybrid (Lockset and Happens-Before)

Almost all recent dynamic approaches use some form of hybrid algorithm that combines Lockset and HBR. Remember that lockset has many false positives and no false negatives, and only fits lock based programs. HBR on the other hand has many false negatives but no false positives. So researchers started implementing solutions that attempt to combine the advantages of both. Among many approaches we highlight [DS91, YRC05, CLL+02, QDLT09]. Hybrid approaches have far better precision that either lockset or hb alone, and a desirable performance as well. In fact, no presented tool provides perfect precision (i.e., no false negatives and no false positives) because, in some way or another, precision is slightly sacrificed in order to obtain a satisfying performance. Solutions typically have a low rate of false negatives and no false positives.

Dataflow Analysis

Dataflow analysis attempts to determine the possible values of a program at a specific point. Although verification of datarace conditions is closer related to guaranteeing time order between events, dataflow plays an essential role in statically assessing the presence of dataraces. Pointer analysis, which determines the set of memory areas a reference may be pointing to,

is fundamental in discovering whether two dereferences from two different threads may (or must) access the same object or variable, as well as asserting that the lock objects protecting two critical regions are, or can be, the same. This is an important distinction. A may-points-to says that it is possible that a race happens, but does not assure it (false positives), whereas a must-point-to analysis states that flagged accesses are racy, but may not point all races (false negatives).

In order to perform a static analysis, the subject program is handled in an intermediate representation that takes the form of a Control Flow Graph (CFG) [All70]. Each statement in the program is represented by a node in the graph, and the edges are the possibles control transfers between statements, according to program sequence and branching. These edges may be intraprocedural (between statements in the same method) or interprocedural when methods are called). Similarly, statement execution resulting from the creation of a new thread can be connected through an interthread edge. For this purpose, [dCLSL01] introduces the Interthread Control Flow Graph (ICFG), which features these new edges. The Interthread Call Graph (ICG), also presented in [dCLSL01], drops some information about flow between sequenced statements, and preserves only edges pertaining method calls and thread spawning. This kind of graph is, therefore, not as suitable to perform pointer analysis, but it better allows to infer properties related to thread synchronization and concurrency anomalies.

The standard way of performing dataflow analysis is through the enunciation of equations associated with each node in the graph, which are solved iteratively by traversing the graph. Program state flows between nodes, with the result of one equation affecting the equation of its successors. The output of a node represents the possible program states after the statement, which is passed to the input of the following node. Alternatively, in the case of a backward analysis (such as live variables), the opposite process takes place.

Pointer analysis is performed by traversing the graph and analyzing memory allocations and reference variables. Whenever an object is allocated or a local variable is defined, its accesses are observed in order to detect conflicts. Whenever a reference variable is re-assigned, or dereferenced, the set of possible memory locations it is pointing to is analyzed so as to infer whether these same locations may have been accessed elsewhere without synchronization. In this process, escape analysis also plays an important goal in optimizing the procedure. Escape analysis determines whether a locally allocated memory space may escape the current subroutine and be used outside, in undetermined points in the program. If a memory location is guaranteed to always be accessed locally, it is guaranteed to never be involved in a datarace and needs not be observed. Another essential mechanism in optimizing static datarace detection is the weaker-than relation, which serves as base for [QDLT09] and [CLL+02]. The weaker-than relation is established between two accesses to the same variable by the same thread. In a code block, if one of the accesses is less protected than the other one, then it is said to be weaker than the later. In this situation, if the weaker access is known to be involved in a datarace, then the remaining access is guaranteed to participate in the same race, without need for further analysis. This way, many accesses may be discarded, eliminating computation time, and avoiding the report of redundant races.

As we can see, dataflow analyzers run through several stages in which the pairs of accesses potentially involved in a race is iteratively reduced. Even though it is a dynamic analyzer, the MulticoreSDK [QDLT09] is a good example, running various analyses based on pointer analysis that reduces to a minimum the set of statements that must be analyzed during runtime. Another notable work is the one around Chord [NAW06]. This is a static detection tool capable of treating various synchronization idioms (and not just locks), with good precision and performance even on large programs.

Because our goal is to develop a static detection tool, the techniques presented above are excellent candidates to form the basis of the approach. Recent static tools are capable of handling many different types of synchronization with a very good precision. The application to transactional memory appears feasible and useful, and even if we intend to apply a variation of the lockset, points-to analysis as used in [dCLSL01] should be the basis for the implementation.

Annotations

Most static analysis tools take the program in their original form, be it before or after the compilation. However, instead of inferring behavior from the program specification, we will see two examples that rely on information explicitly provided by the programmer in order to reason about about properties of the input.

The first example is a type system developed by Flanagan and Freund [FF00], that statically tries to ensure the absence of dataraces at runtime. Some types are inferred, and the authors minimize the inconvenience of requiring the existing source-code to be retyped, but the user still has to review the subject program before submitting it to the verifier.

The second example is the Extended Static Checker for Java version 2, ESC/Java2 [CK04]. This tool is a model checker that statically detects many anomalies, such as possible null pointers, overrunning arrays, etc. It works by analyzing *pragmas* in the form of Java Modeling Language (JML), directives written inside specially formatted comments. Instead of using methods specifically for concurrency anomalies detection, ESC/Java2 takes a broader approach and uses established logic techniques. A theorem prover is called in order to prove various assertions about the program, in a problem-solving-fashion.

While these approaches are interesting, they have the disadvantage of requiring the code to be annotated, even though the authors try to emphasize their advantages and the small amount of annotations required. The revision of the subject program may be infeasible in the case of a large code base. We will not discard of using such techniques to improve the quality of the tool that is intended to be developed. However, it would be preferable to to develop a tool that can take an original uncommented program and be as precise as possible.

2.4.5 JChord

JChord [NAW06] is a framework for static program analysis of Java programs in bytecode form. JChord sits on top of the Soot framework [RC99], which in turn uses the Polyglot framework [NCM03] as its frontend.

Architecture

JChord takes advantage of the basic analysis features of Soot, and extends them with higher-level analyses. JChord defines program analyses by defining equations on the AST nodes. The equations are solved iteratively on the nodes, whose results affect the equations on adjacent nodes, until a fixed point is reached. These analyses are arranged in a modular and compositional way, in that each can be deployed or taken off, on demand. For example, the datarace analysis makes use of pointer analysis, lock analysis, and call-graph analysis, among others. JChord can be extended with user analyses either declaratively, with Datalog equations, or imperatively, by implementing Java code that extends the framework. Out of the box, JChord provides detection of many concurrency anomalies, such as datarace and deadlock. JChord analyzes the Java Byte Code, and makes use of the source-code only to map back anomalies to specific points in code.

The tool will always analyze the whole program, including its libraries, even if those libraries are the JRE. This leads to a peculiar usage experience with the tool, since analyzing even the smallest of programs will take around two minutes. However, it seems well scalable and robust at its current stage.

Usage

JChord's source can be downloaded with all required libraries and deployed within minutes. JChord is ran through an Ant build file that sets up all system properties required for the specific analysis, and then launches the execution. In order to run JChord, the Ant file is summoned, with a target specific to the intended analysis. An argument must be passed, which specifies the working directory for JChord. This directory is where JChord expects to find the `chord.properties` configuration file, which will say where to find all required files related to the target application, as well as the main class. An example is presented in Listing 2.2. This

Listing 2.2: Example of JChord configuration file.

```
1 chord.main.class=HelloWorld
2 chord.class.path=../output/
3 chord.src.path=../output/
4 chord.out=../output/log.txt
5 chord.err=../output/log.txt
6 chord.serial.file=./Hello.ser
```

working directory is also where JChord will place and reuse all output files. These are not only the final output files, with the listing of anomalies, but also intermediate representations of the program in many forms.

JChord generates a visual output for the detected races, which may not be fully intuitive for a new user. In order to help understand this data, we describe the output format in the next section.

Output Format

JChord's datarace detection component generates an HTML output, listing the races found anywhere in the subject program. The output consists of two different reports: grouping by object, and grouping by field. Either one will present the same number of rows and the same detected races, but in a different arrangement. Figure 2.1 is an example of output. The

Datarace Reports (Grouped By Field)				
Details	Trace 1		Trace 2	
	Thread	Memory Access	Thread	Memory Access
1. Dataraces on Hello.x				
1.1	Hello.main(java.lang.String[]) Context: [main]	Hello.main(java.lang.String[]) (Wr) Context: [main]	java.lang.Thread.start() Context: [Hello]	Hello.run() (Wr) Context: [Hello]

Figure 2.1: Sample HTML output produced by JChord.

grouping is made by the yellow rows, and each of the other rows represents a race. The two main columns, `Trace~1` and `Trace 2` indicate the two entities involved in a race. Column `Details` simply provides a numbering for each race, with the major number being a sequence number of a field or object, and the minor number being the sequence number of the race on that item. Columns `Trace~1` and `Trace 2` are each divided into two sub-columns of `Thread` and `Memory Access`. Column `Thread` shows the starting point of execution of the thread making the access. This will be either the main method of the program (in the case of the main thread), or the method `java.lang.Thread.start()` (in case of a user spawned thread). Under this there is the `Context` indication, which is either `main` or the class which is `java.lang.Thread` or a subtype. There is a link here to the point in code where the Thread object is created. The other sub-column of each trace is `Memory Access`, showing the method in which the conflicting access is performed. There is a link to the exact code line, and the indication of whether this is a read (`Rd`) or write (`Wr`) access. Underneath, in the `Context` area, we have the type of the object performing the access, and a link to the point where this object is constructed.

2.4.6 Discussion

Throughout this section we have analyzed the current techniques in use to perform datarace detection, dynamically and statically, in programs that use either locks or thread communication (or both) for synchronization. We have seen how the lockset algorithm is popular for its good results and ease of implementation, in spite of the limitation to locks (which are still the most common synchronization mechanism) and the false positives. We have also seen how the lockset algorithm can be combined with happens-before relations between concurrent accesses in order to highly improve the precision of a dynamic tool, while still maintaining a good performance. We then saw how dataflow analysis based solutions provide good guarantees, either in order to aid a dynamic tool, or to develop a precise static tool. These tools infer properties of programs based on the analysis of a graph-based representation. Finally, we took a brief overview at some static solutions that receive semantic information other than the one that can

be inferred from the source code itself.

It has been concluded that all the analyzed methodologies can in some way be put to practice in the context of transactional memory. The implementation of the lockset algorithm is a particular case of a technique that can be used in a more relaxed form, since the utilization of TM is semantically equivalent to using one single global lock. The check on the locks can be extremely simplified to a binary form, but implementing the algorithm is still not trivial. Dataflow techniques and happens-before relations are related to program states and behavior, respectively, and not to lock utilization practices. Therefore, they may possibly be used in the conception of a transactional testing tool. Remember that HBR is only available in a dynamic context, while dataflow analysis are almost always used in a static environment. We shall use these techniques as the basis for the conception of our tool. Type systems and other approaches based on annotations have the undesired effect of requiring previous modification of source code, and we will put them aside, at least presently.

We expect to be able to develop a static tool that could aid in developing transactional code, both by detecting basic deficiencies in mutual exclusion, as well as detecting accesses that have atomicity violations, such as two-stage accesses.

3

Low-Level Dataraces in Transactional Memory

3.1 Introduction

This chapter discusses the detection of low-level (or weak-isolation) dataraces in TM programs. We aim at demonstrating how this goal can be achieved by using currently available detection methods for monitor-based programs, by converting TM programs to use monitors. In particular, we will discuss the viability of taking transaction blocks and replacing them with blocks protected by a single, global lock that is automatically generated.

In order to validate this approach, we not only discuss exhaustively the theoretic implications of this approach, as we also provide a set of practical experiments and tests, whose results satisfy our claim. This theoretical discussion will take place in Section 3.2, Section 3.3, and Section 3.4, where we will see the semantical difference between these two mechanisms, explore these implications on the different formal definitions of dataraces, and provide arguments for the validity of making such a conversion. In Section 2.4.5 we will take an overview of JChord [NAW06], the tool that serves as basis for our datararace detection. In Section 3.5 our implementation is presented in detail. In Section 3.6 we present the practical tests with which we intend to validate our approach. Finally, in Section 3.7, we summarize this chapter.

3.2 Semantics

In order to establish a parallel between lock-based approaches and transactional memory, we discuss the behavior of each, and the guarantees they provide. We will discuss the semantics of each of these mechanisms, and see how they correlate. By the end of this section, we will

have established if, and when, we can use locks and TM interchangeably.

Given two concurrent sequences of operations, we say that these sequences are *serialized* if one of them starts only after the last operation of the other one has finished. We call *serialization* to an execution ordering that is serialized. If concurrent execution of multiple blocks is guaranteed to leave the system in the same state that would be achieved if these blocks were serialized in some way, then this execution order is equivalent to some serialization, and we say that it is *serializable*.

Locks guarantee mutual exclusion between regions guarded by a common lock, meaning that no two blocks guarded by a common lock should ever execute in parallel. No guarantees are provided about blocks that are not protected by locks, or blocks that are lock-protected but do not hold at least one common lock. We see that by enforcing mutual exclusion, the lock protected blocks are serialized in some way. There is no way of knowing ahead-of-time which serialization will take place; all we know is that such a serialization will be achieved. Consider the example in Figure 3.1.

```
// block~1
synchronized (a) {
    ...
}
```

```
// block~2
synchronized (a,b) {
    ...
}
```

```
// block~3
synchronized (b) {
    ...
}
```

Figure 3.1: Concurrent regions.

Whatever interleaving is observed at runtime, code blocks 1 and 2 will be serialized, as well as 2 and 3. However, there are no guarantees between 1 and 3.

Contrarily to locks, TM does not (necessarily) provide mutual exclusion. The guarantees and semantics of TM vary with the specific system and implementation. Due to the relatively recent emergence of TM, there is still not a commonly adopted interface, and there are many different systems that provide different guarantees. Nevertheless, the most common scenario is to provide Atomicity and Isolation. Atomicity is required to ensure that a transaction's effects are reversed back if a conflict occurs. Without this guarantee, the essential purpose of TM would be frustrated. However, for each transaction in particular, and while it is executing, the most interesting property is that of Isolation, because it is related with the perspective of a single transaction. Each transaction must have the perception that it is the only transaction executing at that moment. A transaction must "feel" like all other concurrent actions have been suspended, and that they will only resume when this transaction has finished. This logical execution is actually what happens in reality when using locks. We can see how the use of transactional memory will provide serializability.

3.3 Dataraces

A datarace is a conflict that happens between two concurrent accesses to shared data, when at least one of them is a write access, and there is no synchronization between the two. A datarace is related to indetermination and invalid states.

The formal datarace conditions are described in the literature as the following [OC03, DS91, CLL+02, dCLSL01]:

1. The two accesses are made to the same memory location, and at least one of them is a write access;
2. Each of the two accesses is performed by a different thread;
3. There is no common lock held by both threads when the accesses are performed;
4. There is no ordering enforcement between the accesses.

We will call conditions 1 and 2 the *universal conditions*, because they are inherent to a concurrent program. Without these conditions, there is no interaction between threads, and each thread is a standalone program, sharing no resources other memory. No concurrency problems related to memory access will arise in this situation. Any datarace definition should include these conditions, independently of the level of race, and the concurrency control mechanisms.

The two remaining conditions are specific to concurrency control mechanisms. Condition 4 says that accesses may happen at the same time, while condition 3 says they are not mutually exclusive. We can see that the purpose of protecting a critical section with a lock is that, even if the accesses may be scheduled to be performed simultaneously, mutual exclusion is enforced, thus providing serialization. Conditions 3 and 4 together say that there is no serialization between the accesses, and the datarace definition revolves around this.

In a transactional context, we will define a low level datarace as lack of serializability, similarly to races in locks. This means that, given the universal conditions, accesses can happen at the same time, and are not serializable because one or both of them are not inside a transaction.

When using locks, a race happens when a critical section is incorrectly unprotected by the relevant lock. Similarly, when using TM, a race happens when a shared variable is incorrectly accessed outside the scope of a transaction. Because a lock protects a critical code section, in the same fashion a transaction does, we will explore the possibility of applying an available lock-based datarace detector to transactional programs.

3.4 Conversion

In Section 3.2 we have seen that, although the pragmatic behavior in transactions and monitor-protected blocks is often the same, the semantics offered by each one is different.

This difference has been explored by Blundell et al. [BLM05], who pointed out the dangers of blindly converting a program with monitors to an "equivalent" one using TM: a program converted by replacing each synchronized block with an atomic block may become incorrect. By performing this conversion, two synchronized blocks using different locks, which were allowed to run concurrently, will now be unable to interact. If the converted blocks relied on some other form of synchronization, such as a *wait/notify* idiom, such a program could incur in a situation similar to a deadlock.

On the opposite scenario, if we consider a TM system that only flags write-write conflicts, then we are dealing with more relaxed restrictions for correctness than usual. In particular, if a program that is correct according to this semantics is evaluated by a typical datarace detection engine, then false conflicts could potentially arise due to unprotected reads, which would not be considered conflicts at runtime.

Despite these differences, we know that, in most cases, the executions of a transaction and a synchronized block have the same pragmatic outcome. Protecting a code region with a lock will make each thread wait until it acquires that lock and is the only one executing this code, resulting in serialization. Declaring a transaction means that the underlying TM system will monitor it, checking for read-write or write-write conflicts, aborting and repeating transactions if required. This also results in an output that should be observable if we were running one of the possible serializations.

Consider the example in Figure 3.2. If we know that locks `X` and `Y` are the same, then we may assume mutual exclusion, and therefore serialization; otherwise, we may not.

```
// block~1
synchronized (X) {
    ...
}
```

```
// block~2
synchronized (Y) {
    ...
}
```

Figure 3.2: Concurrent synchronized blocks.

Now consider the different example in Figure 3.3. We need not satisfy any condition in order to be able to assume serializability of these two blocks. So if we convert transactions to blocks synchronized on a global lock, what is done in practice is that we satisfy the condition of `X` being the same as `Y`. Therefore, we are converting something that should be serialized (according to the conditions provided by the TM system) into something we know will be serialized at runtime.

```
// block~1
atomic {
    ...
}
```

```
// block~2
atomic {
    ...
}
```

Figure 3.3: Concurrent atomic blocks.

We can also see a difference in restrictions. If we look at a program such as the one in Figure 3.2, we have to make some assertions in order to be sure there is no race possible between any of the statements in blocks 1 and 2. In the case of programs like the one in Figure 3.3, no assertion is required to make such an affirmation, as by definition transactional blocks are race-free. If we would take the program in Figure 3.2, and replace all the locks with a global lock, it would have a more restrictive concurrency level. Therefore, the converted program would necessarily contain less races than before, although it could now be incorrect, as Blundell et al. [BLM05] have shown.

We shall replace all transactions, denoted by the `atomic` construct, by a synchronized

block, containing the same statements of the original transaction. All these blocks must be synchronized on a new global lock object, having an unused name in order to avoid name collision. This transformed program is then fed to JChord, described in Section 2.4.5, which will flag any variable access that is made without holding our global lock. In practice, this is expected to be equivalent to the set of all accesses that were made outside the scope of a transaction, in our original program.

It should be noted how these incorrect memory accesses are only considered to be involved in a datarace if we consider the context of weak isolation, in which transactions are only protected from each other. However, it has been concluded in Chapter 1 that even in strong isolation, where a transaction is fully protected, this pattern of non-transactional accesses can be an indicator of errors.

3.5 Datarace Detection

The whole logic of the detection is embodied in a shell script that takes the process through three phases, presented next.

As shown in Figure 3.4, the first phase is the transformation process, achieved with the aid of Polyglot. Our extension of the Polyglot framework includes the parsing of the new `atomic` blocks, which shall be used to denote transactional code regions. It also provides two passes for the Abstract Syntax Tree (AST): one to generate the global lock, and one to replace the atomic blocks with equivalent synchronized blocks. This process shall be described in the following subsections.

The second phase takes this output program and feeds it to JChord, in order to perform the analysis. Some care is required in order to have the output files arranged in a way suitable to be processed by JChord, along with the proper configuration file.

The third and final phase is simply presenting the HTML output from JChord automatically to the user.

The shell script takes as arguments the list of source files to analyze, together with a special argument for indicating the main class of the program.

3.5.1 Parsing

Our starting point is an existing extension to the Polyglot compiler framework [NCM03]. This extension [Dia08] extends the base Java language with a new construct, the `atomic` block. Both the syntax and the intermediate representation are already updated to cope with this new feature. The current implementation of this extension transforms the AST, replacing these atomic blocks with calls to routines of a specific TM library. For the purposes of our work, we have disabled this AST manipulation and provided our own, which is described in Section 3.5.2.

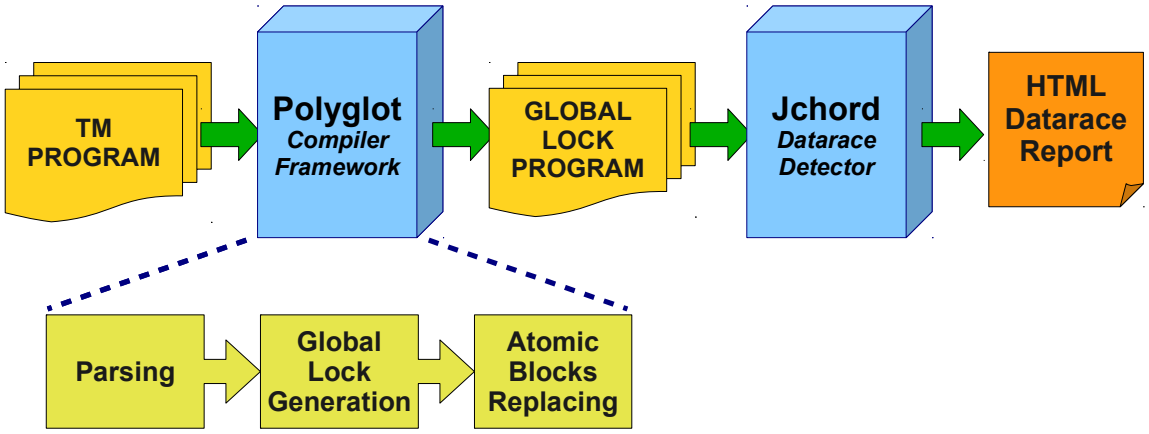

Figure 3.4: Low-level datarace detection approach

3.5.2 Global Lock Generating Phase

We must define a new shared variable that references a global lock. In order to use this lock throughout the program, there are two required decisions: the site at which to declare the variable, and how to name it.

The shared reference should preferably be declared in the main class. The main class should always be stated when calling the script, as this will be required for JChord to make its analysis. If the polyglot extension is being called alone, and not together with JChord through the script, then no main class is required, and the declaration site will be at the first class declaration that is parsed.

The shared lock reference should be named carefully, in order to avoid name collision with existing symbols. Because of the Java naming system, we know we can refer to a variable without ambiguity if we refer to its fully qualified name, i.e., its package, sub-package and enclosing class. We therefore need only worry with name collisions inside the main class. When the declaration context is determined, the tool will generate a random name that is contrary to the naming conventions in Java. This name is then checked to see if it is already in use, and if it is, then a new name is spawned. The process continues iteratively, in a way in which no name should be tried twice, until a name is found that is not in use.

At this stage, a new AST node is created, that represents the declaration of the global lock, of which the following Java declaration is an example:

```
public static final java.lang.Object
        _GLOBAL_LOCK_11056_ = new java.lang.Object();
```

The `public` modifier ensures the reference is available from any context. The `static` modifier means that the reference will be available even if no object instance exists, further ensuring the availability of the reference variable. It also makes sure that the variable is initialized as soon as the program starts. The `final` modifier forbids future modification of this variable. This is expected to be redundant, as no previous accesses to our newly created variable are

allowed, but we make this declaration as a measure of precaution. Finally, we define the variable type simply as `Object`, since every instance of this super type has an intrinsic lock. These intrinsic locks are the ones acquired whenever a `synchronized` keyword is found, and so we construct a simple `Object()` without the need to create a specialized type.

An additional task of the lock generating phase is assembling the configuration file for JChord. The parametrization is mainly related to the required definition of the main class, in order for JChord to have a starting point in the program.

3.5.3 Atomic Blocks Replacing Phase

The task of this pass is to filter all AST nodes referring to `atomic` blocks, and replace them with a `synchronized` block node that acquires the previously generated global lock. This AST node with the reference to the global lock will always use its fully qualified name, including package, in order to avoid ambiguities, or references to undeclared symbols.

The lock generating pass, described in Section 3.5.2, will set a `done` flag as soon as it is able to generate a lock. If the previous pass parses the entire AST without finding a suitable site for variable declaration, either because the main class does not exist, or because no class was defined, then this flag will not be set. The block replacing pass will detect this condition and abort the whole process if required.

3.6 Tests and Experiments

In this section we will take a look at some experiments, performed with the purpose of validating our approach.

