

Runtime Checking of Refinement
for Concurrent Software Components

by

Tayfun Elmas

A Thesis Submitted to the
Graduate School of Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Electrical & Computer Engineering

Koç University

December, 2005

Koç University
Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Tayfun Elmas

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

Assist. Prof. Serdar Taşiran (Advisor)

Assist. Prof. Hüsnü Yenigün (Sabancı University)

Assist. Prof. Attila Gürsoy

Assist. Prof. Öznur Özkasap

Date: _____

To Hatice, Temel and Galip
“Sisters by chance, friends by choice.”

ABSTRACT

In this thesis, we present a runtime technique for checking that a concurrently-accessed data structure implementation, such as a file system or the storage management module of a database, conforms to an executable specification that contains an atomic method per data structure operation. The specification can be provided separately or a non-concurrent, “atomized” interpretation of the implementation can serve as the specification. The technique consists of two phases. In the first phase, the implementation is instrumented in order to record information into a log during execution. In the second, a separate verification thread uses the logged information to drive an instance of the specification and to check whether the logged execution conforms to it. We paid special attention to the general applicability and scalability of the techniques and to minimizing their concurrency and performance impact. The result is a lightweight verification method that provides a significant improvement over testing for concurrent programs.

We formalize conformance to a specification using the notion of refinement: Each trace of the implementation must be equivalent to some trace of the specification. Among the novel features of our work are two variations on the definition of refinement appropriate for runtime checking: I/O and view refinement. These definitions were motivated by our experience with industrial-scale concurrent data structure implementations in the Boxwood project, a novel storage infrastructure. I/O and view refinement checking were implemented as a verification tool named VYRD (**VerifYing** concurrent programs by **R**untime **R**efinement-violation **D**etection). VYRD was applied to the verification of Boxwood, Java class libraries. It was able to detect previously unnoticed subtle concurrency bugs in Boxwood, the known bugs in the Java class libraries and manually constructed examples. Experimental results indicate that our techniques have modest computational cost.

ÖZETÇE

Bu tezde, dosya sistemi ya da veritabanı depolama yönetim birimi gibi koşut zamanlı ortamda çalıştırılan bir yazılım bileşeninin çalıştırılabilir bir belirtime uygunluğunu denetlemek için bir çalışma zamanı denetleme tekniği sunulmaktadır. Belirtim her bir veri yapısı operasyonu için atomik bir metot sağlamaktadır. Belirtim gerçekleştirimden ayrı olarak sağlanabileceği gibi gerçekleştirimin atomik olarak çalışacak hale getirilmiş bir versiyonu da belirtim olarak kullanılabilir. Tekniğimiz iki aşamadan oluşmaktadır. İlk aşamada, gerçekleştirim çalışma sırasında yürütme bilgisini bir günlüğe kaydedecek şekilde donatılır. İkinci aşamada, ayrı bir sınama birimi günlüğe kaydedilmiş yürütme bilgisini kullanarak belirtimin bir örneğini çalıştırır ve kayıtlı yürütmenin belirtime ait yürütmeye uygunluğunu denetler. İlgili tekniğin genel uygulanabilirliğine ve ölçeklenebilirliğine, ayrıca denetlenen bileşenin koşut zaman özelliğine ve başarımına etkisinin en aza indirilmesine önem verilmiştir. Sonuç olarak, koşut zamanlı programlar için standart test yöntemine önemli yenilikler getiren bir doğrulama metodu geliştirilmiştir.

Gerçekleştirimin belirtime uygunluğu arıtma adında bir doğruluk kriteri olarak biçimselleştirilmiştir. Bu kritere göre gerçekleştirimin her bir yürütmesi için ona denk belirtime ait bir yürütme bulunmalıdır. Yöntemin yeni özellikleri arasında en önemlisi girdi/çıkıtı arıtma ve görüş arıtma adında iki arıtma kriterinin tanımlanmasıdır. Boxwood adında endüstriyel ölçekte veri yapısı gerçekleştirimleri içeren bir sistemin doğrulanması çalışmaları bu tanımlara motivasyon kaynağı olmuştur. Girdi/çıkıtı arıtma ve görüş arıtmanın denetlenmesi VYRD (**V**erif**Y**ing concurrent programs by **R**untime **R**efinement-violation **D**etection) adlı doğrulama aracı bünyesinde gerçekleştirilmiştir. VYRD, Boxwood sisteminin, Java sınıf kütüphanesinden birkaç sınıfın doğrulanması amacıyla uygulanmıştır. Sonuçta, Boxwood sisteminde daha önce tespit edilmeyen bir hataya ek olarak, Java sınıf kütüphanesinde ve test amaçlı yazılan örneklerde daha önceden bilinen hatalar yakalanmıştır. Deneyimler, önerilen tekniğin pratikte uygun bir çalışma maliyeti olduğunu ortaya koymuştur.