The first tests are a series of *Hello World* subjects, which are very simple programs that allow us to evaluate the behavior of JChord, as well as to demonstrate the validity of our approach in simple examples.

After these simple tests conceived specifically to test our work, we will use a series of well known tests that are recurrently used to assess the quality of error detection tools. These test programs contain well studied errors, and are often cited in the literature. The source code for each test was obtained either from the IBM Research concurrency benchmark repository [IBM], or from the repository from Brigham Young University [RM09]. These two repositories contain different tests, and different versions of each, and we will always specify which source we used.

Finally, we turn our attention to Lee-TM [AKW+08], a renowned concurrency benchmark specially fitted to assess the performance of transactional memory implementations. We will run a datarace detection on the original code of this benchmark, and then attempt to produce a TM version of it. This converted program will also be scanned for races, with our own conceived tool. We will then compare the results of both versions, and try to extract some conclusions that may provide hints of the validity of our approach.

In order to test our tool, we will also attempt to devise TM versions of these available benchmarks, whenever it makes sense. Some of the errors make no sense in the context of

TM. Others can be ported to TM, but with different meaning. A few can be converted directly, maintaining the essence of the test. In either case, we will always describe the conversion process, and discuss any non-obvious decision.

Each of the following sections describes an experiment. The common purpose to all of them is to assess the validity of our approach. The general procedure is to: 1) take the Java version of the subject and run JChord on it; 2) take the converted TM version and run our wrapping tool on it; and 3) compare both results and extract conclusions. Specific purpose and procedure details are discussed in each experiment, along with a description of each test subject, the preparation, the test results and its analysis, and the conclusions that can be inferred from them. In Section 3.7 a summary and overall discussion of the tests is made, along with the conclusions of this chapter.

3.6.1 Empty Program

In order to get a first experience with JChord, we give it the most simple program possible. The *Empty* program is a single class declaration, with a standard `main` method without any statements. This is shown in Listing 3.1.

Listing 3.1: Empty program test

```
1 public class Empty {
2     public static void main(String[] args)
3     {}
4 }
```

When running JChord on this subject program, an exception related to the BDD library is thrown. There seems to be an error that makes this library unable to process empty domains. We performed the same test on variations of the program, with class variables, methods, and even variable declarations in `main`, all with the same result. Finally, testing the original empty program with only one output statement allows JChord to finish correctly, flagging no races. It seems that the application is unable to process programs without at least one statement.

Conclusion: The framework is unable to process this test.

3.6.2 HelloWorld Single-Threaded

The next simple program is presented in Listing 3.2, which is a single-threaded program that makes a simple output and then terminates.

After analyzing our subject program for 1 minute and 42 seconds, JChord does not find any races, as expected.

Conclusion: Results are corroborative. No races were found, as expected.

Listing 3.2: HelloWorld Monothreaded

```
1 public class HelloMono {
2     public static int x = 0;
3     public static void main(String[] args) {
4         x = 42;
5         System.out.println("Hello: " + x);
6     }
7 }
```

3.6.3 HelloParallel Buggy

The test shown in Listing 3.3 is similar to the previous *HelloWorld* test, but it spawns a child thread and has a simple datarace.

Listing 3.3: HelloWorld Multithreaded, with a bug.

```
1  public class HelloParallel extends Thread {
2      public static int x = 0;
3
4      public static void main(String args[]) {
5          new HelloParallel().start();
6          System.out.println(x);
7      }
8      public void run() {
9          x = 42;
10     }
11 }
```

As it can be seen, both the main thread and the child thread access shared variable `x`. The main thread reads the value, while the spawned thread updates it, with no synchronization between them.

This test consists in running JChord on this program, which takes 1 minute and 48 seconds. The race between lines 6 and 9 is found, as expected.

Conclusion: Results are corroborative. The datarace was found, as expected.

3.6.4 HelloParallel Buggy Synchronized

The test illustrated in Listings 3.4 and 3.5 is mostly the same as the previous one, spawning a child thread that incurs in a conflict with the main thread. This time, one of the accesses is protected by either a `synchronized` or an `atomic` block, but the error is still present because the remaining access is not protected.

This is our first test featuring a transaction. While the synchronized version is passed directly through JChord, the transactional version is pre-processed by our application.

Both tests take around 1 minute and 40 seconds. The existing datarace is correctly found in both cases.

Although our tool will simply output a program much similar to the Java version, this serves as a preliminary validation of our method.

Listing 3.4: HelloWorld Multithreaded, with a bug (synchronized)

```
1 public synchronized void run() {
2     x = 42;
3 }
```

Listing 3.5: HelloWorld Multithreaded, with a bug (atomic)

```
1 public void run() {
2     atomic {  x = 42;  }
3 }
```

Conclusion: Results are corroborative. The datarace was found, as expected.

3.6.5 HelloParallel Buggy Corrected

This test is mostly similar to the previous one, but this time we have fully corrected the bug by protecting both conflicting accesses. The new versions are presented in Listings 3.6 and 3.7, with the protection around lines 7 and 11.

Listing 3.6: HelloWorld Multithreaded, corrected (synchronized)

```
1  public class HelloParallel extends Thread {
2      public static int x = 0;
3    public static void main(String args[]) {
4      HelloParallel hp = new HelloParallel();
5      hp.start();
6      synchronized (hp) {
7        System.out.println(x);
8      }
9    }
10     public synchronized void run() {
11       x = 42;
12     }
13 }
```

Listing 3.7: HelloWorld Multithreaded, corrected (atomic)

```
1  public class HelloParallel extends Thread {
2      public static int x = 0;
3    public static void main(String args[]) {
4      HelloParallel hp = new HelloParallel();
5      hp.start();
6      atomic {
7        System.out.println(x);
8      }
9    }
10     public void run() {
11       atomic { x = 42; }
12     }
13 }
```

As usual, we ran the Java version through JChord and the TM version with our own wrapping tool. Both tests took around 1 minute and 40 seconds, and did not flag any races at all, as was expected.

Conclusion: Results are corroborative. No dataraces were found, as expected.

3.6.6 HelloWorld Modular

In order to assess the capacities of our tool, and the behaviour of JChord, in more complex scenarios, we developed a program that is similar to the previous ones, but whose logic is spread in multiple classes and source files. The envolved classes will this time belong to a named package.

The source code for the Java version of this test is in Listing 3.8. The TM version is essentially the same, with the synchronized `print` method replaced by a regular method with an `atomic` block surrounding all its statements. The race is still present, but it will be a conflict between accesses by different classes. The conflicting field is the static `x` field from class `Main`.

The thread spawned in line 7 runs the code of a `RunMe` object to modify `Main.x`. Meanwhile, the main thread concurrently runs inside the code of the `Printer` class, which prints the contents of the shared variable.

Listing 3.8: HelloWorld Modular

```
1  // FILE: Main.java
2  package multi;
3  public class Main {
4      public static Integer x = 0;
5      public static void main(String args[]) {
6          RunMe rm = new RunMe();
7          MyThread mt = new MyThread(rm);
8          Printer p = new Printer();
9          mt.start();
10         p.print(x);
11     }
12 }
13
14 // FILE: MyThread.java
15 package multi;
16 public class MyThread extends Thread {
17     public MyThread (Runnable r) {
18         super(r);
19     }
20 }
21
22 // FILE: Printer.java
23 package multi;
24 public class Printer {
25     public synchronized void print (Object msg) {
26         System.out.println(msg);
27     }
28 }
29
30 // FILE: RunMe.java
31 package multi;
32 public class RunMe implements Runnable {
33     public void run () {
34         Main.x = 42;
35     }
36 }
```

Both tests took around 1 minute and 50 seconds. The datarace was correctly identified on both cases, with no false positives. Furthermore, the exact point in code for the accesses was also identified correctly, in the different accessing classes.

Conclusion: Results are corroborative. The datarace was found, as expected.

3.6.7 HelloWorld Complex

This version will test the capability of analyzing a more complex program. The two versions of the program are in Listings 3.9 and 3.10. Two child threads will be spawned instead of one, in lines 4 and 5. An additional variable `y` will be used, along with more accesses. The main thread

accesses both variables inside a critical region starting in line 6. The synchronized block uses the `HelloComplex.class` intrinsic lock because it is an available object. Any lock would lead to the same results, since this is the only synchronized block in the program. The two spawned threads increment both variables in lines 13 and 14, without any kind of synchronization.

Listing 3.9: HelloWorld Complex, Java version.

```
1  public class HelloComplex extends Thread {
2    private static int x = 0, y = 0;
3      public static void main(String args[]) {
4      new HelloComplex().start();
5      new HelloComplex().start();
6      synchronized (HelloComplex.class) {
7        x = 42;
8        y = 47;
9      }
10     System.out.println(x);
11     }
12   public void run() {
13     x++;
14     y++;
15   }
16 }
```

Listing 3.10: HelloWorld Complex, TM version.

```
1  public class HelloComplex extends Thread {
2    private static int x = 0, y = 0;
3      public static void main(String args[]) {
4      new HelloComplex().start();
5      new HelloComplex().start();
6      atomic {
7        x = 42;
8        y = 47;
9      }
10     System.out.println(x);
11     }
12   public void run() {
13     x++;
14     y++;
15   }
16 }
```

The Java version is ran directly through JChord, and the TM version is processed by our tool. As expected, both tests detect the same number of races, because the TM version is converted to a program much similar to the Java version.

The tests take 2 minutes each, and find 9 races on both versions. To analyze this result, consider that the accesses performed by `main()` are:

1) W(x) in line 7

2) W(y) in line 8

3) R(x) in line 10

The accesses performed by `run()` are:

4) R(x) in line 13

5) W(x) in line 13

6) R(y) in line 14

7) W(y) in line 14

It is easy to see that there are conflicts between accesses 1-4, 1-5, 2-6, 2-7, and 3-5. Additionally, because two spawned threads are running, the method `run` can conflict with itself. This results in conflicts between accesses 4-5, 5-5, 6-7, and 7-7, summing 9 races in total.

With this experiment we have tested the capability of processing more complex programs, composed of multiple units, and with multiple dataraces that were all correctly pointed. No false positives were present, and no false negatives are expected to exist.

Conclusion: Results are corroborative. The dataraces were found, as expected.

3.6.8 Account

The *Account* test is a well known test discussed in [IBM, RM09, DPE06]. Its theme is a bank account management system. An error in synchronization will lead to inconsistent states, where a bank transfer will take the money out of the source account, but will not deposit it in the destination account.

Test Preparation

Each account object may be accessed by more than one thread, and so each of its methods is synchronized. However, one of the methods, *transfer*, accesses more than one account object, and only synchronizes the access to one of them. This is illustrated in Listing 3.11.

Listing 3.11: *Account* test code snippet

```
1 synchronized void transfer(Account other, double amount) {
2     this.balance -= amount;
3     other.balance += amount; // RACE!
4 }
```

In order to reproduce the error in a transactional context, we encapsulated one of the accesses inside a transaction. In all the other methods, we encapsulated all of the statements in an `atomic` block. This conversion is shown in Listing 3.12.

Listing 3.12: *Account* test – TM version

```
1 void transfer(Account other, double amount) {
2     atomic {
3         this.balance -= amount;
4     }
5     other.balance += amount; // RACE!
6 }
```

Testing Procedure

The first test was performed on the original Java version by running JChord on it, without any modifications. The second test was performed by passing the converted code through our tool, which converts all atomic blocks to a global lock protected block.

Test Results and Discussion

Both tests finished in little over 2 minutes. In the Java version, there were 8 flagged races. These were disregarded as false positives, as they were conflicts with the main thread, which could not happen because the access only takes place after joining with all child threads. On the other hand, the expected races between instances of `Account` were not detected. After

testing some variations of the test subject, we assume that this is because JChord only takes into consideration the type of the object used in a synchronization, and not the instance. Therefore, either synchronizing on `this` or `other` will always cause JChord to assume that all instances are fully protected.

In the converted version, there were 20 reported races. These include the same 8 false positives of the previous test, but we now also see the expected conflicts of different instances of `Account`. This happens because the statement in line 5 is now explicitly outside of a critical region.

We have seen that our tool can be used to point out dataraces in programs that use TM. In this particular test, we have seen a peculiar example of how this procedure for TM programs can be more accurately processed by JChord than the equivalent using locks.

Conclusion: Results are corroborative. The false positives were the same. False negatives appeared in the Java version, but not in the TM version.

3.6.9 Airline

The *Airline* test subject is discussed in [IBM, RM09, DPE06]. It is about an airline company selling tickets until the airplane is full. There is one thread to sell each ticket, but because overbooking is used, there are more threads than actual seats on the plane. Each thread should detect that the plane has been filled, and immediately stop selling tickets. However, there are synchronization errors that could allow selling more tickets than available, or terminating without selling all tickets.

Test Preparation

The original Java version was obtained from [RM09]. The most important code snippet is shown in Listing 3.13.

The TM version is our own modification, shown in Listing 3.14, with all the accesses protected by transactions, and with new checks by the threads to stop the program from entering disallowed states. This actually corrects all errors, as we have not found a satisfiable equivalence of the error.

Testing Procedure

We ran JChord directly on the unmodified Java version. For the TM version, we passed it through our tool, converting transactions to mutual exclusion blocks, and then passing the result to JChord.

Test Results and Discussion

Although there are many issues about this test program, the most blatant are related to temporal synchronization. Regardless of this, there are 4 races pointed out in the original Java version,

Listing 3.13: *Airline* test code snippet, original version.

```
public Bug(int size, int cushion) {
    Num_of_tickets_issued = size;
    Maximum_Capacity = Num_of_tickets_issued - cushion;
    threadArr = new Thread[Num_of_tickets_issued];

    for (int i = 0; i < Num_of_tickets_issued; i++) {
        threadArr[i] = new Thread(this);
        if (StopSales) {
            Num_Of_Seats_Sold--;
            break;
        }
        threadArr[i].start(); // "make the sale !!!"
    }
    if (Num_Of_Seats_Sold > Maximum_Capacity)
        throw new RuntimeException("bug found");
}

public void run() {
    Num_Of_Seats_Sold++; // making the sale
    if (Num_Of_Seats_Sold > Maximum_Capacity)
        StopSales = true;
}
```

Listing 3.14: *Airline* test code snippet, TM version.

```
(...)
    for (int i = 0 ; i < Num_of_tickets_issued ; i++) {
        threadArr[i] = new Thread(this);
        atomic {
            if (StopSales) {
                Num_Of_Seats_Sold--;
                break;
            }
        }
        threadArr[i].start();  // "make the sale !!!"
    }
    for (int i = 0; i < Num_of_tickets_issued ; i++)
        threadArr[i].join();
    if (Num_Of_Seats_Sold > Maximum_Capacity)
        throw new RuntimeException("bug found");
(...)

public void run() {
    atomic {
        if (!StopSales) // Fixes bug~1
            Num_Of_Seats_Sold++; // making the sale
        if (Num_Of_Seats_Sold > Maximum_Capacity)   // checking
            StopSales = true;                      // updating
    }
}
```

between different threads accessing the same variables. The expected conflicts, between each child thread and the main thread, are not discovered. This may be because the main thread is executing in a constructor when these conflicting accesses are performed. On the other hand, as expected, there were absolutely no dataraces detected in our TM version.

We have seen that our tool did not lead to any false positives on this test, and did not flag any conflicting access. This is the expected behavior, as we believe to have corrected all bugs in this program. When disabling one bug correction, there is a conflict promptly pointed, although the most dangerous races were not found. These false negatives are inherent to JChord and are observed in the Java version as well.

Conclusion: Results are corroborative. The false negatives are equivalently observed in both versions.

3.6.10 Piper

The *Piper* test is discussed in [IBM,RM09,DPE06]. The theme is about an airplane with only two seats, and a number of passengers that must make a trip. The error is that all waiting threads simultaneously board the airplane with their passenger, when only one of them should do it at a time. The error is caused by a wrong program logic for waking threads, and can be corrected simply by changing an `if` into a `while`. A code snippet is shown in Listing 3.15 to illustrate this.

Listing 3.15: *Piper* code snippet

```
1  public synchronized void fillPlane (String passenger)
2      throws InterruptedException {
3          // BUG - should be while, not if
4      if ((last + 1) % NUM_OF_SEATS == first)
5          this.wait();
6      passengers[last] = name;  // load passenger to plane
7      last = (last + 1) % NUM_OF_SEATS;
8      this.notifyAll();
9  }
10
11 public synchronized String emptyPlane() throws InterruptedException {
12     while (first == last)
13         this.wait();
14     String name = passengers[first];  // get passenger off the plane
15     first = (first + 1) % NUM_OF_SEATS;
16     this.notifyAll();
17     return name;
18 }
```

Test Preparation

For the first test we used the version available from [RM09].

The usage of `wait()` and `notify()` methods in a transactional memory context is not standard, and it may lead to different behaviors depending on the TM implementation (or

even be forbidden). For this reason, instead of simply replacing synchronized methods with equivalent atomic blocks, we isolated the statements surrounding all *waits* and *notifies*, and put them inside shorter methods. Then, each existing method was modified to call these new methods where appropriate, and each `wait()` and `notify()` was put inside a synchronized block protected by the `this` reference. An example is in Listing 3.16

Listing 3.16: *Piper* test – TM version

```
1  public void fillPlane (String name) throws InterruptedException {
2      // is_full() is atomic
3      if (is_full()) // BUG - should be while, not if !!!
4          synchronized(this) {
5              this.wait();
6          }
7
8      load_passenger(name); // Atomic method
9
10     synchronized(this) {
11         this.notifyAll();
12     }
13 }
```

Testing Procedure

We ran JChord on the original Java version, and our own transformative tool on the TM version.

Test Results and Analysis

Each test took little under two minutes. JChord did not detect any race in either one.

These results were expected, because the error is not related to the usage of monitors. JChord seems to not have shown false negatives in either of the tests.

Conclusion: Results are corroborative. There are no races reported, in both versions.

3.6.11 Clean

Like *Piper*, the *Clean* test does not revolve around issues with monitor based synchronization, but rather on preemption based primitives, such as `wait` and `notify`. This test is discussed in [DPE06, RM09]. The theme of this test is about two threads that perform some form of computation, and synchronize with each other before entering the next cycle, as illustrated in Listing 3.17. The error is that a specific scheduling could allow a notification to be sent before the destination thread was listening for it, leaving both threads sleeping, and effectively deadlocking the application. Because this bug is not related to the use of monitors, we do not expect to find it specifically, although we may find some unprotected data accesses.

Test Preparation

The Java version was obtained from [RM09].

Just like in *Piper*, the TM version was obtained by adapting methods to jump to atomic regions, and the `wait()` and `notify()` calls were put inside `synchronized(this)` blocks.

Listing 3.17: *Clean* example code

```
// First Thread
public void run() {
    for (...) {
        event1.waitForEvent(count);
        event2.signal_event();
    }
}

(...)

// Second Thread
public void run() {
    for (...) {
        event1.signal_event();
        event2.waitForEvent(count);
    }
}
```

Testing Procedure

We ran JChord on the original Java version, and our own transformative tool on the TM version.

Test Results and Analysis

Each test took little under two minutes.

In the original Java version, four dataraces were found, related to accesses to the shared counters. Although this is not the main bug of the program, we will take this result as a basis for comparison.

Our TM version was reported as having the same four races. We ran an additional test, in which the remaining conflicting accesses are also encapsulated in a transaction. The result, as expected, is zero races found.

There were no false positives in this test, and it seems likely that there are no false negatives either.

Conclusion: Results are corroborative. False positives are the same in both versions.

3.6.12 Allocate Vector

The *Allocate Vector* test is discussed in [IBM, DPE06, RM09]. This is a very simple program containing the *Two-Stage Access* bug pattern. This bug, as the name implies, is related to failures in atomicity when the logic of a critical operation is wrongly separated into two consecutive critical code sections. Once again we find ourselves before a bug unrelated to mutual exclusion primitives. Because of this, we do not expect JChord to detect the main error, although we do

not discard the possibility of finding other subtle races. We run this test for confirmation and comparison purposes.

The theme of this test is an allocation bit vector, in which each cell is either 'A' – allocated, or 'F' – free. The problem occurs when a thread checks for a free block, and then allocates it, in two separate operations. This is illustrated in Listing 3.18

Listing 3.18: *Allocate Vector* snippet code

```
1  public void run() {
2
3      (...)
4
5      resultBuf[i] = vector.getFreeBlockIndex(); // Check for free block
6      if (resultBuf[i] != -1) {
7          vector.markAsAllocatedBlock(resultBuf[i]); // Allocate it
8      }
9
10     (...)
11
12 }
```

Test Preparation

The Java version was obtained from [IBM].

The TM version was converted by taking each synchronized method and encapsulating all statements inside an atomic block. Additionally, a new atomic method was conceived, `alloc_block`, that atomically checks and marks a block in the vector. This method can be activated or deactivated in order to test the program with or without the bug.

Testing Procedure

We ran JChord on the original Java version, and our own transformative tool on the TM version, with both the new method activated and deactivated.

Test Results and Analysis

Running JChord on the original Java version took 1 minute and 52 seconds, and identified nine dataraces. The first four of these issues were between the main thread, which read the computed values at the end, and the production methods. These are false positives, because the main thread only checks the values after all child threads have finished and joined. The remaining five conflicts are also false positives, since they could only occur if two threads were created with the same buffer object, which never happens.

The testing of our converted TM version takes 1 minute and 45 seconds, and shows similar results. When we run the version with the atomic method disabled, exactly nine races are detected as well, all of them equivalent cases of the previous results. When we activate this atomic method, three of the races disappear, since they resulted of conflicts inside the logic of this method. The remaining false positives are still observed.

This test has shown yet another case where the conversion of transactions into global locks allows to perform datarace detection. We have seen a case in which the analysis of a Java program with dataraces, and its equivalent TM version, produce the same results.

Conclusion: Results are corroborative. False positives are the same in both versions.

3.6.13 LeeTM

We shall present two experiments regarding a real and renowned concurrency benchmark. LeeTM [AKJ+08] is a benchmark for transactional memory based on the Lee algorithm for iteratively finding routes in electronic circuits. This is an interesting benchmark because each subproblem checks many shared values, which will not be needed in the end, and will be discarded. When the effects are committed, only the values that are to be changed matter, and the read-only ones can be disposed. However, this may still raise conflicts in the case of TM, and so this test is used to check how a TM system behaves in such conditions, although it is implemented using Java monitors.

Test Preparation

In the first test with LeeTM, we will simply run JChord on it, and check if it returns anything. It will be interesting to check whether anomalies will be found in such a proven and widely used program. We will perform this experiment to both check LeeTM's consistency, and compare results with our second test.

For this second test, we will convert all synchronized blocks to atomic. It happens that the source code for LeeTM has only 5 occurrences of the `synchronized` keyword, in contexts where the conversion is very straight-forward.

Results and Result Analysis

JChord takes about 10 minutes to finish analyzing the original Java version of LeeTM. The result is a surprisingly high amount of flagged races. In a total of 99 fields, in 64 objects, where some fields get as much as 500 races, there was a total of 6795 issued dataraces.

The analysis of our TM version also takes around 10 minutes. The result is slightly lower, with 6791 total races. This number is understandable, taking into consideration the result of the first test. Since both versions are very similar, the number of errors found should also be of the same order. Also, because we know that the usage of a global lock imposes a more restrictive mutual exclusion, as discussed in Sections 3.2 and 2.4.1, the number of total races in this scenario is expected to be lower or equal to the first one.

After a deeper analysis of the results, it was found that only 10 fields should really be of concern. The remaining faults were detected inside the JRE libraries. The Java version yielded a total of 52 reports, whereas the TM version resulted in 48. Out of all these potentially harmful issues, only 2 seem to be actual possible dataraces. The remaining are either false positives or benign races, such as a boolean value that is only modified once to notify threads to stop.

The difference in results between the two versions is caused by synchronized blocks using different locks. When these blocks are evaluated by JChord, a fault is reported. However, when converting to TM, all synchronized blocks were transformed into atomic sections, which in turn were automatically replace by blocks synchronized on the global lock. Hence, this *wrong-lock* issue disappeared.

Conclusion: Results are corroborative. The same dataraces were discovered in both versions. The false positives are equivalent.

3.7 Concluding Remarks

The purpose of this chapter has been to discuss the detection of low level dataraces in transactional memory programs, by applying similar existing techniques for programs that use monitors for synchronization. We have discussed theoretically the implications of this approach, and the conditions in which it would be expected to be available.

The deployment of this detection is achieved by a small tool that converts all transactions to a monitor protected critical region, synchronized on a newly generated global lock.

In order to validate this approach, we have prepared a set of tests that contain documented errors, used for testing similar error detecting tools. These test programs are designed for monitor-based programs, and so it is appropriate that we have used them to test our conversion. By verifying that the test results correspond to the expected, that the detection on monitor based programs yields the same results as the one performed on the transactional converted versions, we believe to have shown how this approach can be used to detect dataraces in TM programs.

4

High-Level Anomalies in Transactional Memory

4.1 Introduction

In Chapter 3, we have analyzed the occurrence of errors known as *dataraces*. These issues are related to bugs in source code, resulting from incorrect or inexistent usage of synchronization mechanisms. In this chapter, we will examine a different class of concurrency errors, which could be present even in the absence of dataraces. Although many variants will be explored, they will be collectively addressed here as *high-level anomalies*. In order to avoid nomenclature disorders, the dataraces described in the previous chapter shall be referred to as *low-level dataraces*.

As an example of such a high-level anomaly, consider the program in Listing 4.1, in which a bounded data structure is implemented by wrapping a `java.lang.List` object, whose size should not go beyond `MAX_SIZE`. Before the client code asks for an item to be stored in the data structure, it politely checks if there is room available in the list. All accesses to the `list` field are safely enclosed inside transactions, and therefore no low-level datarace may exist. However, before the item is delivered for storage in the list, the size of the list could already be the maximum allowed, due to interleaving with another thread running the same code. Both calls to `hasSpaceLeft()` and `store()` should have been done inside the same transaction. However, in the comfort of knowing that the methods from this library are atomic and safe, a programmer could easily fail to notice this issue. This is a typical example of an *atomicity violation*, a kind of concurrency error that is subsumed by the class of high-level anomalies.

Throughout this chapter, the issues related to high-level anomalies will be addressed. We will begin by providing a more precise definition of these anomalies, with documented causes

Listing 4.1: Example of an Atomicity Violation.

```
1  private boolean hasSpaceLeft() {
2      atomic {
3          return (this.list.size() < MAX_SIZE);
4      }
5  }
6
7  private void store(Object obj) {
8      atomic {
9          this.list.add(obj);
10     }
11 }
12
13 public void attemptToStore(Object obj) {
14   if (this.hasSpaceLeft()) {
15       // list may be full!
16     this.store(obj);
17   }
18 }
```

and consequences. To meet this purpose, we will first review previous work on high-level anomalies of different sorts, and use it to establish the context of this work.

A method for detecting high-level anomalies will also be presented. This approach will be based on static analysis of Java source code, whereas the majority of previous approaches seem to provide a runtime (dynamic) detection. To reach our goal, a new way to analyze execution threads was developed, which infers the sequences of transactions that are executed in runtime directly from the transactional statements in the code, as well as their ordering.

We will analyze programs only with respect to transactions explicitly declared. As we have seen in Section 2.1.4, however, when executing in strong isolation, concurrent accesses outside transactions will also have transactional semantics. Therefore, these operations could conceptually be involved in an anomaly as well. However, our approach disregards these operations. We assume that even in the case of using strong isolation, all concurrent accesses are explicitly put inside transactions, even though they are safe. Failure to meet this condition in strong isolation does not indicate an error. However, when in weak isolation, non-transactional concurrent accesses have a chance of originating a low-level datarace. In this chapter, we will be assuming that low-level dataraces are handled by the approach described in Chapter 3, and that our subject programs are free of those anomalies.

In Section 4.2 we first survey previous work on high-level anomalies. This will allow us to revisit some concepts that will be needed, as well as defining a starting point in our approach. With these notions comprehended, in Section 4.3 we make a clear definition of the high-level anomalies we intend to detect, with a precise list of requirements which will enable the development of a detection system. The theory behind such a detector is presented in Section 4.4, and its implementation is reviewed in Section 4.5. The developed tool is then put to practice in Section 4.6, through a series of tests and experiments, whose results will be used to assess the usefulness of this work. A summary and analysis of these results is presented in Section 4.7.

Finally, in Section 4.8, we summarize this chapter, enumerate the main conclusions, and enunciate possible future work.

4.2 Related Work

We will now review two trends of work that address anomalies that are visible even in the absence of low-level dataraces. They will be used to provide context and starting point for our definition of anomalies in Section 4.3. First, we examine *atomicity violations* based on the work by Wang and Stoller [WS03], which aims at verifying strict criteria about the parallel execution of a program. Then, we analyze the notion of *high-level dataraces* by Artho et. al [AHB03], which provides a less restrictive and arguably more precise approach, though it leaves out many important anomalies. Finally, in Section 4.2.3, we make a comparison between these two definitions, see how they correlate, take conclusions and provide hints to help us reach our definition in Section 4.3.

4.2.1 Atomicity Violations

We have seen in Chapter 3 how a program that is free of low-level dataraces executes in serializability, since its execution is equivalent to some sequential execution of the transactional or synchronized blocks. However, we have also already seen a case in which this requirement is not enough for a program to be correct, with respect to concurrency. To address this issue, Wang and Stoller [WS03] have introduced a new level of serializability.

The work presented in [WS03] does not target Transactional Memory, and therefore uses some terminology that conflicts with the one we have been using. In this context, a *transaction* is merely a sequence of *events*, performed in order by a single thread, which starts and finishes at arbitrary points. An *event* is a single data access. The authors have chosen to take methods as the boundaries for transactions, as they claim that method atomicity is a common correctness requirement in concurrent programming. This choice is irrelevant in a theoretical context, as the authors emphasize. A transaction does not by itself imply mutual exclusion. It may encapsulate one or more blocks that are synchronized through the use of locks or other mechanism. These mutually exclusive sections are taken into consideration when performing the analysis, in such a way that data accesses inside them are assumed never to happen in parallel. Therefore, we may map the *events* in [WS03] to our own transactions, and their *transactions* to the scope we intend to analyze, such as a method or the whole execution scope of a thread. This will allow us to use their conclusions in the context of Transactional Memory, and reason about high-level anomalies between mutually exclusive regions.

The analysis by Wang and Stoller [WS03] intends to assess the *atomicity* of a set of threads. The term *atomicity* also has a different meaning than the one stated in the ACID properties for transactional systems. A set of threads is atomic if all of their traces are serializable, e.g., if all possible orderings of execution of transactions are equivalent to some sequential and serial execution of all threads. If a set of threads is not serializable, then it contains an *atomicity*

violation, a concurrency defect comparable to low-level dataraces or deadlocks.

So far we have been using the concept of serializability at the level of single transactions, i.e., we have been assessing whether or not two code blocks were serializable. This property meant that the execution of a program was equivalent to some serial execution of all transactions. By using Transactional Memory, with common semantics, every set of transactions had the property of serializability. Now, however, this new notion of serializability from [WS03] introduces the possibility of non-serializability, even in the case that all accesses to shared data are transactional. We shall call *block serializability* to the property that, no matter the schedule, the final state of a program will always be one that would be obtained by a certain sequential execution of all mutually exclusive blocks or transactions. We shall call *thread serializability* to the property that, no matter the execution order of mutually exclusive blocks, the result is always one that would be achieved by a serial execution of all threads, i.e., if each thread could not run in parallel with others.

Thread atomicity may be a too restrictive requirement for concurrent programs. However, it encompasses all high-level anomalies we intend to detect. If it is known that all possible executions of a concurrent program are equivalent to one of its sequential executions, with each of its threads running to the end without interference of others, then it is disregarded as being concurrently correct. Therefore, it seems reasonable to assume that a *thread atomic* program is free of concurrency anomalies. If, on the other hand, thread atomicity may not be inferred, then there is the risk of an atomicity violation.

In [WS03], two algorithms are presented that dynamically detect these conditions. Their work serves as basis for a number of other authors, from which we provide two relevant examples. First, Flanagan and Freund [FF04] perform a dynamic check for method atomicity. Their approach is based on the reduction-based algorithm in [WS03]. This algorithm makes use of *left* and *right movers* [Lip75], which attempt to determine the equivalence of a specific ordering of execution. They have provided many critical optimizations to the original algorithm. The second example is the one from Beckman et. al [BBA08], who implemented a type system for atomicity. Although this is a static approach, it requires annotation of type information before a program may be subject to analysis.

4.2.2 High-Level Dataraces

Meeting the requirement of thread atomicity, as seen in Section 4.2.1, seems to be a too strict condition for correctness. Even by finding method atomicity, as most implementations attempt, many correct programs would be regarded as suffering from an anomaly. Artho et. al [AHB03] address this issue by employing the concept of *high-level dataraces*, a drastically different approach. A high-level datarace is an anomaly that occurs when a set of shared variables is meant to be accessed atomically, but at least one thread fails to do so.

As an example, consider again the previous application with a bounded data structure. This time a counter was added that keeps track of accesses to data, as illustrated in Listing 4.2. Each time an item is stored to (or retrieved from) the list, the counter should be incremented. A new

operation `clean()` deactivates the list and frees its resources. The counter should be reset each time the list is disabled. As we can see from the example, the `clean()` method first empties the list in one atomic block, and then handles the counter in a separate transaction. Other threads could observe the inconsistency of a counter indicating a positive number of accesses when the list is disabled.

Listing 4.2: Example of a high-level datarace

```
1  public void clean() throws IOException {
2      atomic {
3          this.list.clear();
4      }
5      this.updateCounter.reset(); // atomic operation
6  }
7
8  private void store(Object obj) throws IOException {
9      atomic {
10         this.list.add(obj);
11         this.updateCounter.increment();
12     }
13 }
```

The detection of high-level dataraces is supported by the concept of *view consistency* [AHB03]. This property is based on chains of access sets from different transactions. The *view* of a synchronized or atomic block is the set of shared variables that are accessed (read or written) in the scope of that block. The *maximal views* of a thread are those which are not totally enclosed in another view from the same thread. These maximal views are an inference of what variable sets are meant to be used atomically. If a thread makes several accesses to variables in this set, without ever accessing them in the same transaction, then there is a violation of view consistency.

To provide a practical example of these concepts, consider the set of threads shown in Figure 4.1, taken from [AHB03]. In this example, Thread 1 performs a safe access to both fields `x` and `y` of a shared object, inside the same critical region. Thread 4 makes an access solely to `x`, but it does also access both fields inside another transaction. Therefore, the first synchronized block is likely an access that does not need `y`, and this thread is regarded as safe. Notice that the views from Thread 4 are `{{x},{x,y}}` and the only view from Thread 1 is `{x,y}`. The single maximal view in both threads is also `{x,y}`.

Thread 2 only accesses `x`, which is compatible with the remaining threads. However, Thread 3 accesses both `x` and `y` in separate transactions. The two views `{{x},{y}}` are each enclosed in the maximal view of Threads 1 and 4, but they do not form a chain, i.e., `{x}` does not contain `{y}`, and vice-versa. Therefore, Thread 3 is reported as containing an anomaly, since it may violate the inferred assumption that these fields are related and should be handled together.

A program that does not contain view consistency is regarded as containing a high-level datarace. This approach may contain both false positives and false negatives.

An interesting work related to this one was developed by Praun and Gross [vG03]. They

```
//Thread~1
synchronized(c){
    access(x);
    access(y);
}
```

```
//Thread~2
synchronized(c){
    access(x);

}
```

```
//Thread~3
synchronized(c){
    access(x);
}
synchronized(c){
    access(y);

}
```

```
//Thread~4
synchronized(c){
    access(x);
}
synchronized(c){
    access(x);
    access(y);
}
```

Figure 4.1: Four threads that access shared fields. Thread 3 may cause an anomaly.

have implemented a static analysis that attempts to discover these anomalies. Although their goal is to detect atomicity violations, they follow the approach presented in [AHB03]. They employ the concept of *method consistency*, which is derived and much similar to view consistency.

In order to solve a specific class of anomalies that are not detected with view consistency, Artho et. al developed a new approach that addresses a specific problem, *stale-value errors* [AHB04]. Stale-value errors result from data privatization, which occurs when a private copy is made of a shared variable, and this copy is later used to update that same shared variable.

4.2.3 Comparison

View consistency is a different and incomparable criteria from atomicity, since none of them implies the other. The differences between the two criteria have been well addressed in previous work [WS03, AHB03]. Authors in [AHB03] claim that view consistency provides more precise results. In fact, many false anomalies that were reported with atomicity are no longer reported with view consistency. However, false negatives appear. Informally, the notions of *atomicity* and *view consistency* are related and appear similar. However, their formal definitions diverge, as well as the concrete types of issues they address. Atomicity and view consistency offer different guarantees.

We present two examples taken from [WS03] that illustrate this difference. In Listing 4.3, two threads concurrently read shared values. Because no thread updates the shared state, the outcome is invariably the same, regardless of the execution scheduling of the operations. Hence, these threads are atomic. However, they are considered view inconsistent, since Thread 2 accesses `x` and `y` in separate transactions. On the opposite scenario, consider Listing 4.4. If the transaction of Thread 1 runs between both transactions of Thread 2, it will produce an execution which is not serializable, and therefore not atomic. However, because only one single variable is accessed, this set is necessarily view consistent.

The requirement of atomicity may be too restrictive, and may lead to many false positives. View consistency may be seen as a compromise, reducing false positives but generating false

Listing 4.3: A set of transactions that are atomic, but view inconsistent.

```
//Thread~1
    atomic {
        read(x);
        read(y);
    }


//Thread~2
    atomic {
        read(x);
    }
    atomic {
        read(y);
    }
```

Listing 4.4: A set of transactions that are view consistent, but not atomic.

```
1  //Thread~1
2      atomic {
3          write(x);
4      }
5
6
7
8  //Thread~2
9      atomic {
10         read(x);
11     }
12     atomic {
13         write(x);
14     }
```

negatives. However, it should be emphasized the added usefulness of approaches that simply yield false positives *or* false negatives. In the former situation, a user has to discern which reports match real problems, however knowing that all issues, if any exist, have been detected. In the later case, the user will not be sure that all anomalies have been detected, but will be sure that all alarms correspond to real errors. If none of these guarantees is provided, then the usefulness of any approach is highly reduced, since a report may contain only false positives, and in addition may fail to list a real anomaly.

The notion of atomicity is expressed taking into consideration possible execution orderings and schedulings. View consistency, on the contrary, analyzes only data sets, independently of executions. Finally, it is worth noticing that the approach of view consistency works by inference, so at least one correct usage is required in order to detect any inconsistencies at all.

4.3 Definition of Anomalies

Atomicity checking seems to encompass all possible concurrency anomalies. Checking that the execution of a program is necessarily equivalent to some sequential execution of all its threads is a strong guarantee, that facilitates the understanding of programs by reducing any analysis to its serial equivalent. However, atomicity is excessively strict. The false positive rate observed when atomicity is required confirms this discrepancy.

Because of this strictness, we claim that a more relaxed approach should be enough to detect most common high-level anomalies. Instead of pursuing thread atomicity, we intend that atomicity is established only among consecutive transactions. This is justified by the intuition that most errors will come from two consecutive atomic segments in the same thread, which should be merged into a single one. More specifically, the anomalies that are to be detected happen when a thread T1 executes transactions A and B, without running any other transactions between the two, and another thread T2 may execute a parallel transaction C between the runs of A and B, such that the resulting scheduling of the three transactions is not serializable.

All examples we have seen so far fit this scenario.

In order to facilitate reasoning about these conditions, we shall first list all possible scenarios that fit the previous description, in a similar way to the listing of single-variable unserializable patterns presented in [WS03]. We will then compare this criteria to full thread-atomicity, and then review the patterns most likely associated with anomalies.

4.3.1 All High-Level Anomaly Patterns

Read—Read Two consecutive transactions of one thread read shared values `a` and `b`, while other threads could have changed both values between the readings. An example is presented in Figure 4.2

Read—Write A thread performs a transaction that reads `a`. The same thread subsequently executes a transaction which writes `b`. Between these two, other threads could have written `a`, and written or read `b`. An example is presented in Figure 4.3

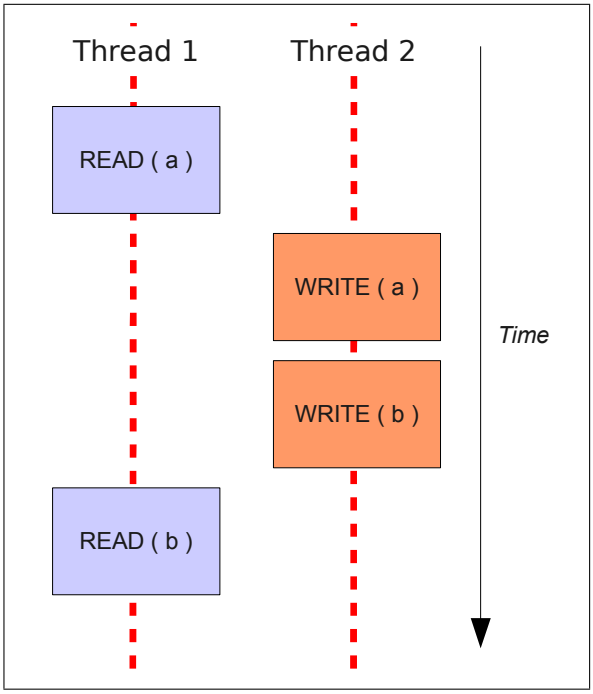

Figure 4.2: Example of a Read–Read anomaly

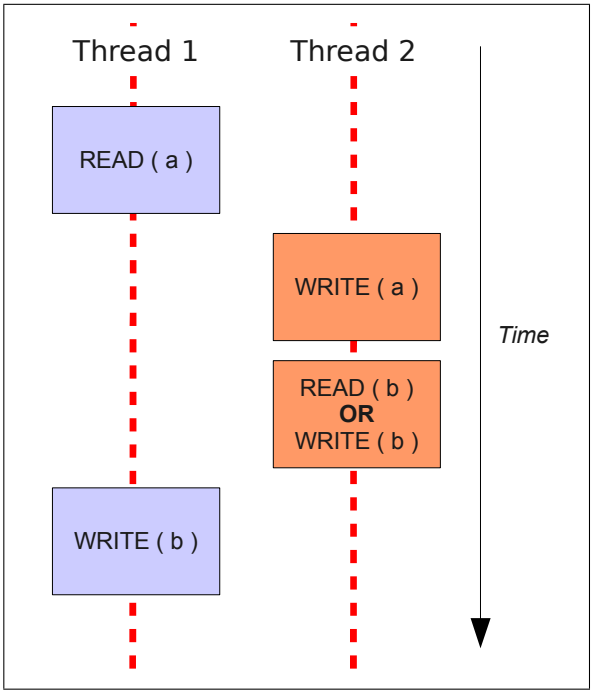

Figure 4.3: Example of a Read–Write anomaly

Write—Read A thread performs a transaction that writes `a`, and a subsequent transaction which reads `b`. Between these two, other threads could have written or read `a`, and written `b`. An example is presented in Figure 4.4

Write—Write A thread performs a transaction that writes `a`, and a subsequent transaction which writes `b`. Between these two, other threads could have written or read `a`, and written

or read `b`. An example is presented in Figure 4.5. Notice that if `a` and `b` are the same, then a reading of `a` by another thread is both sufficient and required to trigger an anomaly. A writting of `a` by the second thread does not change the atomicity of the threads.

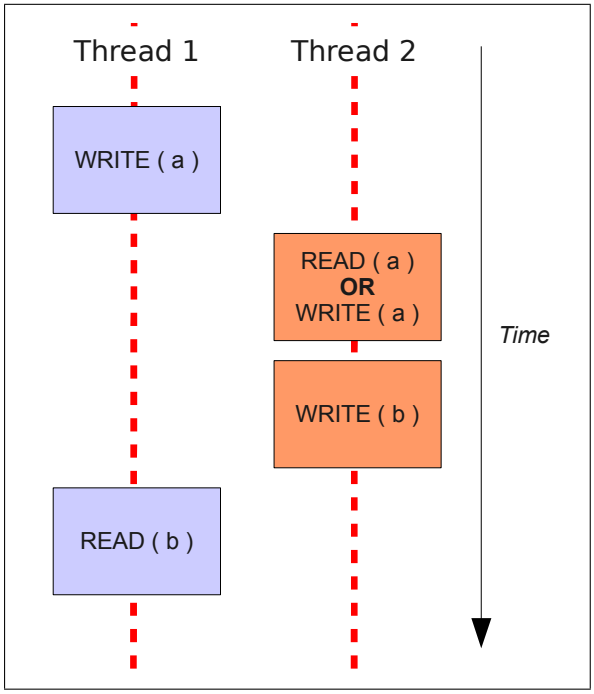

Figure 4.4: Example of a Write–Read anomaly

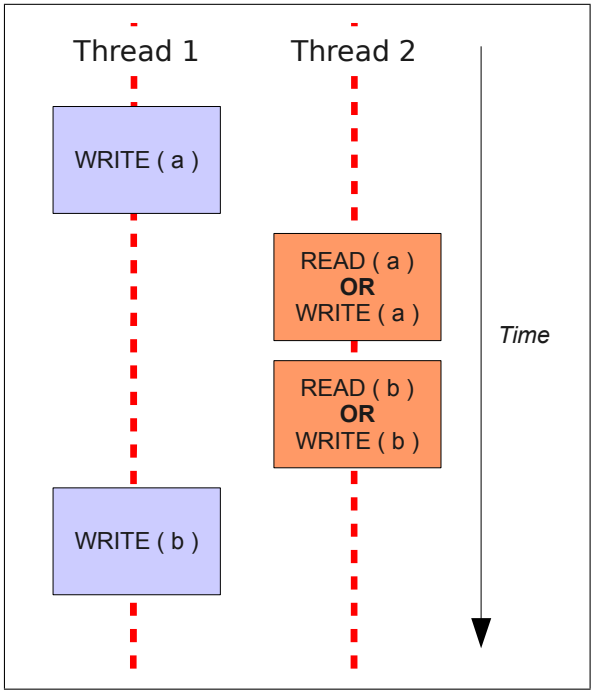

Figure 4.5: Example of a Write–Write anomaly

4.3.2 Difference to Thread-Atomicity

Consider the example in Figure 4.6, which states an important scenario in which thread atomicity and our definition originate different results.

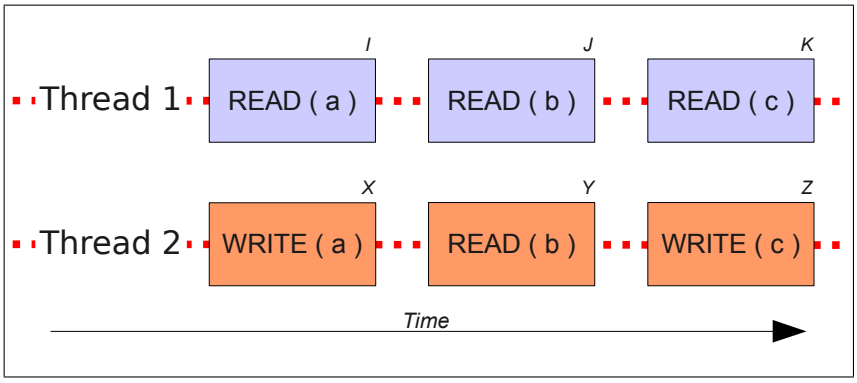

Figure 4.6: Example of a set of threads that are not atomic, but do not present anomalies on any pair of consecutive transactions.

This execution is not globally serializable (not atomic), because the writings of `a` and `c` by Thread 2 could cause Thread 1 to observe two different global states. The set of transactions

$\{I, J, K\}$ is not serializable with $\{X, Y, Z\}$.

However, this scenario does not present an anomaly on any pair of consecutive transactions. Specifically, each consecutive pair of transactions in Thread 1, $\langle I, J\rangle$ and $\langle J, K\rangle$, is serializable with each of the consecutive pairs in Thread 2, $\langle X, Y\rangle$ and $\langle Y, Z\rangle$. Between each two readings of Thread 1, it would be necessary that Thread 2 wrote `b` in order to trigger an anomaly. Similarly, between each two transactions of Thread 2, Thread 1 would have to write `b`. Since this is not the case, this execution is free of anomalies according to our patterns.

4.3.3 Common Patterns

We anticipate that the full detection of all these patterns will still yield many false positives, much like thread serializability would. Furthermore, it is intuitive that many of these patterns will seldom or never be involved in an anomaly. Such is the case of the anomaly illustrated in Figure 4.7. If the first thread writes `a` and then reads it, it does not matter that some other thread may have changed the value. In a strict serializability environment, such as a database, this would be an inconsistency, since one client would be observing changes from another one, thus violating isolation. However, in the context of Transactional Memory, it does not seem common that a programmer would incorrectly assume isolation in this kind of scenario. Since the second transaction of Thread 1 is reading the shared value, it appears that it is not making any assumption about the value that the same thread has written previously.

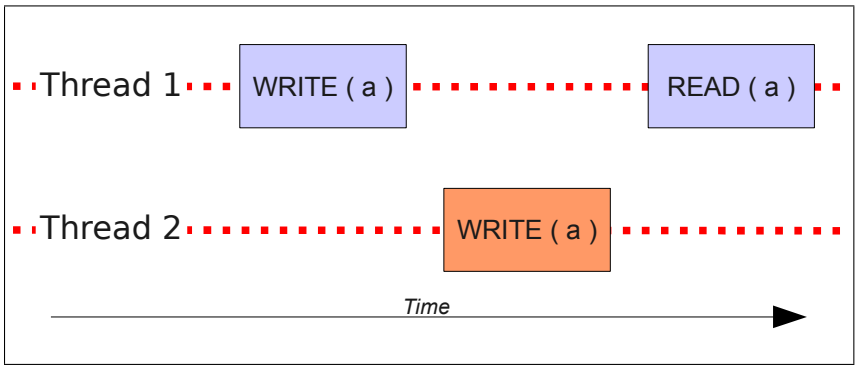

Figure 4.7: An unserializable pattern which does not appear to be anomalous.