ACKNOWLEDGMENTS

I would like to thank the members of my thesis committee for critical reading of this thesis and for their valuable comments.

I would like to express my gratitude to my supervisor *Assist. Prof. Serdar Taşiran*. His experience, understanding and patience added considerably much to my graduate experience, and his valuable comments and advices helped me in adapting intensive academical research.

The work reported in this thesis was supported by Microsoft® Research in Redmond and Silicon Valley. I would like to thank *Jim Larus*, *Roy Levin* and *Sriram Rajamani* for making our collaboration possible. I would like to express my appreciation to *Shaz Qadeer* from Software Productivity Tools Group for his helpful discussions and contributions to my thesis. I am also indebted to *Chandu Thekkath* and *Lidong Zhou* from Boxwood group for their support in understanding and running Boxwood software.

I am grateful to all my friends from KOÇ University, for their close friendship during the whole course of my graduate study: the *Z-20* and *ENG-231* members: *Bengi*, *Gökçe*, *Ozan*, *Zülküf*, *Utkan*, *Başak*; the *CADT* members: *Soner*, *Erkan*, *Nesra*, *Deniz*; as well as dear *Selen*, *Hazal*, *Canan*, and *Nesrin*.

I am also very grateful to friends from EGE University, *Murat*, *Arda*, *Sinan*, *Anıl*, *Cenk*, *Sevcan*, *Kader*, *Çağdaş* and the others for continuing a strong and sincere relationship with me for the rest of my life.

I am thankful to my very special friends *Temel* and *Galip* for their extreme morale support and for always believing me in hard days of my life.

I must give immense gratitude to my parents *Süleyman* and *Zeynep*, my dear sister *Hatice*, my cousins *Peyman*, *Işıl* and *Burcu* for their endless support and patience.

TABLE OF CONTENTS

List of Tables	x
List of Figures	xi
Nomenclature	xiii
Chapter 1: Introduction	1
1.1 Contributions	5
1.2 Organization	6
Chapter 2: Survey on Verification of Concurrent Programs	8
2.1 Correctness for Concurrent Software	8
2.2 Verification Techniques	11
2.2.1 Testing-based Approaches	11
2.2.2 Runtime Monitoring	12
2.2.3 Static Analysis	14
2.2.4 Coverage Metrics	19
2.3 Previous Work on Verifying Conformance to Specification	20
Chapter 3: Concurrent Software Components	24
3.1 Example: Multiset	24
3.1.1 The Specification	25
3.1.2 The Implementation	26
3.2 Syntax	27
3.3 Intermediate Form with Atomic Constructs	28
3.4 Semantics	29
3.5 Concurrent Programs	30

Chapter 4:	Refinement Criteria for Concurrent Components	36
4.1	Refinement as a Correctness Criterion	37
4.2	The Need for I/O-refinement	38
4.3	I/O-refinement	40
4.4	The Need for View-refinement	41
4.5	View-refinement	43
4.5.1	View Definition of A Component	43
4.5.2	Definition of View-refinement	46
Chapter 5:	Checking Refinement at Runtime	49
5.1	Commit Points	49
5.2	Separating the Program from the Checker via Logging	52
5.3	Verifying Mutator Methods	53
5.4	Verifying Observer Methods	56
5.5	The Atomized Implementation as the Specifications	58
5.6	Checking Invariants	59
5.7	The Refinement Checking Algorithm	60
5.8	Commit Blocks	62
Chapter 6:	VYRD: the Runtime Checking Tool	69
6.1	The Testing Strategy	69
6.2	Logging Issues	70
6.3	Analysis at Different Granularities	71
6.4	Programmer's Responsibilities	72
6.4.1	Programmer Defined View Construction	72
6.4.2	Incremental View Computation & Comparison	74
6.4.3	The Specification	75
6.4.4	Code Instrumentation for Logging and Replaying	75
Chapter 7:	Experience	80
7.1	The Boxwood Project	80