Therefore, it seems intuitive that some of these patterns should be discarded when performing an analysis, in order to produce a more useful result. For some patterns, it may not be totally clear whether or not they should be flagged as anomalous. Therefore, we propose that the most intuitively anomalous patterns be addressed, and an approach is developed that allows to interactively enable or disable the reporting of each pattern in the analysis tool.

After analyzing all of the possibly anomalous patterns, three of them intuitively seem to correspond to anomalies, and are presented next.

Non-atomic global read (Read–write–Read or *RwR*) A thread reads a global state in two or more separate transactions. Therefore, it could make assumptions based on a state that

is composed of parts of different global states. The resulting global state could even be inconsistent. An example is presented in Figure 4.8.

```
// Thread~1
atomic { read (x); }
atomic { read (y); }
if (x == y) {
    ...
}
```

```
// Thread~2
atomic {
    write (x);
    write (y);
}
```

Figure 4.8: Example of an RwR anomaly.

Non-atomic global write (Write–read–Write or WrW) The opposite scenario. A thread changes the global shared state, but it breaks the update into several transactions. Other threads could attempt to read the global state and observe the inconsistency resulting from observing several partial states. An example is presented in Figure 4.9.

```
// Thread~1
atomic { write (x = x + 1); }
atomic { write (y = y + 1); }
```

```
// Thread~2
atomic {
    read (x);
    read (y);
    assert (x == y);
}
```

Figure 4.9: Example of a WrW anomaly. The condition of x being the same as y is part of the program invariant.

Non-atomic compare-and-swap (Read–write–Write or RwW) A thread checks a value and then, based on that previously observed state, alters the variable. The value could have changed in between, so the alteration would possibly not make sense anymore. An example is presented in Figure 4.10.

```
// Thread~1
int a;
atomic { a = read (x); }
a = a + 1;
atomic { write (x = a); }
```

```
// Thread~2
atomic { write (x = 10); }
```

Figure 4.10: Example of an RwW anomaly.

These three patterns will be taken as hypothesis for TM high-level anomalies. Our real target is the occurrence of two consecutive transactions that are logically related and should be run as a single transaction. For this, all possible scenarios described previously may be considered. However, in order to refine the anticipated lack of precision of results, it is proposed that these three common patterns be taken to verify whether or not they fit the typical anomaly scenarios. Throughout Sections 4.4 and 4.5 a static approach will be presented that tests these patterns for transactions that should be merged. This tool may be configured on a per-case

basis to consider each pattern an anomaly or a benign pattern. The tool will be tested in Section 4.6 with these three common patterns in order to assess the usefulness of both the tool and the patterns.

4.4 Anomaly Detection Approach

In order to assess the atomicity of a program, it is necessary to know the set of transactions that are executed, as well as the set of variables that are read and written by each transaction. However, it is not possible to have this information before running the program, since control flow may change while the transactions are being executed, changing the number of times each transaction is carried out and what data is accessed inside each one.

Our approach will perform a sort of symbolic execution, starting from the bootstrap point of each thread, and conservatively assume that every transactional block is executed at its point. Similarly, we will assume that inside each transaction every read and write operation that is stated in the source code will be performed.

The idea is to obtain a set of conservative execution traces, that represents all possible executions of this thread. Each trace will be a sequence of atomic blocks which are executed in the respective trace. This will let us have all pairs of consecutive transactions that may be carried out, as well as the point in code where they will be performed.

These traces conservatively contain all atomic blocks that may possibly be executed, and covers all anomalies previously defined that may occur at runtime. By analyzing this data, one may determine anomalies that may possibly occur at runtime. For example, if it is determined that two subsequent transactions from one thread may have another thread interfere between them, and the data access pattern matches one of the problematic cases previously identified, then we may report an anomaly.

When implementing this approach, in Section 4.5, each set of traces of the same thread will be handled as a single one, in the form of a tree. This will allow us to use a single object to represent all traces of a thread, and ease the verification of anomalies between two concurrent threads. To this end, each tree will have special nodes that represent disjunctions in the execution, for points where control-flow could take one of multiple alternative paths.

Since we are only concerned with detecting concurrency anomalies in Transactional Memory, we must only trace transactional accesses to shared data. All other statements may be discarded from the trace, including statements in transactions that access only local data, and non-transactional accesses to shared variables. Notice that these non-transactional operations may be the source of low-level dataraces. We assume that this issue is handled by the approach described in Chapter 3.

Having set the guidelines of our method, in the remainder of this section we will discuss in further detail the relevant aspects that must be considered when defining the approach.

4.4.1 Transaction Nesting

In the case of atomic blocks containing sub-transactions in their code, both the top-level transaction and its sub-transactions will be handled as a single one. It should be taken into consideration that some Transactional Memory systems actually provide different semantics for nested blocks as seen in Section 2.1.4. For now, however, we will take this more simplistic approach by assuming flat nesting. Therefore, the lower level atomic blocks can be discarded. More precisely, if we trace an atomic block while already in a transactional context, this block will be replaced by its sub-statements.

4.4.2 Method Calls

When tracing the execution method of a thread, whenever a call statement is reached, it is replaced with the statements inside the target method, in a process which will be called *expansion*. If in turn this method calls other methods, then the process is repeated. An example of expansion is presented in Figure 4.11. Expansion enables the viewing of all sequences of transactions performed by a thread, as though the execution was composed of one single method.

Care must be taken in order to avoid infinite expansions, such as in cases when two methods simultaneously call each other. In this case, the expansion should stop before the third time that a method is in the call stack. If it stops at the second time, as would be intuitive, then some execution scenarios might not be covered. Considering again the example on Figure 4.11, if method `B()` had not been expanded a second time, then the tracer would not see a possible anomaly resulting from `Transaction2()` being followed by `Transaction1()`.

```
1  void A() {
2      Transaction1();
3      B();
4  }
5
6  void B() {
7      Transaction2();
8      if(...)
9          A();
10 }
11
12 void Main() {
13     A();
14 }
```

```
1  void Main_Expanded () {
2      Transaction1(); // call A
3      Transaction2(); // call B
4      if(...) {
5          Transaction1(); // call A'
6          Transaction2(); // call B'
7          if(...) {
8              // do not expand A again
9          }
10             // return B'
11         // return A'
12     }
13     // return A
14 }
```

Figure 4.11: Method expansion. Care must be taken in aborting repeated expansions.

4.4.3 Alternative Execution Statements

We will now consider *disjunction* points in a program. Disjunctions are control structures such as an *if* statement, or a *switch/case* statement. They provide two or more alternative paths of execution, from which only one should be taken, after which execution should again be the same, independently of the previous choice. This looks much like a fork and join. Disjunction

nodes bring the problem of not knowing in advance which branch will be executed. However, we know that at least and at most one of them will be executed, and at which point, relatively to the previous and following nodes in the trace. Therefore, we can have a special node in the trace that is not a transaction, but rather a disjunction, a set of sequences of transactions, from which one will be taken, never more than one. This allows to expose anomalies with nodes preceding and following the disjunction, while not raising anomalies between different branches.

4.4.4 Loop Statements

Loops are also typically available in most languages, and in many variants. All of them hold a code block that should or could be executed, a number of times that is frequently undetermined at compile time. Atomicity approaches typically take into consideration the interactions that two consecutive executions of the inner block may have with itself. However, in our case, anomalies between the end of a loop and the beginning of its next iteration seem rather unintuitive. In any case, if this happens, there could never be any anomaly that is not covered by the representation of two consecutive executions of the inner block. Therefore, a loop statement will cause the addition of two nodes to the trace, each representing an iteration of the inner block of the loop.

4.4.5 Other Control-Flow Structures

Other control structures should be evaluated on a per-case basis, but most times they may be reduced to an equivalent case of one of the previous. If we take chained *if*'s, or *else-if* statements, they may be replaced with a proper set of disjunction nodes. Exception throwing and handling statements, such as *try-catch*, may be replaced by a sequence of disjunctions, like a code block whose execution could stop between each pair of statements. The same method is applicable to loops that end prematurely, such as by the use of a *break* statement or similar.

4.4.6 Discussion of the Approach

In order to obtain a useful perception of the running of the program, three other subjects must be considered. The data presented next is not strictly necessary to analyze the program, but will highly boost its usefulness. In case that this data not available, each of these three may default to a conservative evaluation. In that case the precision of the analysis will suffer, raising the chance of false positives, though false negatives will not.

Static Threads Analysis

First, the instantiation of threads at runtime. Statically, one may only analyze the classes corresponding to threads that may be run. In Java, this is either a class with a main method, which spawns a main thread from the JVM, or a class that is a subtype from either `java.lang.Runnable`

or `java.lang.Thread`. This code is not an indication of which threads may actually be running during execution, or the number of each type of thread. This is especially important in cases when a thread class may lead to two instances that interfere with each other. If this analysis is not available, then it may be assumed that at least two instances of each thread type are running.

Happens-in-Parallel Analysis

Then, beside the number and type of threads, it is important to know the time at which each of them will perform which operation. To determine that an operation from a thread may interfere between two transactions from another one, it is imperative to guarantee that this interleaving is actually possible at runtime. If a thread is guaranteed to start only after another one has finished, then there is no way there can be an anomaly between them. The use of *start-join* idioms, as well as of *wait-and-notify*, may alter the control flow in such a way that makes it impossible for certain executions to happen in ways that would raise anomalies. If this analysis is not available, then it may be assumed that there are no guarantees as to the time at which threads and transactions may run, i.e., each transaction from a thread may occur between any other two transactions from another thread.

Points-to Analysis

Finally, in order to report an anomaly, it is necessary to determine which data is being accessed by each transaction. If all transactions involved in a possible anomaly access completely disjoint sets of data, then there is no possible anomaly. Knowing that only fields of shared objects, and cells of shared arrays may be subject of an anomaly, we may discard accesses to local variables. In order to determine an anomaly, it is necessary to know that transactions are accessing the same object, and for this, some pointer analysis would be required. This would also help the static threads analysis and happens-in-parallel analysis, knowing if two thread objects are the same. If this analysis is not available, it may be assumed that all accesses to a field of objects of the same class, is made to the same object, i.e., that there is only one instance of each type, and that it is shared between all threads. Similarly, it may be assumed that all thread references point to the same thread object.

The correctness of this analysis is subject to the correctness of the criteria determined for anomalies, i.e., the three common anomalies seen in Section 4.3.3 (*RwR*, *WrW* and *RwW*). There will be no false negatives in this analysis, with respect to these conditions; as we have seen, any possible occurrence of one of these patterns will be detected. Any real anomaly that is not detected, or any alarm that does not correspond to a real anomaly, should be handled by refining these conditions.

4.5 Implementation

In order to implement the approach described in Section 4.4, a static analysis framework is necessary. Our choice was the Polyglot framework [NCM03] for extensions to the Java language. We used a previously available language extension [Dia09], which was addressed in Section 3.5, that already recognizes atomic blocks and makes them available for further processing in the form of an AST.

Having a program representation in the form of an Abstract Syntax Tree (AST), including nodes for atomic blocks, the trace information may be extracted from it. Remember that these traces are themselves trees, rather then a simple sequence of transactions, because of disjunction nodes. The original AST is traversed iteratively, until we have our own trace tree.

4.5.1 Trace Tree Structure

An AST generated by polyglot contains a high amount of control information which is unnecessary for our analysis. On the other hand, creating AST nodes and performing transformations in agreement with the Polyglot API is quiet complex, and requires compliance with procedures which are not oriented for this kind of analysis. Therefore, we have designed our own tree class, with specific tree nodes. Our program representation features five main kinds of nodes:

Access nodes represent an access to a field of a shared object. There are subtypes of nodes for read access, write access, and multiple accesses of different types performed by the same statement.

Atom nodes represent an atomic block, a transaction.

Call nodes represent a call to a method, which shall be replaced by the statements of that method.

If nodes are used for binary disjunctions, a fork in the code with two possible branches of execution. The alternative branch may be empty, for cases when a block may or may not be executed.

Sequence nodes represent a sequence of statements. Generally, the body of *if* nodes and method call nodes will be sequences, unless they consist of a single instruction.

These node types suffice to perform our analysis. It should be considered that there is no need for loop-representing nodes, since an occurrence of a loop node in the original program's AST will lead to the inclusion of two iterations of the body of the loop. The loop node may be discarded.

4.5.2 First Phase — Method Listing

The first pass traverses the original program AST and processes it, generating an equivalent representation using the new nodes. We have implemented a visitor pass that is in accordance

with the polyglot framework. This visitor registers the occurrence of method declarations and processes the body of each method. Whenever the body of a method is parsed, the visitor starts tracing events in a new trace tree. Access nodes, atom nodes, call nodes, if nodes and sequence nodes are used to represent the occurrence of analogous operations in the original code of the method. When the end of the method body is reached, this trace is put in a registry of methods, mapping the signature of each parsed method to the respective trace tree. This mapping will be used in the next phase. After this first pass, the relevant data is available in instances of our own classes.

4.5.3 Second Phase — Expansion and Merging

Each thread starting method is traversed, and every method call inside it is replaced by the tree of that method, in an iterative fashion. A sort of call stack and a counter keep track of the methods that are being expanded at a time, in order to avoid infinite expansions. This pass also merges nested atomic blocks into a single one. Variable accesses outside atomic blocks are discarded. We now have a full trace tree for each thread start method, free of method calls. Since all calls have been replaced by the body of the target method, each trace contains all the accesses that a thread may perform. At this point, the data is gathered in a form that is suitable for analysis.

4.5.4 Third Phase — Analysis

We have implemented an analyzer class which encapsulates the logic of the analysis. An instance of this analyzer receives pairs of traces that run in parallel, and returns a list of the detected anomalies, if any. The analyzer iterates the transactions of one of the traces. Between each pair of these transactions, the analyzer checks the possible parallel transactions performed by the second trace. The analysis of a disjunction node originates the checking of two different pairs, consisting of the first node of each branch, and the node that immediately precedes the disjunction node. If an anomalous pattern matches the analyzed transactions, then an anomaly report is added to the list.

4.5.5 Static Analysis

Neither static-threads, happens-in-parallel, nor points-to analysis are performed at the moment. For now, this approach conservatively assumes that exactly two instances of each thread type are being executed in parallel, and that all transactions may be performed at an unknown time. Finally, only the object types and field names are compared in order to determine a simultaneous access; the analyzer assumes that all accesses are performed on the same object. This raises the maximum possible amount of conflicts, although it reports a lot of false positives.

4.5.6 Standard or Unavailable Methods

Calls to standard methods of the Java Runtime Environment (JRE) library will also generate call nodes. However, when method expansion is performed, there will be no trace tree associated with these methods, because their body was not scanned.

Because we do not intend to analyze the whole specification of the JRE, calls to such methods are ignored. The same approach covers the use of third-party libraries whose code is not supplied. Whenever such calls are found, they are discarded as if no accesses to shared variables could be performed.

Analyzing all Java standard methods would be costly, and likely to provide little, if any, advantage. Furthermore, the source code of the JRE is frequently not available. Analyzing methods from inside the JRE would require analyzing compiled methods in bytecode format, which is out of the scope of our work. An alternative would be to allow the user to explicitly specify what operations are performed by each method in the JRE or in an external library.

4.6 Validation of the Approach

In this section we will analyze the behavior of the tool described in Sections 4.4 and 4.5. To meet this purpose, we will run a series of tests, composed of simple programs that simulate real-world scenarios and show peculiar concurrency anomalies. Each test contains a number of known anomalies, which our tool should ideally detect. The tests presented here have taken from the literature, and have been used to validate previous works on detection of concurrency anomalies [BBA08, AHB03, AHB04, FF04, vG03, IBM]. In their original descriptions, each test is typically only abstractly described, showing only a portion of the program code, or pseudocode. Therefore, these test programs have been instantiated into full programs in order to be used in the tests. During this implementation, some decisions had to be made due to some aspects that are not specified in detail, or not at all. Two additional tests were specifically developed for testing our tool.

Each of the following subsections describe a specific test case. For each test case, code snippets are provided to help illustrate the idea behind the logic of the program. Additionally, a *Subject* paragraph briefly describes the test. The paragraph for *Documented Anomalies* lists the previously known anomalies present in each test, which should ideally be reported during our validation. An optional *Other Anomalies* paragraph lists additional anomalies that may not be documented in the original description of the test, or that may have been introduced due to implementation details. The *Expected Results* paragraph shows a prediction of the results, considering the anomalies present in the program, the anomalous patterns, and the expected behavior of our application. The *Results* paragraph provides the output of the test, followed by a discussion and analysis of the observed results. In Section 4.8 the totality of these results will be taken in consideration when concluding the chapter.

4.6.1 Test: Connection

Subject: An example adapted from [BBA08], illustrating a network chat application. An object represents a network connection, and a Graphical User Interface (GUI) client uses services from this connection. The most important parts of the code are presented in Listing 4.5.

While implementing this example, some code which was not provided in the original paper [BBA08] had to be created. The implementations of the `Counter` class, the `connect()` method of class `Connection`, and the `main` method on `GUI` class which starts ten threads, have been implemented specifically to test our tool. Additionally, the `GUI` class now extends `java.lang.Thread`, with the `run()` method randomly choosing to send a message or to disconnect.

Listing 4.5: Code snippet for the Connection test.

```
1  // Connection Class
2  final Counter counter;
3  java.net.Socket socket;
4
5  boolean isConnected() {
6      atomic { return (socket != null); }
7  }
8
9  void send(String msg) {
10     atomic {
11         socket.write(msg);
12         counter.increment();
13     }
14 }
15
16 void disconnect() {
17     atomic {
18         socket.close();
19         socket = null;
20     }
21     counter.reset();  // socket could have changed!
22 }
23
24 // GUI Class
25 void trySendMsg(String msg) {
26     if (connection.isConnected())
27         connection.send(msg);  // connection could have closed!
28 }
```

Documented Anomalies:

1. The method `GUI.trySendMessage()` calls two atomic methods, `isConnected()` on line 26, and `send()` on line 27. The socket could have closed before the message is sent.

2. The method `Connection.disconnect()` on line 16 handles the socket atomically, but then resets `counter` outside the transaction, in line 21. The counter could be reset at a wrong time (e.g, after connection has been reestablished).

Expected Results: Anomaly 1 should not be detected (false negative), because the `socket.write(msg)` call on line 11 is not perceived as a writing operation. Because this is a standard method from the JRE, and its code is not intended to be evaluated, it is skipped from the analysis. However, the dereferencing of `socket` yields a reading operation, and therefore, an *RwR* anomaly is registered in place of the correct *RwW* anomaly.

Anomaly 2 should be detected without problems. It is a *WrW* anomaly, since two related variables are modified in separate transactions. Method `disconnect()` sets `socket` to `null` and the `counter` to zero.

Results: The behavior for anomaly 1 is as expected: the correct anomaly is not detected, and instead a different one is reported in its place. We will consider this as the occurrence of a false negative followed by a false positive. Anomaly 2 is correctly detected in place. Note that if we remove statement `socket = null` on line 19, no anomalies are reported, since there is no writing of fields. However, there is still an anomaly. In this case, the only way to detect it would be to try to give some semantic to method calls. For example, instead of disregarding methods with unavailable source, we could assume that these methods somehow internally change the calling object. This would also allow us to correctly detect anomaly 1. Notice that the observed false negative does not result from inappropriate patterns, but rather from variable accesses which are not perceived by the analysis, since they are performed from inside the JRE.

Results Summary: One false negative. One false positive. One correctly detected anomaly.

4.6.2 Test: Coordinates'03

Subject: Running example from [AHB03]. A coordinate pair managed by a single shared object. Parallel threads read and write the pair as a whole, or accessing each variable in separate. The pseudo-code for this test is in Figure 4.12.

```
// Thread t1
atomic {
    write x;
    write y;
}
```

```
// Thread t2
atomic {
    read x;
}
```

```
// Thread t3
atomic {
    read x;
}
atomic {
    read y;
}
```

```
// Thread t4
atomic {
    read x;
}
atomic {
    read x;
    read y;
}
```

Figure 4.12: Pseudo-code for Coordinates'03 test.

Documented Anomalies: Thread 3 is unsafe: the readings of `x` and `y` are performed separately. The pair could have changed between these accesses.

Other Anomalies: It is arguable whether or not the remaining threads incur in anomalies. A more precise semantical specification would be required. From the remaining threads, Thread 4

is the other one with multiple transactions. State could have changed between them, but it does not seem to harm execution. The original paper [AHB03] considers Thread 4 to be safe. Even though a more specific context would be needed to determine whether or not this is an anomaly, we will consider that Thread 4 is safe.

Expected Results: Both anomalies in Threads 3 and 4 should be detected.

Results: The anomaly in Thread 3 has been detected. The arguable anomaly in Thread 4 has also been detected, which we shall consider a false positive. There are two additional false positives with a local variable in Thread 4, which may only be solved with *points-to* analysis.

Results Summary: One anomaly correctly detected. Three false positives.

4.6.3 Test: Local Variable

Subject: This example was taken from [AHB03]. In a very simple program, a thread makes a local copy of a shared variable, modifies the copy, and then updates the shared variable. The major part of the code is in Listing 4.6. This is a specially interesting example because the original paper [AHB03] provides it as an example of an anomaly they can not detect.

Listing 4.6: Code snippet for Local Variable test.

```
1  public void run() {
2      int tmp;
3      atomic {
4          tmp = x.getValue();
5      }
6      tmp++;
7      atomic {
8          x.setValue(tmp);
9      }
10 }
```

Documented Anomalies: There could be a concurrent access that modifies the variable, resulting in an anomaly such as a lost update.

Expected Results: Clean detection of this anomaly. This scenario seems to fit the exact pattern of an *RwW* anomaly.

Results: More results than expected. The present *RwW* anomaly was correctly reported. Additionally, however, there was another report for an *RwR* anomaly, involving field `x` and subfield `x.n`. This is clearly an undesired report, as these accesses should be considered equivalent. We will consider this a false positive. This case could possibly be solved by regarding both accesses as if they targeted the same variable.

Results Summary: One anomaly correctly identified. One false positive.

4.6.4 Test: NASA

Subject: This is the NASA Remote Agent example from [AHB03]. It addresses a real flaw in the Remote Agent spacecraft controller, previously discovered through the application of a model checker.

The system state of a spacecraft is a table of variables, each containing the value read by a specific sensor. Multiple threads, called tasks, execute on board the spacecraft, attempting top achieve a specific goal. An executing task may require that certain properties of the system state hold for the duration of the task. Once the property has been achieved, the task marks the *achieved* flag of the property to true. A daemon thread monitors the execution, waking up at regular intervals to verify system invariants. The problematic points of the program are shown in Listing 4.7.

Listing 4.7: Code snippet for NASA test.

```
// Task
atomic {
    table[N].value = v;
}
    /* achieve property */
atomic {
    table[N].achieved = true;
}

// Daemon
while (true) {
    atomic {
        if (table[N].achieved && system_state[N] != table[N].value)
            issueWarning();
    }
}
```

Documented Anomalies: Suppose that a task has just achieved the property, and is about to set flag *achieved* to true. Suppose that, in the mean time, the property is destroyed by some unexpected exterior event. Suppose also that the daemon wakes up and performs checks. It verifies that the property does not hold, and that the achieved flag is false (hence, there is no anomaly). If the task now resumes and sets the flag to true, a property will be marked as achieved when it is not. The daemon has missed a possibly critical anomaly.

Expected Results: Our approach should properly report the *WrW* anomaly. A task thread modifies two fields of a cell in the `table` array, in separate transactions. The daemon thread may be scheduled to read both values between the writes.

Results: Exactly as expected.

Results Summary: One anomaly correctly detected.

4.6.5 Test: Coordinates'04

Subject: A simple test taken from [AHB04], which resembles the previous Coordinates test. The Coordinates object this time has two additional operations: `swap()` to exchange the values of both fields, and `reset()` which resets both coordinates to the default value. The implementation of these two operations is shown in Listing 4.8.

Listing 4.8: Code snippet for Coordinates'04 test.

```
1  public void swap() {
2      int oldX;
3      atomic {
4          oldX = coord.x;
5          coord.x = coord.y; // swap X
6          coord.y = oldX; // swap Y
7      }
8  }
9
10 public void reset() {
11     atomic {
12         coord.x = 0;
13     } // inconsistent state (0, y)
14     atomic {
15         coord.y = 0;
16     }
17 }
```

Documented Anomalies: As the code shows, the `reset()` operation is not atomic, and writes the coordinates in separate transactions. If a thread is scheduled to run `swap()` while another thread is in between both transactions of `reset()`, it would be swapping inconsistent values, therefore resulting in an inconsistent state even after `reset()` has finished.

Expected Results: The anomaly should be reported correctly as a *WrW*. Additionally, more anomalies could be reported, since we have `swap()` followed by `reset()`, accessing the same fields. These would be false positives.

Results: The documented anomaly was correctly detected. Additionally, 3 false positives were reported. They derive from the fact that `swap()` reads and writes all variables, and `swap()` writes `x`. According to our patterns, there is the chance that these operations are somehow related, and so warnings are issued.

Results Summary: One anomaly correctly detected. Three false positives.

4.6.6 Test: Buffer

Subject: Sample from [AHB04]. A buffer is shared between multiple threads, which remove and add elements to the buffer. Because it is assumed that each element is not added to the buffer more than once, only removal and addition operations must be atomic. The code for this test is shown in Listing 4.9.

Listing 4.9: Code snippet for the Buffer test.

```
int value, fdata;
while (true) {
    atomic {
        value = buffer.next();
    }
    fdata = f(value); // long computation
    atomic {
        buffer.add(fdata);
    }
}
```

Documented Anomalies: There are no documented anomalies. This program was analyzed in [AHB04] as an example of a correct program which leads to false positives. Because both transactional blocks access the buffer, this seems to be an atomicity violation. The buffer access protocol seems to ensure that the retained data remains thread-local. However, it must be enforced that no element is present in the buffer more than once. It is not clear whether this enforcement would be easy, or if it would result in the elimination of the anomaly report. A deeper detail would be needed, perhaps with a real case, in order to fully assess this test case.

Other Anomalies: This program was created as an example of a correct program that raises anomalies in most approaches. Our detector also raises false positives, because much semantic information would be needed to avoid it.

Expected Results: The most natural anomaly to detect would be RwW, as the buffer is read and then updated. The other false positives will be less intuitive. We expect WrW and RwR because of fields head and tail.

Results There were 7 anomaly reports. Because there are no documented anomalies, we will consider these false positives. However, a more concrete instantiation of this example would be required to precisely evaluate the behavior of our approach. The data structure used in this program is not a complete buffer, but rather a wrapper class containing multiple fields read and written in order to simulate the behavior of a buffer.

One of the incorrectly reported anomalies is the expected *RwW*. Other 2 resulted from redundant accesses, much like what happened in the Local Variable test in Section 4.6.3. One of these was an *RwR* anomaly reported, involving fields `buffer` and `buffer.head`. The other one was also an *RwR*, with fields `buffer` and `buffer.cell`.

Results Summary: Seven false positives.

4.6.7 Test: Double Check

Subject Sample from [AHB04]. Another correct program which is prone to result in false positives. The main part of the code is in Listing 4.10. The program performs a long computation on a shared value. In order to avoid synchronization overheads and provide efficiency, the value is read on one transaction, updated, and then stored again in another transaction. Before storing, the thread checks that the old value is still the same, and discards the result if it has been changed. Therefore, this program is safe.

Listing 4.10: Code snippet for the Double Check test.

```
1  int value, fdata;
2  boolean done = false;
3
4  while (!done) {
5      atomic {
6          value = shared.field;
7      }
8      fdata = f(value); // long computation
9      atomic {
10         if (value == shared.field) {
11             shared.field = fdata;
12             done = true;
13         }
14     }
15 }
```

Documented Anomalies: This test was also created as an example of a correct program that will raise an anomaly, which should be an *RwW*. However, it is a false positive, since the thread verifies the safety conditions before writing. Some *points-to* analysis would be needed to avoid this. The original paper [AHB04] says it would require model-checking.

Expected Results: Only this false *RwW* should be flagged.

Results: The referred false *RwW* has been reported. The first transaction reads `field`, and the subsequent transaction updates it. Between these transactions, another thread could have updated the value.

Another false positive was reported, because of subfields and redundant accesses. If we disregard conflicts between accesses to a field, and accesses to its subfields, then we could eliminate this false positive.

Results Summary: Two false positives.

4.6.8 Test: StringBuffer

Subject: A simulation of `java.lang.StringBuffer` from [FF04]. This relates to a real anomaly discovered in the `StringBuffer` class from the JRE. The program is shown in Listing 4.11. For this implementation, the `MyStringBuffer` class was built as a wrapper to a `java.lang.StringBuffer` object, having only the method `append()` overridden. Calls to `length()` and `getChars()` are delegated to the wrapped object.

Listing 4.11: Code snippet for StringBuffer.

```
1  public MyStringBuffer append(MyStringBuffer other) {
2
3      int len = other.length();  // atomic
4
5      // ...other threads may change sb.length(),
6      // ...so len does not reflect the length of 'other' anymore
7
8      char[] value = new char[len];
9      other.getChars(0, len, value, 0);  // atomic
10
11     // ...
12
13     return this;
14 }
```

Documented Anomalies: In the `append()` method, the size of the `other` buffer is read in line 3, then used to create array of correct size in line 8. The `other` buffer could have changed meanwhile, effectively leaving the array with an incorrect size.

Expected Results: No anomalies should be detected, since both transactions only read the wrapped `StringBuffer`. Ideally, an *RwR* should be reported, but we would have to know what happens inside `length()` and `getChars()`. If we disregard both methods as both reading and writing, we would have *RwR* and *WrW*.

Results: As expected, no results reported.

Results Summary: One false negative.

4.6.9 Test: Account

Subject: A simple example from [vG03], illustrating a bank account manager. There is an `update()` method in the `Account` class, which is used to make a deposit. As shown in Listings 4.12 and 4.13, the `update()` method first checks the balance of the account in one atomic block, and then updates the balance in a separate one.

Listing 4.12: Code snippet for the Account test (a).

```
1  class Account {
2
3      int balance = 100;
4
5      int read () {
6          atomic {
7              return balance;
8          }
9      }
10
11     void update (int a) {
12         int tmp = read();
13         atomic {
14             balance = tmp + a;
15         }
16     }
17 }
```

Listing 4.13: Code snippet for the Account test (b).

```
1  class Update extends Thread {
2
3      static Account a;
4
5
6      static void main
7      (String[] args) {
8          a = new Account();
9          new Update().start();
10         new Update().start();
11     }
12
13     public void run() {
14         Update.a.update(42);
15     }
16
17 }
```

Documented Anomalies: The method `update()` has two transactions that should be performed as a single one. It reads `balance`, and then uses this value to update the current state. However, between these accesses the balance could have changed, resulting in an *RwW* anomaly.

Expected Results: The two separate transactions access the same field of the same class. Our approach should cleanly report this *RwW* anomaly.

Results: As expected, the anomaly was reported.

Results Summary: One anomaly correctly detected.

4.6.10 Test: Jigsaw

Subject: Real example of Jigsaw CMS, adapted from [vG03]. It contains interesting anomalies discovered during experiments of their approach. The relevant adapted code from Jigsaw is presented in Listing 4.14.

The boolean field `closed` in line 1 indicates the availability of the `entries` field, a map containing `ResourceStore` objects. The `checkClosed()` method atomically asserts that the map is available, and the `lookUpEntry()` method atomically returns the entry associated with the specified key. The atomic method `shutdown` empties the map, and sets `closed`. Finally, the non-atomic method `loadResourceStore()` calls the first two methods, checking the availability of the map, and retrieving a specific entry if it is available.

In our implementation, the `main()` method spawns a number of `Runner` threads. Each of these threads retrieves some resource store from the map, and then calls `shutdown()`.

Listing 4.14: Code snippet for the Jigsaw test.

```
1  boolean closed = false;
2  Map entries = new HashMap();
3
4  void checkClosed() {
5      atomic {
6          if (closed) throw new RuntimeException();
7      }
8  }
9  synchronized Entry lookupEntry(Object key) {
10     atomic {
11         return (Entry) entries.get(key);
12     }
13 }
14 void shutdown() {
15     atomic {
16         entries.clear();
17         closed = true;
18     }
19 }
20 ResourceStore loadResourceStore() {
21     checkClosed(); // R(closed)
22     Entry e = lookupEntry(key);
23     return e.getStore();
24 }
```

Documented Anomalies: The method `loadResourceStore()` in line 20 is not atomic. It first calls atomic `checkClosed()` in line 21, which asserts the availability of the object. After that, it retrieves the entry, in line 22, oblivious to a possible modification of this state by a current thread, resulting in an *RwR* anomaly.

Expected Results: Correct detection of the *RwR* anomaly. A possible false positive *WrW* may also be reported, resulting from calling `shutdown()` at the end.

Results: The present *RwR* anomaly was correctly detected. The false positive did not show up because method `entries.clear()` in line 16 is not assumed to modify `entries`. If we force a write, by setting `entries = null` after this statement, then the false *WrW* is reported. We will consider this false positive as a definite result.

Results Summary: One anomaly correctly detected. One false positive.

4.6.11 Test: Over-reporting

Subject: Example taken from [vG03] to illustrate a correct program which leads to a false positive (*over-reporting*) when using their approach. The relevant code is in Listing 4.15. A map data structure is shared among threads, which is initialized using lazy initialization: whenever a thread tries to initialize the map, a check is performed to see if the map was already available. If so, initialization is skipped, thus increasing efficiency.

Listing 4.15: Code snippet for the Over-reporting test.

```
1  /* Map */
2  Object[] keys, values;
3  boolean init_done = false;
4  void init() {
5      if (!init_done)
6          atomic {
7              init_done = true;
8              // update keys and values
9          }
10 }
11 Object get(Object key) {
12     atomic {
13         return ...  // read keys and values
14     }
15 }
16
17 /* Client */
18 static Map m;
19 public static void main(String args[]) {
20     m = new Map();
21     new MapClient().start();
22     new MapClient().start();
23 }
24 public void run() {
25     m.init();  // lazy initialization
26     Object o = m.get(key);
27 }
```

Documented Anomalies: There are no documented high-level anomalies in this test. This sample was provided in [vG03] as an example of a correct program that leads to a false positive.

However, there is a blatant low-level datarace, since the first check in `if (!init_done)` in line 5 is performed outside the scope of a transaction. This is a similar problem to what happens with the double-checked locking pattern [Goe], which contains a serious flaw due to particularities of the Java memory model. This issue could alter the meaning of the test results, since it is assumed that programs do not contain low-level anomalies.

Expected Results: No anomalies are expected to be reported. This false positive should not arise in the current implementation. However, assuming reads or writes from unavailable methods could cause false positives.

Results: There were two false *RwR* anomalies reported. Because the `keys` field is also read in the initialization, it is assumed that this access is related to the subsequent accesses in method `get()`.

The first false positive is caused by redundant accesses, and could be easily eliminated by slightly refining the detection tool. The second false positive is more serious, and would only be resolved by changing the anomaly specification. Notice that the atomic block in `init()` writes both `keys` and `values`, and the atomic block in `get()` reads both variables. These accesses could be the basis for a modification of the anomaly conditions.

Even though there is a low-level datarace which could interfere with test results, these false warnings do not seem to be related to it, and we will consider these warnings in the final results.

Results Summary: Two false positives.

4.6.12 Test: Underreporting

Subject: Example taken from [vG03] to illustrate an anomalous program which leads to a false negative (*under-reporting*) when using their approach. A shared counter is implemented through a wrapper class, which contains the current value and an `inc()` method, which increments the value by the given argument and returns the new value. As shown in Listing 4.16, an idiom is developed which uses this functionality to duplicate the present value. The counter is incremented by zero, returning the unaltered current value. This value is then provided as an increment argument, in order to duplicate it.

Listing 4.16: Code snippet for the Under-reporting test.

```
1  public class Counter {
2      int i;
3      int inc(int a) {
4          atomic {
5              i = i + a;
6              return i;
7          }
8      }
9  }
10
11 // Thread code
12 public void run() {
13     int i = c.inc(0);
14     c.inc(i);  // value could have changed between calls
15 }
```

Documented Anomalies: The code inside `run()` calls method `inc()` twice, one to check the state, and another one to update it. State could change in between, or be retrieved by another thread. If this happens, the update of one of the threads could be lost.

Expected Results: Our approach should correctly flag the *RwW* anomaly.

Results: The anomaly was correctly reported. This program runs two transactions, and they both read and write the current value. Because the counter value is read and then update in the subsequent transaction, an *RwW* anomaly is triggered.

Results Summary: One anomaly correctly detected.

4.6.13 Test: Allocate Vector

Subject: The same *Allocate Vector* test used in Chapter 3, taken from [IBM]. Multiple threads share an allocation vector, which features methods for checking free indexes and marking cells as allocated. The problematic portion of the code is in Listing 4.17.

Listing 4.17: Code snippet for the Allocate Vector test.

```
1  private void alloc_block(int i) throws Exception {
2      resultBuf[i] = vector.getFreeBlockIndex();  // atomic
3      if (resultBuf[i] != -1)
4          vector.markAsAllocatedBlock(resultBuf[i]);  // atomic
5  }
6
7  // ...
8
9  public void run() {
10
11     // ...
12     for (int i = 0; i < resultBuf.length; i++)
13         alloc_block(i);
14
15     for (int i = 0; i < resultBuf.length; i++) {
16         if (resultBuf[i] != -1)
17             vector.markAsFreeBlock(resultBuf[i]);
18     }
19
20     // ...
21 }
```

Documented Anomalies: A thread gets the index of a free block in line 2, and afterwards marks it as allocated in line 4. Both operations are atomic, but the cell could have been allocated by another thread.

Expected Results: The `getFreeBlockIndex()` method reads the shared vector, and the `markAsAllocatedBlock()` method alters it. Our tool should report this *RwW* anomaly correctly. Additionally, false positives may appear resulting from calling the `markAsFreeBlock()` at the end, on line 17.

Results: The documented anomaly was correctly flagged. An additional false positive is reported. Both `getFreeBlockIndex()` and `markAsAllocatedBlock()` methods perform reading and writing accesses to the shared vector. Therefore, an incorrect *RwW* anomaly is reported between the invocations of these two transactional methods. If we adjusted our tool to disregard *RwW* anomalies when the first transaction also writes the accessed field, we could eliminate this false positive.

Results Summary: One anomaly correctly detected. One false positive.

4.6.14 Test: Knight Moves

Subject: This test was specifically conceived to test our approach. The relevant code is in Listing 4.17. This program the optimum number of a moves a knight must make to capture another piece in a chess board. The solution may be obtained either sequentially, through a single thread that performs a graph search, or concurrently. The concurrent solution works by spawning a thread to check each move of the knight, and comparing it with the best move found so far by other threads.

Listing 4.18: Code snippet for the Knight Moves test.

```
public static void main(String[] args) {
    KnightMoves km = new KnightMoves(knight, prey);
    km.solve();  // solve board concurrently
    System.out.println("Moves: " + km.get_solution(...));

    km.solve_correct();  // solve board sequentially
    System.out.println("Correct Moves: " km.get_solution(...));
}

// ...

public int get_solution(...) {
    int x;
    atomic { x = solution[...][...]; }
    return x;
}

// ...

public void solve() {
    new Solver(...).start();
    try { s.join(); } catch (Exception e) {}
}

// ...

private void check_and_set_solution() {
    if (km.get_solution(me) <= moves)  // atomic
        // ...  // solution could have changed
    km.set_solution(me, moves);  // atomic
}

// ...

public void run() {
    // ...
    if (check_and_set_solution() < 0)
        return;
    // ...
    new Solver(...).start();
    // ...
}
```

Documented Anomalies: There is an error in the concurrent solution. The `run()` method of each thread is based on the `check_and_set_solution()`, which checks the state of the best solution found in line 28, and then possibly updates it on line 30. Parallel threads could have changed the shared solution between these two atomic operations.

Expected Results: The documented anomaly should be reported correctly, because the two atomic blocks involved in the anomaly access the same shared field, first reading and then writing it. Possible false positives could possibly arise due to the additional `solve_correct()` method call in line 6, which also generates a solution, and the two calls to `get_solution()` in lines 4 and 7 which read the achieved solution.

Results: There were three anomaly reports. The first is relative to the real anomaly. The remaining two are false positives inside the `main()` method. Notice that the invocation of `solve()` in line 3 waits for all generated threads to finish before it returns. Therefore, the remaining accesses by `main()` are guaranteed to happen without interleavings with other threads.

Results Summary: One anomaly correctly detected. Two false positives.

4.7 Test Results

We now present a summary of the results of the tests listed and discussed in Section 4.6. Fourteen tests were performed. The results are summarized in Table 4.1. Column *Anomalies* lists the total anomalies present in each test, which should ideally be detected. Column *Warnings* shows for each test the total anomaly reports issued by our tool. From these reports, those which correspond to real anomalies are shown in column *Correct*, and those which are false positives are counted in column *False*. Column *Categories* states the nature of the false positives, as discussed in the remainder of this section. Finally, column *Missed* shows the anomalies which were not detected, originating false negatives.

In a total of 12 anomalies present in these programs, 10 were correctly pointed out: 2 *RwR* anomalies, 3 *WrW* and 5 *RwW*. The remaining 2 anomalies have not been detected.

The false negatives were not due to imprecision of the anomaly patterns, but rather to data accesses in JRE classes, whose source code is not available. These anomalies involve standard methods from the JRE which may possibly read or update internal data. In each of these missed anomalies, one of the conflicting operations was performed by a standard Java method. Since this code is not available, these methods are ignored, thus resulting in imprecision. As a possible approach to solve this problem, methods for which code is unavailable could be assumed to both read and modify the object to which the call is made, as well as any object which is passed as argument. Another alternative is to consider user-provided information that indicates which methods perform read or write operations over which objects.

Table 4.1: Test results summary.

Test	Anomalies	Warnings	Correct	False	Categories	Missed
[BBA08] Connection	2	2	1	1	*C*	1
[AHB03] Coordinates'03	1	4	1	3	*2B, 1C*	0
[AHB03] Local Variable	1	2	1	1	*A*	0
[AHB03] NASA	1	1	1	0	—	0
[AHB04] Coordinates'04	1	4	1	3	*3C*	0
[AHB04] Buffer	0	7	0	7	*2A, 1B, 2C, 2D*	0
[AHB04] Double-Check	0	2	0	2	*1A, 1B*	0
[FF04] StringBuffer	1	0	0	0	—	1
[vG03] Account	1	1	1	0	—	0
[vG03] Jigsaw	1	2	1	1	*C*	0
[vG03] Over-reporting	0	2	0	2	*1A, 1C*	0
[vG03] Under-reporting	1	1	1	0	—	0
[IBM] Allocate Vector	1	2	1	1	*C*	0
Knight Moves	1	3	1	2	*2B*	0
Total	**12**	**33**	**10**	**23**	**5A, 6B, 10C, 2D**	**2**

In addition to the correctly detected anomalies, there were also 23 false positives (70% of total warnings). We group the causes for these imprecisions in 4 different categories.

First, out of these 23 false warnings, 5 were due to redundant reading operations. For example, in a read operation `object.field`, two readings are actually being performed: `object` and `field`. It makes no sense that two instances of this statement be involved in a RwR. However, our tool is for now regarding these two readings as potentially being involved in an anomaly. It would be possible to eliminate these false positives if it was considered that both readings access the same data. These false warnings will be described as belonging in category *A*.

Another 6 false positives are related to interleavings which are not atomic, even though they are correct. Additional semantic information would have to be provided or inferred in order to correctly evaluate these cases. Most of these would need another level of analysis, such as a model checker, and a single one would be solved by providing *points-to* analysis. These false warnings will be described as belonging in category *B*.

Of the remaining false positives, 10 of them could be eliminated by refining the definition of the *common patterns* described in Section 4.3, with alterations that are indeed intuitive. For example, an *RwR* anomaly could be ignored if the second transaction would write both values involved. However, these alterations should be made carefully, as they could harm the overall behavior in other tests. Eliminating specific false positives could also result in new false negatives. These false warnings will be described as belonging in category *C*.

Finally, the remaining 2 false reports are also related to correct accesses which are matched by our anomaly patterns. There does not seem to be a simple way to adapt the anomaly patterns in order to leave out these correct accesses, without seriously compromising the precision of the approach. These false positives belong in category *D*.

4.8 Concluding Remarks

We have defined and implemented a new approach to static detection of high-level concurrency anomalies in Transactional Memory programs. This new approach works by conservatively tracing transactions and matching the interference between each consecutive pair of transactions against a set of defined anomaly patterns. Our methodology also allows us to assess the usefulness of new patterns, and is believed to help identify patterns of commonly occurring errors.

We have analyzed common criteria for reporting high-level anomalies, and attempted to provide a more useful criteria by defining three common patterns. These patterns have been put to practice with our developed tool, with satisfactory results. The combined usage of these patterns and our tool provided results which are at least as satisfactory as those of other approaches. We may therefore conclude that these conservative traces of transactions are a reasonable indicator of the behavior of a program, since our results rival with those of dynamic approaches. We can also see that the analysis of transaction pairs, instead of whole thread executions, works with a good precision.

The developed tool could be improved by further work in refining it with better error patterns. The addition of *points-to* and *happens-in-parallel* analyses would also help to improve the tool. These are hard to implement with precision in a static analysis, but would boost the usefulness of this work, and would be essential to conceive an application for a production scenario. Another improvement would be achieved by enabling the analysis of standard or unavailable methods, by implementing the strategies hinted in Section 4.5.6. By solving the issue of redundant read accesses, explained in Section 4.7, the detection results would also be considerably improved.

5

Conclusion

5.1 Conclusions

Transactional Memory (TM) is a promising concept that may possibly replace some of the currently available concurrency mechanisms in the near future. Being an emerging technology, TM still lacks the maturity and standards of other paradigms. Although the TM model tends to be less error-prone than more classical approaches to concurrency control, they are not completely free from concurrency anomalies, which actually rather similar to those observed in lock-based programs. Due to the semantic differences between TM and lock-based models, one can not blindly use the already existing tools and mechanisms for analyzing TM programs. Consequently, there is a clear shortage of tools and mechanisms to analyze and detect errors in TM programs.

With this work we contributed to improve the ease of development using the TM programming model. In Section 1.3 we made the general claim that doing static analysis of TM programs allow to detect runtime anomalies with reasonable precision.

To support the above general claim, we have also argued that the detection of low-level dataraces in TM programs may be reduced to the problem of detecting dataraces in an equivalent monitor-based program. To ground this argument, we proposed the techniques and methods described in Chapter 3. These techniques and methods were implemented in a tool and evaluated against well-known test programs with documents bugs. The observed results validated the effectiveness of our tool and confirmed our claim.

An additional argument used to support the general claim was that most of the high-level anomalies in TM programs result from interactions of pairs of consecutive transactions from one thread with a third transaction from another thread. Chapter 4 describes our definition and implementation of a new approach to static detection of high-level concurrency anomalies in

TM programs. This new approach works by conservatively tracing transactions and matching the interference between each consecutive pair of transactions against a set of defined anomaly patterns. The approach was again evaluated against well-known tests programs with documents bugs. The observed results validated the effectiveness of our tool and confirmed our claim.

Thus, we may conclude that static analysis is an effective and reliable approach to the detection of anomalies in TM programs.

5.2 Future Work

The work presented throughout this thesis is open to possible further developments and enhancements. It is also suitable to be the basis of future works that target static analysis and transformation of TM programs. In this section we present possible future work, divided in two major areas. Firstly, we shall enumerate possible enhancements that could be performed on the detection tool described in Chapter 4, and which have already been hinted in that chapter. Afterwards, we present possible optimizations that could be performed on TM programs, which would be related to this work, and could possibly use the implementations and conclusions we have provided.

5.2.1 Enhancements on the Detection of High-Level Anomalies

Provide Additional Static Analyses

The tool described in Chapter 4 would yield more precise results if it was provided with additional static analyses. Adding *points-to* and *may-happen-in-parallel* analyses would increase precision and help eliminate many of the false anomaly reports. These kinds of analysis have been the subject of intense study, and are available in a wide variety of algorithms. One could possibly take the effort of adapting one of these approaches to work in conjunction with our tool.

Eliminate Redundant Accesses

Many of the false positives reported by our tool are related with redundant read accesses. The dereferencing of subfields of fields yields multiple accesses which could undesirably trigger an anomaly. A more sophisticated interpretation of field accesses would improve the detection results.

Analyze Unavailable Methods

Another major source of incorrect or imprecise results is the skipping of methods whose source code is not available. Our tool currently ignores calls to methods that have not been parsed by it, such as standard methods of the JRE. Three approaches are suggested that could solve this issue. First, the assumption that unavailable methods invariably read and write all involved

objects and fields would enable the report of certain missed anomalies, even though it would potentialy increase the occurrence of false positives. Secondly, having the user explicitly declare the read and write accesses to fields originated from unavailable methods would more precisely increase the accuracy of the detection. Even though this would require the exhaustive annotation of every unavailable method, this mechanism could be used in conjunction with the first one, thus providing a compromise between the two. And finally, it would be possible to conjugate our approach with an analysis of compiled programs in byte-code form, which would be definitely able to assess the read and write operations of every method.

Pin-point Anomaly Causes in Code

Currently, when reporting potential anomalies, our tool indicates the line in the source code of the atomic blocks that triggered the anomaly. While this may be the most straight-forward report to provide, it is not always easy to interpret. If a library provides atomic methods, and the client code performs calls to these methods which should be encapsulated in a transaction, then our tool will report an anomaly pertaining the transactions inside the library. By using call-stack information, it would be possible to pin-point more precisely the statements in the client code which should have been transactional.

Improve Results Presentation

The current version of our tool simply outputs a large block of text information, containing much unfiltered data related with the analysis. In order to deploy this tool for a more frequent use, it would be necessary to provide a more readable output. An application for graphical presentation of this data would highly increase the value of this output information. Furthermore, this would possibly allow further conclusions on the behavior of transactional programs, thus enabling future work on TM program analysis.

Increase the Flexibility of Trying new Anomaly Patterns

Currently, the architecture for the matching of anomaly patterns is quiet rigid. The alteration of the anomaly patterns at this stage is not excessively complex, although it would pose an obstacle to an exhaustive experiment regarding a high number of different patterns. It would be possible to make this architecture more parameterizable, thus easing the enabling or disabling of each pattern. A more ambitious solution would be to develop a notation to describe anomaly patterns, and modify the tool to conform with this notation.

5.2.2 Possible Optimizations on TM Programs

Transform Highly-Conflicting Transactions

Some transactional programs may contain a structure which leads to a high number of conflicts between transactions. On its turn, a high rate of conflicts forces the underlying TM system to abort transactions and take corrective actions more often, which leads to a greater overhead. In

such a situation, the goal of optimizing processor usage by allowing more concurrency would have been failed, even though the usage of atomic blocks is still a more desirable programming model than locking. The use of optimistic transactions in this case actually damages performance instead of helping. If one would analyze TM programs for such high-conflicting transactions, then these transactions could be signaled to be run in a pessimistic mode, running as though the code was guarded by a lock.

The application of the tool described in Chapter 4 already provides the list of all fields that a transaction may possibly access. In conjunction with a variation of the translation tool described in Chapter 3, transactions which are suspected to raise a high number of conflicts could be transformed in order to become lock-based synchronized regions.

Eliminate Never-Conflicting Transactions

The use of transactions implies the management of some control information by the underlying TM system. If we must manage these data structures without benefits, then we are incurring in an unnecessary overhead, with time and spatial implications. A program which contains unnecessary transactions could be optimized if these transactions were removed. In a similar fashion to what happens with highly-conflicting transactions, the tool described in Chapter 4 could be adapted to check for transactions which have no possible field accesses in common, and therefore may not conflict. The application of a transformation tool derived from the one described in Chapter 3 could replace these transactions by the equivalent non-transactional code.

Bibliography

[AAK+06] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. *IEEE Micro*, 26(1):59–69, 2006.

[AHB03] C. Artho, K. Havelund, and A. Biere. High-level data races, 2003.

[AHB04] Cyrille Artho, Klaus Havelund, and Armin Biere. Using block-local atomicity to detect stale-value concurrency errors. In Farn Wang, editor, *ATVA*, volume 3299 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2004.

[AKJ+08] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Lee-tm: A non-trivial benchmark for transactional memory. In *ICA3PP '08: Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing*. LNCS, Springer, June 2008.

[AKW+08] Mohammad Ansari, Christos Kotselidis, Ian Watson, Chris Kirkham, Mikel Luján, and Kim Jarvis. Lee-tm: A non-trivial benchmark suite for transactional memory. In *ICA3PP '08: Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*, pages 196–207, Berlin, Heidelberg, 2008. Springer-Verlag.

[All70] Frances E. Allen. Control flow analysis. Technical report, IBM Corporation, July 1970.

[ATLM+06] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[BBA08] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. *SIGPLAN Not.*, 43(10):227–244, 2008.

[BLM05] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[BM] Michael Brukman and Andrew C. Myers. Polyglot Parser Generator. Hosted at http://www.cs.cornell.edu/projects/polyglot/ppg.html.

[CCM+06] J.W. Chung, H. Chafi, C.C. Minh, A. McDonald, B. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. *High-Performance Computer Architecture, International Symposium on*, 0:266–277, 2006.

[CK04] David R. Cok and Joseph R. Kiniry. Esc/java2: Uniting esc/java and jml - progress and issues in building and using esc/java2. In *In Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*. Springer–Verlag, 2004.

[CLL+02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. *SIGPLAN Not.*, 37(5):258–269, 2002.

[Cor06] James R. Cordy. Source transformation, analysis and generation in txl. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 1–11, New York, NY, USA, 2006. ACM.

[dCLSL01] Jong deok Choi, Alexey Loginov, Vivek Sarkar, and Alexey Logthor. Static datarace analysis for multithreaded object-oriented programs. Technical report, IBM Research Division, Thomas J. Watson Research Centre, 2001.

[Dia08] Ricardo Dias. Extending java with atomic blocks for transactional memory systems. Technical report, Universidade Nova de Lisboa, 2008.

[Dia09] Ricardo Dias. Source-to-source java stm framework compiler. Technical report, Departamento de Informática FCT/UNL, April 2009.

[DLC08] Ricardo Dias, João Lourenço, and Gonçalo Cunha. Developing libraries using software transactional memory. *ComSIS*, 5(2), December 2008.

[DMS04] *DMS: Program Transformations for Practical Scalable Software Evolution*. IEEE Press, 2004.

[DMS05] *Re-engineering C++ Component Models Via Automatic Program Transformation*. IEEE, 2005.

[DPE06] Matthew B. Dwyer, Suzette Person, and Sebastian Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 92–104, New York, NY, USA, 2006. ACM.

[DS91] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. *SIGPLAN Not.*, 26(12):85–96, 1991.

[EA03] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, 2003.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. *SIGPLAN Not.*, 35(5):219–232, 2000.

[FF04] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, New York, NY, USA, 2004. ACM.

[FQ03a] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. *PLDI03*, 2003.

[FQ03b] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. *SIGPLAN Not.*, 38(5):338–349, 2003.

[Goe] Brian Goetz. Double-checked locking: Clever, but broken. http://www.javaworld.com/jw-02-2001/jw-0209-double.html.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM.

[HFP+] Scott E. Hudson, Andrea Flexeder, Michael Petter, et al. CUP – LALR Parser Generator in Java. Hosted at http://www2.cs.tum.edu/projects/cup.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[IBM] IBM's Concurrency Testing Repository.

[Jon03] Joel Jones. Abstract syntax tree implementation idioms. *Pattern Languages of Program Design*, 2003. Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003) http://hillside.net/plop/plop2003/papers.html.

[KCH+06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 209–220, New York, NY, USA, 2006. ACM.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[Lie04] Sean Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, May 2004.

[Lip75] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[loc] Lock_lint – static data race and deadlock detection tool for c. `http://developers.sun.com/solaris/articles/locklint.html`.

[LR06] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI*, pages 308–319. ACM Press, 2006.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *CC*, pages 138–152, 2003.

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38(10):167–178, 2003.

[QDLT09] Yao Qi, Raja Das, Zhi Da Luo, and Martin Trotter. Multicoresdk: a practical and efficient data race detector for real-world applications. In *PADTAD '09: Proceedings of the 7th Workshop on Parallel and Distributed Systems*, pages 1–11, New York, NY, USA, 2009. ACM.

[Raz06] Aoun Raza. *A Review of Race Detection Mechanisms*, pages 534–543. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006.

[RC99] Laurie Hendren Vijay Sundaresan Patrick Lam Etienne Gagnon Raja Vallée-Rai and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[RM09] Neha Rungta and Eric G. Mercer. Clash of the titans: Tools and techniques for hunting bugs in concurrent programs. In *PADTAD'09*, July 2009.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[SMD+06] Arrvindh Shriraman, Virendra J. Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N. Scherer, Iii Michael, and F. Spear. Hardware acceleration of software transactional memory. Technical report, Dept. of Computer Science, Univ. of Rochester, 2006.

[SQ03] Markus Schordan and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, August 2003.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[Ste93] Nicholas Sterling. Warlock: A static data race analysis tool. In *Winter USENIX*, pages 97–106, San Diego, California, January 1993.

[vG03] Christoph von Praun and Thomas R. Gross. Static detection of atomicity violations in object-oriented programs. In *Journal of Object Technology*, page 2004, 2003.

[vHKO02] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[Vis04] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[VJL07] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214, New York, NY, USA, 2007. ACM.

[WS03] Liqiang Wang and Scott D. Stoller. Run-time analysis for atomicity. *Electronic Notes in Theoretical Computer Science*, 89(2):191–209, 2003. RV '2003, Run-time Verification (Satellite Workshop of CAV '03).

[YRC05] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39(5):221–234, 2005.

Source Code of Test Subjects

The source code for the tests performed for validation of our approach is now presented.

A.1 Tests with Low-Level Dataraces

The detection of low-level dataraces is addressed in Chapter 3.

Many tests are divided in two versions: one which uses Java monitors and synchronized blocks, and another one which uses atomic blocks for TM. The Java version files will contain the extension `.java`, and the TM version files will have the extension `.atom.java`. The Java version files will be totally listed. The files relative to the TM versions will only be listed when they contain differences.

A.1.1 Empty Program

This test is discussed in Section 3.6.1, on Page 42.

Empty.java

```
1 public class Empty {
2     public static void main(String[] args)
3     {}
4
5 }
```

A.1.2 HelloWorld Single-Threaded

This test is discussed in Section 3.6.2, on Page 42.

HelloMono.java

```
1  public class HelloMono {
2      public static int x = 0;
3      public static void main(String[] args) {
4          x = 42;
5          System.out.println("Hello: " + x);
6      }
7  }
```

A.1.3 HelloParallel Buggy

This test is discussed in Section 3.6.3, on Page 43.

HelloParallel.java

```
1  public class HelloParallel extends Thread {
2      public static int x = 0;
3
4      public static void main(String args[]) {
5          new HelloParallel().start();
6          System.out.println(x);
7      }
8      public void run() {
9          x = 42;
10     }
11 }
```

A.1.4 HelloParallel Buggy Synchronized

This test is discussed in Section 3.6.4, on Page 43.

HelloParallel.java

```
1  public class HelloParallel extends Thread {
2      public static int x = 0;
3
4      public static void main(String args[]) {
5          new HelloParallel().start();
6          System.out.println(x);
7      }
8      public synchronized void run() {
9          x = 42;
10     }
11 }
```

HelloParallel.atom.java

```
public class HelloParallel extends Thread {
    public static int x = 0;

    public static void main(String args[]) throws InterruptedException {
        new HelloParallel().start();
        System.out.println(x);
    }
    public void run() {
        atomic {
            x = 42;
        }
    }
}
```

A.1.5 HelloParallel Buggy Corrected

This test is discussed in Section 3.6.5, on Page 44.

HelloParallel.java

```
public class HelloParallel extends Thread {
    public static int x = 0;

    public static void main(String args[]) {
        new HelloParallel().start();
        synchronized (hp) {
            System.out.println(x);
        }
    }
    public synchronized void run() {
        x = 42;
    }
}
```

HelloParallel.atom.java

```
public class HelloParallel extends Thread {
    public static int x = 0;

    public static void main(String args[]) {
        new HelloParallel().start();
        atomic {
            System.out.println(x);
        }
    }
    public void run() {
```

```
11          atomic {
12              x = 42;
13          }
14      }
15  }
```

A.1.6 HelloWorld Modular

This test is discussed in Section 3.6.6, on Page 44.

Main.java

```
1  package multi;
2
3  public class Main {
4      public static Integer x = 0, y = 0;
5
6      public static void main(String args[]) {
7          RunMe rm = new RunMe();
8          MyThread mt = new MyThread(rm);
9          Printer p = new Printer();
10         mt.start();
11         p.print(x);
12     }
13 }
```

MyThread.java

```
1  package multi;
2
3  public class MyThread extends Thread {
4
5      public MyThread (Runnable r) {
6          super(r);
7      }
8  }
```

RunMe.java

```
1  package multi;
2
3  public class RunMe implements Runnable {
4
5      public void run () {
6          Main.x = 42;
7      }
8  }
```

Printer.java

```
1  package multi;
2
3  public class Printer {
4
5      public synchronized void print (Object msg) {
6          System.out.println(msg);
7      }
8  }
```

Printer.atom.java

```
1  package multi;
2
3  public class Printer {
4
5      public void print (Object msg) {
6          atomic {
7              System.out.println(msg);
8          }
9      }
10 }
```

A.1.7 HelloWorld Complex

This test is discussed in Section 3.6.7, on Page 45.

HelloComplex.java

```
1  public class HelloComplex extends Thread {
2
3      private static int x = 0, y = 0;
4
5      public static void main(String args[]) {
6
7          new HelloComplex().start();
8          new HelloComplex().start();
9
10         synchronized (HelloComplex.class) {
11             x = 42;
12             y = 47;
13         }
14
15         System.out.println("Hello (cruel) world!");
16         System.out.println("And now, a number:");
17         System.out.println(x);
18     }
```

```
19
20      public void run() {
21          x++;
22          y++;
23      }
24  }
```

HelloComplex.atom.java

```
1   public class HelloComplex extends Thread {
2
3       private static int x = 0, y = 0;
4
5       public static void main(String args[]) {
6
7           new HelloComplex().start();
8           new HelloComplex().start();
9
10          atomic {
11              x = 42;
12              y = 47;
13          }
14
15          System.out.println("Hello (cruel) world!");
16          System.out.println("And now, a number:");
17          System.out.println(x);
18      }
19
20      public void run() {
21          x++;
22          y++;
23      }
24  }
```

A.1.8 Account

This test is discussed in Section 3.6.8, on Page 47.

ManageAccount.java

```
1   package account.ibm;
2
3   public class ManageAccount extends Thread {
4       Account account;
5       static Account[] accounts = new Account[10];// we may add more later to
6                                                   // increase the parallelism
7                                                   // level
8       static int num = 2;// the number of the accounts
```

```
9       static int accNum = 0;// index to insert the next account
10      int i;// the index
11
12      public ManageAccount(String name, double amount) {
13          account = new Account(name, amount);
14          i = accNum;
15          accounts[i] = account;
16          accNum = (accNum + 1) % num;// the next index in a cyclic order
17      }
18
19      public void run() {
20          account.depsite(300);
21          account.withdraw(100);
22          Account acc = accounts[(i + 1) % num];// transfering to the next account
23          account.transfer(acc, 99);
24      }
25
26      static public void printAllAccounts() {
27          for (int j = 0; j < num; j++) {
28              if (ManageAccount.accounts[j] != null) {
29                  ManageAccount.accounts[j].print();
30                  ;// print it
31              }
32          }
33      }
34
35  }// end of class ManageAccount
```

Main.java

```
1   package account.ibm;
2
3   import java.io.*;
4
5   public class Main {
6
7       public static void main(String[] args) {
8
9           PrintStream out = null;
10          try {
11              if (args.length > 0) {
12                  out = new PrintStream(new FileOutputStream(args[0]));
13                  //directint all the "out" printing to the file.
14                  if (args.length == 1) {// the default value=little
15                      System.out.println("The default value=little");
16                  }
17                  out.print("<Account program,");
18                  if (args.length == 2) {// the concurrency is optional
19                      String concurrencyLevel = args[1];
```

```
20                  if (concurrencyLevel.compareTo("little") == 0) {
21                      ManageAccount.num = 2;
22                      System.out.println("concurrency level = little");
23                  } else if (concurrencyLevel.compareTo("average") == 0) {
24                      ManageAccount.num = 5;
25                      System.out.println("concurrency level = average");
26                  } else if (concurrencyLevel.compareTo("lot") == 0) {
27                      ManageAccount.num = 10;
28                      System.out.println("concurrency level = lot");
29                  } else {// wrong second argument
30                      System.out
31                              .println("The second argument should be one of the
                                    following:\n little,average or lot");
32                      System.exit(1);
33                  }
34              } else {// more than 2 arguments
35                  if (args.length > 2) {
36                      System.out
37                              .println("The program can accept only one or two
                                    arguments");
38                      System.exit(1);
39                  }
40              }
41          } else {// main's parameter is missing
42              System.out.println("The output file name is missing");
43              System.exit(1);
44          }
45          System.out.println("The Initial values:");
46          ManageAccount[] bank = new ManageAccount[ManageAccount.num];
47              // 2 is the default size
48          String[] accountName = { new String("A"), new String("B"),
49                  new String("C"), new String("D"), new String("E"),
50                  new String("F"), new String("G"), new String("H"),
51                  new String("I"), new String("J"), };
52          for (int j = 0; j < ManageAccount.num; j++) {
53              bank[j] = new ManageAccount(accountName[j], 100);
54              ManageAccount.accounts[j].print();
55              ;// print it
56          }
57
58          // start the threads
59          for (int k = 0; k < ManageAccount.num; k++) {
60              bank[k].start();
61          }
62
63          // wait until all are finished
64          for (int k = 0; k < ManageAccount.num; k++) {
65              bank[k].join();
66          }
67          System.out.println("The final values:");
```

```
68              ManageAccount.printAllAccounts();
69
70              // updating the output file
71              boolean less = false, more = false;// flags which will indicate
72                                                 // the kind of the bug
73              for (int k = 0; k < ManageAccount.num; k++) {
74                  if (ManageAccount.accounts[k].amount < 300) {
75                      less = true;
76                  } else if (ManageAccount.accounts[k].amount > 300) {
77                      more = true;
78                  }
79              }
80              if ((less == true) && (more == true))
81                  out
82                          .print(" There is amount with more than 300and there is
                                amount with less than 300, No Lock>");
83              if ((less == false) && (more == true))
84                  out.print(" There is amount with more than 300, No Lock>");
85              if ((less == true) && (more == false))
86                  out.print(" There is amount with less than 300, No Lock>");
87              if ((less == false) && (more == false))
88                  out.print(" All amounts are 300,None>");
89              out.close();
90          } catch (Exception e) {// FileNotFound,Security
91              if (out != null)
92                  out.close();
93          }
94
95      }// end of function main
96  }// end of class Main
```

Account.java

```
1   package account.ibm;
2
3   public class Account {
4
5       double amount;
6       String name;
7
8       public Account(String nm, double amnt) {
9           amount = amnt;
10          name = nm;
11      }
12
13      synchronized void deposit(double money) {
14          amount += money;
15      }
16
```

```
17      synchronized void withdraw(double money) {
18          amount -= money;
19      }
20
21      synchronized void transfer(Account ac, double mn) {
22          amount -= mn;
23          ac.amount += mn;
24      }
25
26      synchronized void print() {
27          System.out.println(name + "--" + amount);
28      }
29
30  }
```

Account.atom.java

```
1   package bruno.account;
2
3   public class Account {
4
5       double balance;
6
7       String name;
8
9       public Account(String name, double balance) {
10          this.balance = balance;
11          this.name = name;
12      }
13
14      void deposit(double money) {
15          atomic {
16              balance += money;
17          }
18      }
19
20      void withdraw(double money) {
21          atomic {
22              balance -= money;
23          }
24      }
25
26      void transfer(Account other, double amount) {
27          atomic {
28              balance -= amount;
29          }
30          other.balance += amount; //RACE!!!
31
32      }
```

```
33
34      void print() {
35          atomic {
36              System.out.println(name + "(" + balance + ")");
37          }
38      }
39
40  }
```

A.1.9 Airline

This test is discussed in Section 3.6.9, on Page 48.

Main.java

```
1   package fse2006airline;
2
3   public class Main {
4
5       /*
6        * Second parameter is the number of threads
7        * Third parameter is the cushion
8        */
9       private static int numberThreads = 10;
10      private static int cushion = 3;
11
12      public static void main(String[] args) {
13          if (args.length < 2){
14              System.out.println("ERROR: Expected 2 parameters");
15          }else{
16              numberThreads = Integer.parseInt(args[0]);
17              cushion = Integer.parseInt(args[1]);
18              new Bug("test",numberThreads,cushion);
19          }
20      }
21  }
```

Bug.java

```
1   package fse2006airline;
2
3   import java.io.FileOutputStream;
4   import java.io.FileNotFoundException;
5   import java.io.IOException;
6
7   public class Bug implements Runnable{
8
9       static int  Num_Of_Seats_Sold =0;
```

```
10      int         Maximum_Capacity, Num_of_tickets_issued;
11      boolean     StopSales = false;
12      Thread      threadArr[] ;
13      FileOutputStream output;
14
15      private String fileName;
16
17      public Bug (String fileName, int size, int cushion){
18          this.fileName = fileName;
19      Num_of_tickets_issued = size;
20
21           Maximum_Capacity = Num_of_tickets_issued - cushion;
22           threadArr = new Thread[Num_of_tickets_issued];
23
24  /**
25   * starting the selling of the tickets:
26   * "StopSales" indicates to the airline that the max capacity was sold & that they
         should stop issuing tickets
27   */
28          for( int i=0; i < Num_of_tickets_issued; i++) {
29
30             threadArr[i] = new Thread (this) ;
31  /**
32   * first the airline is checking to see if it's agents had sold all the seats:
33   */
34              if( StopSales ){
35                  Num_Of_Seats_Sold--;
36                  break;
37              }
38   /**
39   * THE BUG : StopSales is updated by the selling posts ( public void run() ), and by
         the time it is updated
40   *           more tickets then are alowed to be are sold by other threads that are
         still running
41   */
42              threadArr[i].start();  // "make the sale !!!"
43          }
44
45          if (Num_Of_Seats_Sold > Maximum_Capacity)
46              throw new RuntimeException("bug found");
47      }
48  /**
49   * the selling post:
50   * making the sale & checking if limit was reached ( and updating "StopSales" ),
51   */
52      public void run() {
53
54      Num_Of_Seats_Sold++;                          // making the sale
55
56      if (Num_Of_Seats_Sold > Maximum_Capacity)     // checking
```

```
57                StopSales = true;                    // updating
58        }
59    }
```

Bug.atom.java

```
1   // import org.deuce.Atomic;
2
3   import java.io.FileOutputStream;
4   import java.io.FileNotFoundException;
5   import java.io.IOException;
6
7   public class Bug implements Runnable
8   {
9       private static final boolean BUG_0=false;
10      private static final boolean BUG_1=false;
11      private static final boolean BUG_2=false;
12
13      static int  Num_Of_Seats_Sold=0;
14      int         Maximum_Capacity;
15      int         Num_of_tickets_issued;
16      boolean     StopSales=false;
17      Thread      threadArr[];
18      FileOutputStream output;
19
20      private String fileName;
21
22      /* size=numberThreads=Num_of_tickets_issued */
23      public Bug(String fileName, int size, int cushion)
24      {
25          boolean out=false;
26
27          this.fileName = fileName;
28          Num_of_tickets_issued = size;
29
30          Maximum_Capacity=Num_of_tickets_issued-cushion;
31          threadArr=new Thread[Num_of_tickets_issued];
32
33          /**
34           * starting the selling of the tickets:
35           * "StopSales" indicates to the airline that the
36           * max capacity was sold & that they should stop
37           * issuing tickets
38           */
39          for (int i=0; i < Num_of_tickets_issued; i++)
40          {
41              threadArr[i]=new Thread(this);
42
43              /**
```

```
44              * first the airline is checking to see
45              * if it's agents had sold all the seats:
46              */
47             if (check_stop_sales() != 0)
48             {
49                 out=true;
50                 break;
51             }
52
53             /**
54              * THE BUG : StopSales is updated by the selling
55              * posts ( public void run() ), and by the time it
56              * is updated more tickets then are allowed to be
57              * are sold by other threads that are still running
58              */
59             threadArr[i].start();  // "make the sale !!!"
60         }
61
62         if (!BUG_0)
63         {
64             /**
65              * Fixes Bug 0
66              */
67             for (int i=0; i < Num_of_tickets_issued
68                      && threadArr[i] != null; i++)
69             {
70                 try
71                 {
72                     threadArr[i].join();
73                 }
74                 catch (Exception e) {}
75             }
76         }
77
78         if (!BUG_2)
79         {
80             /**
81              * It is possible for the thread loop to exit before
82              * the threads are finished.
83              * In that case Num_Of_Seats_Sold  must be decremented.
84              *
85              * Fixes Bug 2.
86              */
87             if (StopSales && !out)
88                 Num_Of_Seats_Sold--;
89         }
90
91         /* Debug stuff
92            System.out.println("sold: "+Num_Of_Seats_Sold
93            +" max: "+Maximum_Capacity);
```

```
94          */
95
96          if (Num_Of_Seats_Sold > Maximum_Capacity)
97          {
98              System.out.println("ups!");
99              throw new RuntimeException("this should not happen");
100         }
101     }
102
103     private int check_stop_sales()
104     {
105         atomic
106         {
107             if (StopSales)
108             {
109                 Num_Of_Seats_Sold--;
110                 return -1;
111             }
112
113             return 0;
114         }
115     }
116
117     /**
118      * the selling post:
119      * making the sale & checking if limit was reached
120      * ( and updating "StopSales" ),
121      */
122     public void run()
123     {
124         atomic
125         {
126             if (BUG_1)
127                 Num_Of_Seats_Sold++; // making the sale
128             else if (!StopSales) // Fixes bug 1
129                 Num_Of_Seats_Sold++; // making the sale
130
131             if (Num_Of_Seats_Sold > Maximum_Capacity)   // checking
132                 StopSales=true;                        // updating
133         }
134     }
135 }
```

A.1.10 Piper

This test is discussed in Section 3.6.10, on Page 50.

IBM_Airlines.java

```
1  package fse2006piper;
2
3  import java.io.*;
4
5  ublic class IBM_Airlines {
6      static int NUM_OF_SEATS = 1; // number of seats on the plane
7      private static int _numberOfThreads = 32; // num of threads from input
8
9      public static void main(String[] args) {
10         int numSeats = 2;
11         if (args != null && args.length == 3) {
12             NUM_OF_SEATS = Integer.parseInt(args[0]);
13             _numberOfThreads = Integer.parseInt(args[1]);
14             numSeats = Integer.parseInt(args[2]);
15         }
16
17         IBM_Airlines airline = new IBM_Airlines();
18
19         Piper p = new Piper("file", _numberOfThreads * NUM_OF_SEATS, numSeats);
20
21         for (int i = 1; i <= _numberOfThreads; i++) { // Create all Producer
22                                                       // threads
23             new Thread(new Consumer(p, _numberOfThreads)).start();
24             new Thread(new Producer(p, "passenger" + i)).start();
25         }
26         // System.out.println("End of main");
27     }
28
29     static class Producer implements Runnable {
30
31         private Piper _piper;
32         private String _name;
33
34         public Producer(Piper p, String n) {
35             _piper = p;
36             _name = n;
37         }
38
39         public void run() {
40             try {
41                 for (int i = 0; i < NUM_OF_SEATS; i++)
42                     // fill the piper with passengers
43                     _piper.fillPlane(_name);
44             } catch (InterruptedException i) {
45                 System.err.println(i);
46             }
47         }
48     }
49
50     static class Consumer implements Runnable {
```

```
51
52          private Piper _Piper;
53          private int _numberOfThreads;
54
55          public Consumer(Piper p, int numberOfThreads) {
56
57              _numberOfThreads = numberOfThreads;
58              _Piper = p;
59          }
60
61          public void run() {
62
63              try {
64                  for (int i = 0; i < NUM_OF_SEATS; i++)
65                      // empty the plane
66                      _Piper.emptyPlane();
67
68              } catch (InterruptedException i) {
69                  System.err.println(i);
70              }
71          }
72      }
73  }
```

Piper.java

```
1   package fse2006piper;
2
3   import java.io.*;
4   import java.lang.*;
5
6   public class  Piper {
7
8       static int NUM_OF_SEATS = 2;  // so capacity = number_of_seats -1
9       private int _first, _last;
10      private String[] _passengers;
11      private String _fileName;
12      static int _fillCount;
13      static int  _emptyCount;
14
15      public Piper(String fileName, int fillCount, int numSeats) {
16
17          _first = 0;
18          _last = 0;
19          NUM_OF_SEATS = numSeats;
20          _passengers = new String[NUM_OF_SEATS];
21          _fileName = fileName;
22          _fillCount = _emptyCount = fillCount;
23          setFile();
```

```
24      }
25
26      public void setFile() {
27      }
28
29      public synchronized void fillPlane( String name) throws InterruptedException {
30
31          // BUG - should be while, not if !!!
32          if (( _last + 1) % NUM_OF_SEATS == _first)
33              this.wait();
34
35           _passengers[_last] = name; //load passenger to plane
36
37          //close the file, no more passengers
38          synchronized(this) { if(--_fillCount == 0) closeFile(); }
39
40          _last = (_last + 1) % NUM_OF_SEATS;
41          this.notifyAll();
42      }
43
44      public synchronized String emptyPlane() throws InterruptedException {
45
46          while ( _first == _last)
47              this.wait();
48
49          String  name = _passengers[_first]; //get passenger off the plane
50          synchronized(this){ this._emptyCount--; }
51
52          _first = ( _first + 1) % NUM_OF_SEATS;
53          this.notifyAll();
54
55          return name;
56      }
57
58      public void closeFile() {
59      }
60  }
```

Piper.atom.java

```
1   import java.io.*;
2   import java.lang.*;
3
4   public class  Piper {
5       static int NUM_OF_SEATS = 2;  // so capacity = number_of_seats -1
6       private int _first, _last;
7       private String[] _passengers;
8       private String _fileName;
9       static int _fillCount;
```

```
10      static int  _emptyCount;
11
12      public Piper(String fileName, int fillCount, int numSeats) {
13          _first = 0;
14          _last = 0;
15          NUM_OF_SEATS = numSeats;
16          _passengers = new String[NUM_OF_SEATS];
17          _fileName = fileName;
18          _fillCount = _emptyCount = fillCount;
19          setFile();
20      }
21
22      public void setFile()
23      { }
24
25      private boolean is_full() {
26          atomic {
27              return ( _last + 1) % NUM_OF_SEATS == _first;
28          }
29      }
30
31      private boolean is_empty() {
32          atomic {
33              return _first == _last;
34          }
35      }
36
37      private void load_passenger(String name)
38      {
39          atomic {
40              _passengers[_last] = name;   // load passenger to plane
41
42              if(--_fillCount == 0)
43                  closeFile(); //close the file, no more passengers
44
45              _last = (_last + 1) % NUM_OF_SEATS;
46          }
47      }
48
49      public void fillPlane(String name)
50          throws InterruptedException {
51          // BUG - should be while, not if !!!
52          if (is_full())
53              synchronized(this) {
54                  this.wait();
55              }
56
57          load_passenger(name);
58
59          synchronized(this) {
```

```
60              this.notifyAll();
61          }
62      }
63
64      private String unload_passenger() {
65          atomic {
66              String name=_passengers[_first];
67
68              this._emptyCount--;
69
70              _first = ( _first + 1) % NUM_OF_SEATS;
71
72              return name;
73          }
74      }
75
76      public String emptyPlane()
77          throws InterruptedException {
78          while (is_empty())
79              synchronized(this) {
80                  this.wait();
81              }
82
83          // unload passenger off the plane
84          String name=unload_passenger();
85
86          synchronized(this) {
87              this.notifyAll();
88          }
89
90          return name;
91      }
92
93      public void closeFile()
94      { }
95  }
```

A.1.11 Clean

This test is discussed in Section 3.6.11, on Page 51.

Main.java

```
1   package fse2006clean;
2
3   public class Main {
4       static int iFirstTask = 1;
5       static int iSecondTask = 1;
6       static int iterations = 12;
```

```
7
8       public void run() {
9           Event new_event1 = new Event();
10          Event new_event2 = new Event();
11
12          for (int i = 0; i < iFirstTask; i++)
13              new FirstTask(new_event1, new_event2, iterations).start();
14          for (int i = 0; i < iSecondTask; i++)
15              new SecondTask(new_event1, new_event2, iterations).start();
16      }
17
18      public static void main(String[] args) {
19          if (args.length < 3) {
20              System.out.println("ERROR: Expected 3 parameters");
21          } else {
22              iFirstTask = Integer.parseInt(args[0]);
23              iSecondTask = Integer.parseInt(args[1]);
24              iterations = Integer.parseInt(args[2]);
25              Main t = new Main();
26              t.run();
27          }
28      }
29  }
```

FirstTask.java

```
1   package fse2006clean;
2
3   public class FirstTask extends Thread {
4       int iterations;
5       Event event1, event2;
6
7       public FirstTask(Event e1, Event e2, int iterations) {
8           this.event1 = e1;
9           this.event2 = e2;
10          this.iterations = iterations;
11      }
12
13      public void run() {
14          int count = 0;
15          count = event1.count;
16          for (int i = 0; i < iterations; i++) {
17              event1.waitForEvent(count);
18              count = event1.count;
19              event2.signal_event();
20          }
21      }
22  }
```

SecondTask.java

```
1  package fse2006clean;
2
3  public class SecondTask extends Thread {
4      int iterations;
5      Event event1, event2;
6
7      public SecondTask(Event e1, Event e2, int iterations) {
8          this.event1 = e1;
9          this.event2 = e2;
10         this.iterations = iterations;
11     }
12
13     public void run() {
14         int count = 0;
15         count = event2.count;
16         for (int i = 0; i < iterations; i++) {
17             event1.signal_event();
18             event2.waitForEvent(count);
19             count = event2.count;
20         }
21     }
22 }
```

Event.java

```
1  package fse2006clean;
2
3  public class Event {
4      public int count = 0;
5
6      public synchronized void waitForEvent(int remote_count) {
7          if (remote_count == count)
8              try {
9                  wait();
10             } catch (InterruptedException e) {
11             }
12     }
13
14     public synchronized void signal_event() {
15         count = (count + 1) % 100;
16         notifyAll();
17     }
18 }
```

Main.atom.java

```java
public class Main {
    static int iFirstTask = 1;
    static int iSecondTask = 1;
    static int iterations = 12;

    public void run() {
        Event new_event1 = new Event(1);
        Event new_event2 = new Event(2);
        Thread[] threads = new Thread[iFirstTask + iSecondTask];
        int j = 0;

        for (int i = 0; i < iFirstTask; i++) {
            threads[j++] = new FirstTask(new_event1, new_event2, iterations, i);
            threads[j - 1].start();
        }

        for (int i = 0; i < iSecondTask; i++) {
            threads[j++] = new SecondTask(new_event1, new_event2, iterations, i
                    + 8 + iFirstTask);
            threads[j - 1].start();
        }

        for (Thread th : threads)
            try {
                th.join();
            } catch (Exception e) {
            }

        System.out.println("c1: " + new_event1.get_count());
        System.out.println("c2: " + new_event2.get_count());
    }

    public static void main(String[] args) {
        if (args.length == 3) {
            iFirstTask = Integer.parseInt(args[0]);
            iSecondTask = Integer.parseInt(args[1]);
            iterations = Integer.parseInt(args[2]);

            System.out.println("its: " + iterations);
        }

        Main t = new Main();
        t.run();
    }
}
```

FirstTask.atom.java

```java
public class FirstTask extends Thread {
```

```
    int iterations;
    Event event1, event2;
    int t;

    public FirstTask(Event e1, Event e2, int iterations, int t) {
        this.t = t;
        this.event1 = e1;
        this.event2 = e2;
        this.iterations = iterations;
    }

    public void run() {
        int count = event1.count;
        for (int i = 0; i < iterations; i++) {
            event1.waitForEvent(count, t);
            count = event1.count;
            event2.signal_event(t);
        }
    }
}
```

SecondTask.atom.java

```
public class SecondTask extends Thread {
    int iterations;
    Event event1, event2;
    int t;

    public SecondTask(Event e1, Event e2, int iterations, int t) {
        this.t = t;
        this.event1 = e1;
        this.event2 = e2;
        this.iterations = iterations;
    }

    public void run() {
        int count = event2.count;
        for (int i = 0; i < iterations; i++) {
            event1.signal_event(t);
            event2.waitForEvent(count, t);
            count = event2.count;
        }
    }
}
```

Event.atom.java

```
public class Event {
```

```
2       public int count = 0;
3       private int ev;
4
5       public Event(int e) {
6           ev = e;
7       }
8
9       public int get_count() {
10          atomic {
11              return count;
12          }
13      }
14
15      public boolean is_in_sync(int remote) {
16          atomic {
17              return remote == count;
18          }
19      }
20
21      public void waitForEvent(int remote_count, int i) {
22          if (is_in_sync(remote_count))
23              try {
24                  System.out.println("ev" + ev + " " + "th" + i + " waiting");
25                  synchronized (this) {
26                      wait();
27                  }
28                  System.out.println("ev" + ev + " " + "th" + i + " unblocked");
29              } catch (InterruptedException e) {
30              }
31          else
32              System.out.println("ev" + ev + " " + "th" + i
33                      + " unsynchronized counts");
34      }
35
36      private void inc_count() {
37          atomic {
38              count = (count + 1) % 100;
39          }
40      }
41
42      public void signal_event(int i) {
43          inc_count();
44
45          synchronized (this) {
46              notifyAll();
47          }
48
49          System.out.println("ev" + ev + " " + "th" + i + " notifyed");
50      }
51  }
```

A.1.12 Allocate Vector

This test is discussed in Section 3.6.12, on Page 52.

Test.java

```
1  package Bug;
2
3  import java.io.*;
4
5  /**
6   * class Test: Used to test class AllocationVector.
7   */
8  public class Test {
9      /**
10      * Indicates number of threads runs to perform.
11      */
12     private static final int runsNum = 1;
13
14     /**
15      * MAIN METHOD. Gets from command-line: 1. Name of output file. 2.
16      * Concurrency Parameter (little,average,lot).
17      *
18      * @param args
19      *             command-line arguments as written above.
20      */
21
22     public static void main(String[] args) {
23         for (int i = 0; i < runsNum; i++) {
24             runTest(args);
25         }
26     }
27
28     /**
29      * Gets from 'args': 1. Name of output file. 2. Concurrency Parameter
30      * (little,average,lot).
31      *
32      * @param args
33      *             command-line arguments as written above.
34      */
35     public static void runTest(String[] args) {
36         AllocationVector vector = null;
37         TestThread1 Thread1 = null;
38         TestThread1 Thread2 = null;
39         int[] Thread1Result = null;
40         int[] Thread2Result = null;
41         FileOutputStream out = null;
42
43         /**
44          * Reading command-line arguments.
```

```
45          */
46         try {
47             if (args.length != 2) {
48                 throw new Exception();
49             }
50
51             // Opening output file with name 'args[0]' for append write.
52             out = new FileOutputStream(args[0], false);
53
54             // Checking concurrency parameter correctness.
55             if ((args[1].compareTo("little") != 0)
56                     && (args[1].compareTo("average") != 0)
57                     && (args[1].compareTo("lot") != 0)) {
58                 throw new Exception();
59             }
60         } catch (Exception e) {
61             System.err.println("Invalid command-line arguments...");
62             System.exit(1);
63         }
64
65         /**
66          * If here, then command-line arguments are correct. Therefore,
67          * proceeding according to the concurrency parameter value.
68          */
69         // Setting threads run configuration according to concurrency parameter.
70         if (args[1].compareTo("little") == 0) {
71             vector = new AllocationVector(20000);
72             Thread1Result = new int[1000];
73             Thread2Result = new int[1000];
74         } else if (args[1].compareTo("average") == 0) {
75             vector = new AllocationVector(10000);
76             Thread1Result = new int[2000];
77             Thread2Result = new int[2000];
78         } else if (args[1].compareTo("lot") == 0) {
79             vector = new AllocationVector(5000);
80             Thread1Result = new int[5000];
81             Thread2Result = new int[5000];
82         }
83
84         // Creating threads, starting their run and waiting till they finish.
85         Thread1 = new TestThread1(vector, Thread1Result);
86         Thread2 = new TestThread1(vector, Thread2Result);
87         Thread1.start();
88         for (int i = 0; i < 100000; i++)
89             ; // "Pause" between threads run to try "hide"
90         // the BUG.
91         Thread2.start();
92         try {
93             Thread1.join();
94             Thread2.join();
```

```
95          } catch (InterruptedException e) {
96              System.err.println("Error joining threads...");
97              System.exit(1);
98          }
99
100         // Checking correctness of threads run results and printing the
101         // according tuple to output file.
102         try {
103             if (Thread1Result[0] == -2) {
104                 out
105                         .write("<Test, Thread1 tried to allocate block which is
                                allocated, weak-reality (Two stage access)>\n"
106                                 .getBytes());
107             } else if (Thread1Result[0] == -3) {
108                 out
109                         .write("<Test, Thread1 tried to free block which is free,
                                weak-reality (Two stage access)>\n"
110                                 .getBytes());
111             } else if (Thread2Result[0] == -2) {
112                 out
113                         .write("<Test, Thread2 tried to allocate block which is
                                allocated, weak-reality (Two stage access)>\n"
114                                 .getBytes());
115             } else if (Thread2Result[0] == -3) {
116                 out
117                         .write("<Test, Thread2 tried to free block which is free,
                                weak-reality (Two stage access)>\n"
118                                 .getBytes());
119             } else {
120                 out.write("<Test, correct-run, none>\n".getBytes());
121             }
122         } catch (IOException ex) {
123             System.err.println("Error writing to output file...");
124             System.exit(1);
125         }
126     }
127 }
```

TestThread1.java

```
1  package Bug;
2
3  /**
4   * class TestThread1: Used to run thread which allocates and frees blocks
5   * by given AllocationVector object.
6   */
7  public class TestThread1 extends Thread {
8      /**
9       * Reference to class AllocationVector object with which the thread
```

```
10      * will work.
11      */
12     private AllocationVector vector = null;
13
14     /**
15      * An array to which the resulting allocated blocks indexes will be
16      * stored. It's lenght indicates the number of accesses, which to
17      * perform to 'vector'.
18      */
19     private int[] resultBuf = null;
20
21     /**
22      * Constructor: Constructs thread which will work on class
23      * AllocationVector object 'vec', to which the thread will perform
24      * 'resBuf.length' accesses of allocation and 'resBuf.length' accesses
25      * of frees. The resulting allocated blocks indexes will be stored to
26      * 'resBuf'.
27      * @param vec
28      *            class AllocationVector object on which the thread will
29      *            work.
30      * resBuf Buffer for resulting allocated blocks indexes, which also
31      *            indicates the number of access which to perform to 'vec'.
32      */
33     public TestThread1(AllocationVector vec, int[] resBuf) {
34         if ((vec == null) || (resBuf == null)) {
35             System.err.println("Thread start error...");
36             System.exit(1);
37         }
38         vector = vec;
39         resultBuf = resBuf;
40     }
41
42     /**
43      * Perform 2 * 'resultBuf.length' accesses to 'vector', in the
44      * following way: 'resultBuf.length' blocks allocations.
45      * 'resultBuf.length' blocks frees. NOTE: If allocation/free block
46      * error occurs, function sets resultBuf[0] to -2/-3.
47      */
48     public void run() {
49         try {
50             // Allocating 'resultBuf.length' blocks.
51             for (int i = 0; i < resultBuf.length; i++) {
52                 resultBuf[i] = vector.getFreeBlockIndex();
53                 if (resultBuf[i] != -1) {
54                     vector.markAsAllocatedBlock(resultBuf[i]);
55                 }
56             }
57
58             // Freeing 'resultBuf.length' blocks.
59             for (int i = 0; i < resultBuf.length; i++) {
```

```
60                  if (resultBuf[i] != -1) {
61                      vector.markAsFreeBlock(resultBuf[i]);
62                  }
63              }
64          } catch (Exception e) {
65              if (e.getMessage().compareTo("Allocation") == 0) {
66                  resultBuf[0] = -2;
67              } else {
68                  resultBuf[0] = -3;
69              }
70          }
71      }
72  }
```

AllocationVector.java

```
1   package Bug;
2
3   /**
4    * class AllocationVector: Used to manage allocation and freeing of blocks. BUG
5    * DOCUMENTATION: There is a synchronization GAP between the methods
6    * "getFreeBlockIndex" and "markAsAllocatedBlock", in which anything can be
7    * done.
8    */
9   public class AllocationVector {
10      /**
11       * Character vector which holds information about allocated and free blocks,
12       * in the following way: if vector[i] == 'F' -> i-th block is free. if
13       * vector[i] == 'A' -> i-th block is allocated.
14       */
15      private char[] vector = null;
16
17      /**
18       * Constructor: Constructs AllocationVector for 'size' blocks, when all
19       * blocks are free.
20       *
21       * @param size
22       *             Size of AllocationVector.
23       */
24      public AllocationVector(int size) {
25          // Allocating vector of size 'size', when all blocks are assigned
26          // to free.
27          vector = new char[size];
28          for (int i = 0; i < size; i++) {
29              vector[i] = 'F';
30          }
31      }
32
33      /**
```

```
34      * Returns index of free block, if such exists. If no free block, then
35      * -1 is returned.
36      *
37      * @return Index of free block if such exists, else -1.
38      */
39     synchronized public int getFreeBlockIndex() {
40         int i;
41         int count;
42         int startIndex;
43         int interval;
44         int searchDirection;
45         double randomValue;
46         int[] primeValues = { 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
47                 37, 41, 43, 47, 53 };
48         randomValue = Math.random();
49 
50         // Choosing randomly start entry for search.
51         startIndex = (int) Math.floor((vector.length - 1) * randomValue);
52 
53         // Choosing randomly increment/decrement prime value.
54         interval = primeValues[(int) Math.floor((primeValues.length - 1)
55                 * randomValue)];
56 
57         // Choosing randomly search direction and starting the search from
58         // randomly
59         // choosen start entry in that direction with the randomly choosen
60         // interval.
61         if (randomValue > 0.5) {
62             // Searching forward.
63             for (i = startIndex, count = 0; (count < vector.length)
64                     && (vector[i] != 'F'); i = i + interval, i %= vector.length,
                        count++)
65                 ;
66         } else {
67             // Searching backward.
68             for (i = startIndex, count = 0; (count < vector.length)
69                     && (vector[i] != 'F'); count++) {
70                 i = i - interval;
71                 if (i < 0) {
72                     i = i + vector.length;
73                 }
74             }
75         }
76 
77         if (count == vector.length) {
78             return -1; // Indicates "no free block".
79         } else {
80             return i; // Returns the index of the found free block.
81         }
82     }
```

```
83
84      /**
85       * Marks i-th block as allocated.
86       *
87       * @param i
88       *            Index of block to allocate. NOTE: If allocating already
89       *            allocated block, then Exception is thrown.
90       */
91      synchronized public void markAsAllocatedBlock(int i) throws Exception {
92          if (vector[i] != 'A') {
93              vector[i] = 'A'; // Allocates i-th block.
94          } else {
95              throw new Exception("Allocation");
96          }
97      }
98
99      /**
100      * Marks i-th block as free.
101      *
102      * @param i
103      *            Index of block to free. NOTE: If freeing already free block,
104      *            then Exception is thrown.
105      */
106     synchronized public void markAsFreeBlock(int i) throws Exception {
107         if (vector[i] != 'F') {
108             vector[i] = 'F'; // Frees i-th block.
109         } else {
110             throw new Exception("Freeing");
111         }
112     }
113
114 }
```

TestThread1.atom.java

```
1  public class TestThread1 extends Thread {
2
3      private AllocationVector vector = null;
4
5      private int[] resultBuf = null;
6
7      public TestThread1(AllocationVector vec, int[] resBuf) {
8          if ((vec == null) || (resBuf == null)) {
9              System.err.println("Thread start error...");
10             System.exit(1);
11         }
12         vector = vec;
13         resultBuf = resBuf;
14     }
```

```
15
16      private void alloc_block(int i) throws Exception {
17          atomic {
18              resultBuf[i] = vector.getFreeBlockIndex();
19              if (resultBuf[i] != -1)
20                  vector.markAsAllocatedBlock(resultBuf[i]);
21          }
22      }
23
24      public void run() {
25          try {
26              // Allocating 'resultBuf.length' blocks.
27              for (int i = 0; i < resultBuf.length; i++)
28                  alloc_block(i);
29
30              // Freeing 'resultBuf.length' blocks.
31              for (int i = 0; i < resultBuf.length; i++) {
32                  if (resultBuf[i] != -1) {
33                      vector.markAsFreeBlock(resultBuf[i]);
34                  }
35              }
36          } catch (Exception e) {
37              if (e.getMessage().compareTo("Allocation") == 0) {
38                  resultBuf[0] = -2;
39              } else {
40                  resultBuf[0] = -3;
41              }
42          }
43      }
44  }
```

AllocationVector.atom.java

```
1   public class AllocationVector {
2       private char[] vector = null;
3
4       public AllocationVector(int size) {
5           vector = new char[size];
6           for (int i = 0; i < size; i++) {
7               vector[i] = 'F';
8           }
9       }
10
11      public int getFreeBlockIndex() {
12          atomic {
13              int i;
14              int count;
15              int startIndex;
16              int interval;
```

```
17              int searchDirection;
18              double randomValue;
19              int[] primeValues = { 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
                     43, 47, 53 };
20              randomValue = Math.random();
21
22              startIndex = (int)Math.floor((vector.length - 1) * randomValue);
23              interval = primeValues[(int)Math.floor((primeValues.length - 1) *
                    randomValue)];
24
25              if (randomValue > 0.5) {
26                  for (i = startIndex,count = 0; (count < vector.length) &&
27                            (vector[i] != 'F')
28                            ; i = i + interval, i %= vector.length, count++);
29              } else {
30                  for (i = startIndex,count = 0; (count < vector.length) &&
31                            (vector[i] != 'F')
32                            ; count++) {
33                      i = i - interval;
34                      if (i < 0) {
35                          i = i + vector.length;
36                      }
37                  }
38              }
39
40              if (count == vector.length) {
41                  return -1; // Indicates "no free block".
42              }
43              else {
44                  return i; // Returns the index of the found free block.
45              }
46          }
47      }
48
49      public void markAsAllocatedBlock(int i) throws Exception {
50          atomic {
51              if (vector[i] != 'A') {
52                  vector[i] = 'A'; // Allocates i-th block.
53              } else {
54                  throw new Exception("Allocation");
55              }
56          }
57      }
58
59      public void markAsFreeBlock(int i) throws Exception {
60          atomic {
61              if (vector[i] != 'F') {
62                  vector[i] = 'F'; // Frees i-th block.
63              } else {
64                  throw new Exception("Freeing");
```

```
65                  }
66              }
67          }
68  }
```

A.1.13 LeeTM

This test is discussed in Section 3.6.13, on Page 54.

Due to the excessively large extent of the source code, we have omitted most of it, and present only the parts of the TM files which differ from the original version. The complete source code for the Lee-TM benchmark [AKW+08] may be obtained from http://intranet.cs.man.ac.uk/apt/projects/TM/LeeBenchmark.

LeeRouter.atom.java

```
1   // (...)
2
3       public WorkQueue getNextTrack() {
4           atomic {
5               if(work.next != null) {
6                   return work.deQueue();
7               }
8           }
9           return null;
10      }
11
12      public boolean layNextTrack(WorkQueue q, int [][][]tempg) {
13          // start transaction
14          boolean done = false;
15          atomic {
16              done = connect(q.x1, q.y1, q.x2, q.y2, q.nn, tempg, grid);
17              if(DEBUG && done) {
18                  debugQueue.next = debugQueue.enQueue(q);
19              }
20          }
21          return done;
22          // end transaction
23      }
24
25  // (...)
```

LeeThread.atom.java

```
1   // (...)
2
3   public class LeeThread extends Thread {
4
```

```
5     public static boolean stop = false;
6     boolean finished = false;
7     public boolean sampleNow = false;
8     public boolean doneSample = true;
9    // public static long totalTracks=0;
10    public static long totalLaidTracks=0;
11    public long myLaidTracks=0;
12    private static Object lock = new Object();
13    public String hardware = System.getenv("HOSTNAME");
14    protected static ThreadLocal<ThreadState> _threadState = new ThreadLocal<
          ThreadState>() {
15        protected ThreadState initialValue() {
16            ThreadState ts;
17            atomic {
18                ts = new ThreadState();
19            }
20            return ts;
21        }
22    };
23    static ThreadLocal<Thread> _thread = new ThreadLocal<Thread>() {
24        protected Thread initialValue() {
25            Thread t;
26            atomic {
27                t = null;
28            }
29            return t;
30        }
31    };
32
33    // (...)
34
35    protected static void collectStatistics(ThreadState threadState) {
36        // collect statistics
37        atomic {
38            totalLaidTracks+=threadState.myLaidTracks;
39            threadState.reset();  // set up for next iteration
40        }
41    }
42
43    // (...)
44 }
```

A.2 Tests with High-Level Anomalies

The detection of high-level anomalies is addressed in Chapter 3.

A.2.1 Test: Connection

This test is discussed in Section 4.6.1, on Page 75.

Connection.atom.java

```
1  package bba08;
2
3  import java.io.IOException;
4  import java.net.Socket;
5
6  class Connection {
7
8      private final Counter counter;
9      private Socket socket;
10
11     public Connection() {
12         this.socket = null;
13         this.counter = new Counter();
14     }
15
16     // BCT
17     public void connect() {
18         this.socket = new Socket();
19     }
20
21     public void disconnect() throws IOException {
22         atomic {
23             this.socket.close();
24             this.socket = null;
25         }
26         this.counter.reset();
27     }
28
29     public boolean isConnected() {
30         atomic {
31             return (this.socket != null);
32         }
33     }
34
35     public void send(String msg) throws IOException {
36         atomic {
37             this.socket.getOutputStream().write(msg.getBytes());
38             this.counter.increment();
39         }
40     }
41
42 }
```

Counter.atom.java

```
1  package bba08;
2
```

```
3  public class Counter {
4
5      int n = 0;
6
7      public void reset() {
8          atomic {
9              n = 0;
10         }
11     }
12
13     public void increment() {
14         atomic {
15             n = n + 1;
16         }
17     }
18
19 }
```

GUI.atom.java

```
1  package bba08;
2
3  import java.io.IOException;
4  import java.util.Random;
5
6  class GUI extends Thread {
7
8      Connection connection;
9
10     public GUI(Connection connection) {
11         this.connection = connection;
12     }
13
14     public static void main(String args[]) {
15         Connection connection = new Connection();
16         connection.connect();
17         for (int i = 0; i < 10; i++)
18             new GUI(connection).start();
19     }
20
21     boolean trySendMsg(String msg) throws IOException {
22         if (this.connection.isConnected()) {
23             this.connection.send(msg);
24             return true;
25         } else {
26             return false;
27         }
28     }
29
```

```
30      public void run() {
31          Random rand = new Random();
32          int command;
33          try {
34              do {
35                  command = rand.nextInt(10);
36                  if (command == 0)
37                      this.connection.disconnect();
38                  else {
39                      byte[] bytes = new byte[rand.nextInt(10)];
40                      rand.nextBytes(bytes);
41                      String msg = new String(bytes);
42                      this.trySendMsg(msg);
43                  }
44              } while (command != 0);
45          } catch (IOException e) {
46              e.printStackTrace();
47          }
48      }
49
50  }
```

A.2.2 Test: Coordinates'03

This test is discussed in Section 4.6.2, on Page 76.

Coord.atom.java

```
1   package cartho_03.coord;
2
3   public class Coord {
4       private double x, y;
5
6       public Coord(double px, double py) {
7           x = px;
8           y = py;
9       }
10
11      double getX() {
12          atomic {
13              return x;
14          }
15      }
16
17      double getY() {
18          atomic {
19              return y;
20          }
21      }
```

```
22
23      Coord getXY() {
24          atomic {
25              return new Coord(x, y);
26          }
27      }
28
29      void setX(double px) {
30          atomic {
31              x = px;
32          }
33      }
34
35      void setY(double py) {
36          atomic {
37              y = py;
38          }
39      }
40
41      void setXY(Coord c) {
42          atomic {
43              x = c.x;
44              y = c.y;
45          }
46      }
47  }
```

Main.atom.java

```
1   package cartho_03.coord;
2
3   public class Main {
4
5       private Coord c;
6
7       public static void main(String[] args) {
8           new Main().execute();
9       }
10
11      public void execute () {
12          this.c = new Coord(0, 0);
13          new T1().start();
14          new T2().start();
15          new T3().start();
16          new T4().start();
17      }
18
19      class T1 extends Thread {
20          public void run() {
```

```
21              Coord d1 = new Coord(1, 2);
22              c.setXY(d1);
23          }
24      }
25
26      class T2 extends Thread {
27          public void run() {
28              double x2 = c.getX();
29              System.out.println(x2);
30          }
31      }
32
33      class T3 extends Thread {
34          public void run() {
35              double x3 = c.getX();
36              double y3 = c.getY();
37              System.out.println(x3);
38              System.out.println(y3);
39          }
40      }
41
42      class T4 extends Thread {
43          public void run() {
44              double x4 = c.getX();
45              System.out.println(x4);
46              Coord d4 = c.getXY();
47              x4 = d4.getX();
48              double y4 = d4.getY();
49              System.out.println(x4);
50              System.out.println(y4);
51          }
52      }
53
54  }
```

A.2.3 Test: Local Variable

This test is discussed in Section 4.6.3, on Page 77.

Local.atom.java

```
1  package cartho_03.local;
2
3  public class Local extends Thread {
4
5      static Cell x = new Cell();
6
7      public static void main(String[] args) {
8          new Local().start();
```

```
9           new Local().start();
10      }
11
12      public void run() {
13          int tmp;
14          atomic {
15              tmp = x.getValue();
16          }
17          tmp++;
18          atomic {
19              x.setValue(tmp);
20          }
21      }
22
23      static class Cell {
24          int n = 0;
25
26          int getValue() {
27              return n;
28          }
29
30          void setValue(int x) {
31              n = x;
32          }
33      }
34
35  }
```

A.2.4 Test: NASA

This test is discussed in Section 4.6.4, on Page 78.

Main.atom.java

```
1   package cartho_03.nasa;
2
3   public class Main {
4
5       public static void main(String[] args) {
6           Cell[] table = new Cell[100];
7           Object[] system_state = new Object[100];
8
9           new Daemon(table, system_state).start();
10
11          new Task(table).start();
12          new Task(table).start();
13          new Task(table).start();
14
15      }
```

```
16
17 }
```

Task.atom.java

```
1  package cartho_03.nasa;
2
3  public class Task extends Thread {
4
5      public Cell[] table;
6
7      public Task(Cell[] table) {
8          super();
9          this.table = table;
10     }
11
12     public void run() {
13
14         int N = 42;
15
16         Object v = new Object();
17
18         atomic {
19             table[N].value = v;
20         }
21
22         /* achieve property */
23
24         atomic {
25             table[N].achieved = true;
26         }
27     }
28 }
```

Daemon.atom.java

```
1  package cartho_03.nasa;
2
3  public class Daemon extends Thread {
4      public Cell[] table;
5      public Object[] system_state;
6
7      public Daemon(Cell[] table, Object[] system_state) {
8          super();
9          this.table = table;
10         this.system_state = system_state;
11     }
12
```

```
13      public void run() {
14
15          int N = 42;
16
17          while (true) {
18              atomic {
19                  if (table[N].achieved && system_state[N] != table[N].value)
20                      issueWarning();
21              }
22          }
23      }
24
25      private void issueWarning() {
26          throw new RuntimeException("PANIC!!!");
27      }
28  }
```

Cell.atom.java

```
1   package cartho_03.nasa;
2
3   public class Cell {
4
5       public Object value;
6
7       public boolean achieved;
8
9   }
```

A.2.5 Test: Coordinates'04

This test is discussed in Section 4.6.5, on Page 79.

CoordMain.atom.java

```
1   package cartho_04.coord;
2
3   public class CoordMain extends Thread {
4
5       Coord coord;
6
7       public static void main(String[] args) {
8           Coord c = new Coord();
9           new CoordMain(c).start();
10          new CoordMain(c).start();
11      }
12
13      public CoordMain(Coord coord) {
```

```
14          this.coord = coord;
15      }
16
17      public void run() {
18          swap();
19          reset();
20      }
21
22      public void swap() {
23          int oldX;
24          atomic {
25              oldX = coord.x;
26              coord.x = coord.y; // swap X
27              coord.y = oldX; // swap Y
28          }
29      }
30
31      public void reset() {
32          atomic {
33              coord.x = 0;
34          } // inconsistent state (0, y)
35          atomic {
36              coord.y = 0;
37          }
38      }
39
40      static class Coord {
41          int x, y;
42      }
43  }
```

A.2.6 Test: Buffer

This test is discussed in Section 4.6.6, on Page 80.

Fig5.atom.java

```
1   package cartho_04.fig5;
2
3   public class Fig5 extends Thread {
4       Buffer buffer;
5
6       public Fig5(Buffer buffer) {
7           super();
8           this.buffer = buffer;
9       }
10
11      public static void main(String[] args) {
12          Buffer buffer = new Buffer();
```

```
13          new Fig5(buffer).start();
14          new Fig5(buffer).start();
15          new Fig5(buffer).start();
16      }
17
18      public void run() {
19          work();
20      }
21
22      public void work() {
23          int value, fdata;
24          while (true) {
25              atomic {
26                  value = buffer.next();
27              }
28              fdata = f(value); // long computation
29              atomic { // Data flow from previous block!
30                  buffer.add(fdata); // However, the program is correct because
31              } // the buffer protocol ensures that the
32          } // returned data remains thread-local.
33      }
34
35      int f(int x) {
36          return x * x;
37      }
38
39      static class Buffer {
40          // I know it doesn't make sense, we're just trying to simulate reads and
                writes
41          int cell = 0;
42          int head = 0;
43          int tail = 0;
44
45          public int next() {
46              int res;
47              atomic {
48                  head = head + 1;
49                  res = cell;
50              }
51              return res;
52          }
53
54          public void add(int x) {
55              atomic {
56                  tail = tail + 1;
57                  cell = x;
58              }
59          }
60
61      }
```

```
62 }
```

A.2.7 Test: Double Check

This test is discussed in Section 4.6.7, on Page 81.

Fig6.atom.java

```
1  package cartho_04.fig6;
2
3  public class Fig6 extends Thread {
4
5      static Cell shared;
6
7      public static void main(String[] args) {
8          new Fig6().start();
9          new Fig6().start();
10         new Fig6().start();
11     }
12
13     public void run() {
14         do_transaction();
15     }
16
17     public void do_transaction() {
18         int value, fdata;
19         boolean done = false;
20         while (!done) {
21             atomic {
22                 value = shared.field;
23             }
24             fdata = f(value); // long computation
25             atomic {
26                 if (value == shared.field) {
27                     shared.field = fdata;
28                 // The usage of the locally computed fdata is safe because
29                 // the shared value is the same as during the computation.
30                 // Our algorithm and previous atomicity-based approaches
31                 // report an error (false positive).
32                     done = true;
33                 }
34             }
35         }
36     }
37
38     int f(int x) {
39         return x * x;
40     }
41
```

```
42      static class Cell {
43          public int field;
44      }
45
46  }
```

A.2.8 Test: StringBuffer

This test is discussed in Section 4.6.8, on Page 82.

MyStringBuffer.atom.java

```
1   package ff04.stringBuffer;
2
3   /*
4    * Simulate java.lang.StringBuffer
5    */
6   public final class MyStringBuffer {
7
8       private java.lang.StringBuffer buffer;
9
10      public MyStringBuffer(String string) {
11          this.buffer = new StringBuffer(string);
12      }
13
14      public static void main(String args[]) {
15          MyStringBuffer ham = new MyStringBuffer("ham");
16          MyStringBuffer burger = new MyStringBuffer("burger");
17          ham.append(burger);
18          System.out.println(ham);
19      }
20
21      public MyStringBuffer append(MyStringBuffer other) {
22
23          int len = other.length();
24
25          // ...other threads may change sb.length(),
26          // ...so len does not reflect the length of 'other'
27
28          char[] value = new char[len];
29          other.getChars(0, len, value, 0);
30
31          // ...
32
33          return this;
34      }
35
36      public int length() {
37          int len;
```

```
38          atomic {
39              len = this.buffer.length();
40          }
41          return len;
42      }
43
44      public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) {
45          atomic {
46              this.buffer.getChars(srcBegin, srcEnd, dst, dstBegin);
47          }
48      }
49
50
51  }
```

A.2.9 Test: Account

This test is discussed in Section 4.6.9, on Page 82.

Main.atom.java

```
1   package pg03.account;
2
3   public class Main {
4
5       static Account a;
6       public static void main(String[] args) {
7           a = new Account();
8           new Update().start();
9           new Update().start();
10      }
11
12  }
```

Account.atom.java

```
1   package pg03.account;
2
3   public class Account {
4
5       int balance;
6
7       int read () {
8           atomic {
9               return balance;
10          }
11      }
12
```

```
13      //int update (int a) {
14      void update (int a) {
15          int tmp = read();
16          atomic {
17              balance = tmp + a;
18          }
19      }
20  }
```

Update.atom.java

```
1   package pg03.account;
2
3   public class Update extends Thread {
4       public void run() {
5           //Example.a.update(123);
6           Main.a.update(123);
7       }
8   }
```

A.2.10 Test: Jigsaw

This test is discussed in Section 4.6.10, on Page 83.

ResourceStoreManager.atom.java

```
1   package pg03.jigsaw;
2
3   import java.util.HashMap;
4   import java.util.Map;
5
6   public class ResourceStoreManager {
7
8       boolean closed = false;
9       Map entries = new HashMap();
10
11      void checkClosed() {
12          atomic {
13              if (closed)
14                  throw new RuntimeException();
15          }
16      }
17
18      public static void main(String args[]) {
19          ResourceStoreManager rsm = new ResourceStoreManager();
20          new Runner(rsm).start();
21          new Runner(rsm).start();
22          new Runner(rsm).start();
```

```
23      }
24
25      // not synched!!! HLDR, but we get a false negative!
26      ResourceStore loadResourceStore() {
27          checkClosed(); // R(closed)
28          Entry e = lookupEntry(new Object()); // R(entries), W(entries)
29          return e.getStore();
30      }
31
32      synchronized Entry lookupEntry(Object key) {
33          Entry e;
34          atomic {
35              e = (Entry) entries.get(key);
36              if (e == null) {
37                  e = new Entry();
38                  entries.put(key, e);
39                  entries=null;
40              }
41          }
42          return e;
43      }
44
45      void shutdown() {
46          atomic {
47              entries.clear();
48              closed = true;
49          }
50      }
51
52      public class ResourceStore {
53
54      }
55
56      public class Entry {
57          public ResourceStore getStore() {
58              return null;
59          }
60      }
61
62      static public class Runner extends Thread {
63          private ResourceStoreManager rsm;
64          public Runner(ResourceStoreManager rsm) {
65              this.rsm = rsm;
66          }
67          public void run() {
68              System.out.println(rsm.loadResourceStore());
69              rsm.shutdown();
70          }
71      }
72  }
```

A.2.11 Test: Over-reporting

This test is discussed in Section 4.6.11, on Page 84.

MapClient.atom.java

```
package pg03.over;

public class MapClient extends Thread {
    static Map m;

    public static void main(String args[]) {
        m = new Map();
        new MapClient().start();
        new MapClient().start();
    }

    public void run() {
        // lazy initialization
        m.init();
        m.get(new Object());
        // ...
    }
}
```

Map.atom.java

```
package pg03.over;

import java.util.Random;

public class Map {

    Object[] keys, values;
    volatile boolean init_done = false;

    void init() {
        if (!init_done)
            atomic {
                init_done = true;
                // update keys and values
                Random rand = new Random();
                keys = new Object[rand.nextInt(10) + 1];
                values = new Object[keys.length];
            }
    }

    Object get(Object key) {
        Object res = null;
```

```
23          atomic {
24              // read keys and values
25              for (int i = 0 ; i < keys.length ; i++) {
26                  if (key.equals(keys[i])) {
27                      res = values[i];
28                      break;
29                  }
30              }
31          }
32          return res;
33      }
34  }
```

A.2.12 Test: Underreporting

This test is discussed in Section 4.6.12, on Page 86.

Main.atom.java

```
1  package pg03.under;
2
3  public class Main extends Thread {
4
5      static Counter c;
6
7      public static void main(String[] args) {
8          c = new Counter();
9          new Main().start();
10         new Main().start();
11     }
12
13     public void run() {
14         int i = c.inc(0);
15         c.inc(i);
16     }
17
18 }
```

Counter.atom.java

```
1  package pg03.under;
2
3  public class Counter {
4
5      int i;
6
7      int inc(int a) {
8          int res;
```

```
9           atomic {
10              i += a;
11              res = i;
12          }
13          return res;
14      }
15  }
```

A.2.13 Test: Allocate Vector

This test is discussed in Section 4.6.13, on Page 87.

The subject for this test is exactly the same as the TM version of the *Allocate Vector* test for low-level datarace detection. Therefore, we have refrained from duplicating it. Refer to Section A.1.12 on Page 128 for the source code.

A.2.14 Test: Knight Moves

This test is discussed in Section 4.6.14, on Page 88.

Main.atom.java

```
1   package orium.knight;
2
3   import java.awt.Point;
4   import java.util.Random;
5
6   public class Main {
7       public static void main(String[] args) {
8           Random rnd = new Random();
9           Point knight = new Point(Math.abs(rnd.nextInt()) % KnightMoves.WIDTH,
10                  Math.abs(rnd.nextInt()) % KnightMoves.WIDTH);
11          Point prey = new Point(Math.abs(rnd.nextInt()) % KnightMoves.WIDTH,
12                  Math.abs(rnd.nextInt()) % KnightMoves.WIDTH);
13
14          KnightMoves km = new KnightMoves(knight, prey);
15          int s;
16          int cs;
17
18          System.out.println("Knight @ " + knight.toString());
19          System.out.println("Prey @ " + prey.toString());
20
21          km.solve();
22
23          System.out.println("Moves: " + (s = km.get_moves()));
24
25          km.solve_correct();
26
27          System.out.println("Correct Moves: " + (cs = km.get_moves()));
```

```
28
29          if (cs != s)
30              System.out.println("err");
31      }
32  }
```

KnightMoves.atom.java

```
1   package orium.knight;
2
3   import java.awt.Point;
4
5   public class KnightMoves {
6       public static final int WIDTH = 8;
7       private Point knight;
8       private Point prey;
9       private int solution[][];
10
11      public KnightMoves(Point knight, Point prey) {
12          solution = new int[WIDTH][WIDTH];
13          this.knight = knight;
14          this.prey = prey;
15      }
16
17      private void reset_solution() {
18          for (int i=0; i < WIDTH; i++)
19              for (int j=0; j < WIDTH; j++)
20                  solution[i][j]=Integer.MAX_VALUE;
21      }
22
23      public void solve() {
24          /* This would be much faster with breadth first search */
25          reset_solution();
26          Solver s = new Solver(this, (Point)knight.clone(), 0);
27          s.start();
28          try { s.join(); } catch (Exception e) {}
29      }
30
31      public Point get_knight() {
32          return knight;
33      }
34
35      public Point get_prey() {
36          return prey;
37      }
38
39      public int get_solution(Point p) {
40          int x;
41          atomic {
```

```
42              x = solution[p.x][p.y];
43          }
44          return x;
45      }
46
47      public void set_solution(Point p, int m) {
48          atomic {
49              solution[p.x][p.y] = m;
50          }
51      }
52
53      public int get_moves() {
54          return get_solution(prey);
55      }
56
57      public void solve_correct() {
58          reset_solution();
59          new SolverCorrect(this, (Point)knight.clone(), 0).go();
60      }
61  }
```

Solver.atom.java

```
1   package orium.knight;
2
3   import java.awt.Point;
4   import java.lang.Thread;
5   import java.util.Random;
6
7   public class Solver extends Thread {
8       private KnightMoves km;
9       private Point me;
10      private int moves;
11
12      public Solver(KnightMoves km, Point me, int moves) {
13          this.km = km;
14          this.me = me;
15          this.moves = moves;
16      }
17
18      private int check_and_set_solution() {
19          /* Check if other thread has a better solution */
20          if (km.get_solution(me) <= moves)
21              return -1;
22
23          km.set_solution(me, moves);
24          return 0;
25      }
26
```

```
27      public void run() {
28          Point[] next = new Point[8];
29          int c = 0;
30          int m[] = { 1, -1 };
31          Solver[] solver = new Solver[8];
32          int s_c = 0;
33          Random rnd = new Random();
34
35          if (check_and_set_solution() < 0)
36              return;
37
38          if (me.equals(km.get_prey()))
39              return;
40
41          for (int i = 1; i <= 2; i++)
42              for (int j = 1; j <= 2; j++)
43                  if (i != j)
44                      for (int k = 0; k < 2; k++)
45                          for (int l = 0; l < 2; l++)
46                              next[c++] = new Point(i * m[k] + me.x, j * m[l]
47                                      + me.y);
48
49          for (Point n : next)
50              if (n.x >= 0 && n.y >= 0 && n.x < KnightMoves.WIDTH
51                      && n.y < KnightMoves.WIDTH) {
52                  solver[s_c] = new Solver(km, n, moves + 1);
53                  solver[s_c++].start();
54              }
55
56          for (int i = 0; i < s_c; i++)
57              try {
58                  solver[i].join();
59              } catch (Exception e) {}
60      }
61  }
```

SolverCorrect.atom.java

```
1   package orium.knight;
2
3   import java.awt.Point;
4   import java.lang.Thread;
5
6   public class SolverCorrect {
7       private KnightMoves km;
8       private Point me;
9       private int moves;
10
11      public SolverCorrect(KnightMoves km, Point me, int moves) {
```

```
12          this.km = km;
13          this.me = me;
14          this.moves = moves;
15      }
16
17      public void go() {
18          Point[] next = new Point[8];
19          int c = 0;
20          int m[] = { 1, -1 };
21
22          /* Check if other thread has a better solution */
23          if (km.get_solution(me) <= moves)
24              return;
25
26          km.set_solution(me, moves);
27
28          if (me.equals(km.get_prey()))
29              return;
30
31          for (int i = 1; i <= 2; i++)
32              for (int j = 1; j <= 2; j++)
33                  if (i != j)
34                      for (int k = 0; k < 2; k++)
35                          for (int l = 0; l < 2; l++)
36                              next[c++] = new Point(i * m[k] + me.x, j * m[l]
37                                      + me.y);
38
39          for (Point n : next)
40              if (n.x >= 0 && n.y >= 0 && n.x < KnightMoves.WIDTH
41                      && n.y < KnightMoves.WIDTH) {
42                  new SolverCorrect(km, n, moves + 1).go();
43              }
44      }
45  }
```