7.2	The BLinkTree Module	82
7.2.1	The Concurrent Design of BLinkTree	82
7.2.2	The Boxwood’s Implementation of BLinkTree	84
7.2.3	Verifying the BLinkTree Module	84
7.2.4	Instrumenting the BLinkTree Module	86
7.2.5	Inserting Instrumentation Code into The Existing Classes	87
7.3	The Storage Modules of Boxwood	88
7.3.1	Verification of Cache + Chunk Manager	88
7.3.2	The Bug Detected in the Cache Module	89
7.4	Microbenchmarks	89
7.4.1	Multithreaded Java Class Libraries	90
7.4.2	Multiset Implementations	91
7.5	Evaluating VYRD	92
7.5.1	Error Catching Capability	92
7.5.2	Computational Overhead Measurements	94
Chapter 8:	Conclusions	101
8.1	State of the Art	101
8.2	Summary	102
8.3	Future Directions	103
8.3.1	VyrdMC: Verifying Refinement During Model Checking	103
8.3.2	Measuring Coverage for Concurrency	105
8.3.3	Static Methods for Verifying Refinement	105
Bibliography		107
Vita		116

LIST OF TABLES

7.1	Time to detection of errors	93
7.2	Overhead of logging for I/O- and view-refinement	94
7.3	Running time breakdown	95

LIST OF FIGURES

3.1	Specifications of the Multiset operations	25
3.2	Implementation of the Multiset operations	33
3.3	An example syntax for concurrent component implementations	34
3.4	The constructs of ActLang	35
4.1	Refinement of Multiset.	39
4.2	Buggy implementation of FindSlot	41
4.3	Refinement violations in the buggy version of Multiset.	42
4.4	Abstraction function of the Multiset example	45
4.5	Three different representation for Multiset whose views are the same.	45
5.1	Commit points for Multiset operations	50
5.2	The communication between the test program and the checker through the log.	52
5.3	Checking refinement for observer methods.	57
5.4	An example Multiset specification derived from the implementation code. . .	65
5.5	Runtime refinement checking algorithm	66
5.6	The procedure that performs checks at each commit point	67
5.7	The implementation of InsertPair with the commit block marked.	68
6.1	Example test program for Multiset.	77
6.2	Incremental version of the abstraction function for Multiset	78
6.3	The wrapper class implemented as the Multiset specification.	79
7.1	Boxwood block diagram	81
7.2	The Insert subroutine with the conditional commit point annotations. . . .	96
7.3	Decomposition of a b-link tree to indexing structure and leaf structure. . . .	97

7.4	Part of the view updated incrementally for a leaf pointer node.	97
7.5	The pseudo code for WRITE method of Cache	98
7.6	The pseudo code for COPY-TO-CACHE method of Cache	98
7.7	The pseudo code for FLUSH method of Cache	99
7.8	Code fragments illustrating the error in <code>java.util.Vector</code>	99
7.9	Code fragments illustrating the error in <code>StringBuffer</code>	100
8.1	The architecture of VyrdMC	103

NOMENCLATURE

ADT	Abstract Data Type
CPU	Central Processing Unit
GB	Giga bytes
I/O-refinement	Input/Output-refinement
KLOC	Thousand Lines of Code
LOC	Lines of Code
MC	Model Checking
PC	Personal Computer
POR	Partial Order Reduction
RAM	Random Access Memory
VYRD	V erif Y ing concurrent programs by R untime R efinement-violation D etection
VyrdMC	Refinement Checking with VYRD in Model Checking

Chapter 1

INTRODUCTION

Many widely-used software systems, such as file systems, databases, Internet services, and standard Java and C# class libraries [1, 2], operate in concurrent environments. These systems have concurrently-accessed components at their core [23, 24]. Functional correctness in such systems is just as important as performance, since functional errors may have serious consequences such as data loss, corruption, or system crash. Functional errors due to concurrency are hard to detect and reproduce through conventional approaches like testing [5].

In this thesis, we view a concurrently-accessed software component as an implementation of an abstract data structure that provides public methods that allow its clients to make operations (modifications/queries) on the abstract data structure. Functional correctness of a component is important in that it provides to the rest of the system a reliable data structure that is free of concurrency errors.

Clients of a component expect its methods to run so that each operation on the abstract data structure appear as if it is being performed atomically. In an atomic execution, each operation is performed as if the thread running the operation updates the data structure state without being interleaved with the other threads. A component in which the whole body of each method is globally synchronized naturally satisfies this property. However, performance requirements force developers to use as small synchronized blocks as possible and to use intricate synchronization mechanisms. This makes these systems prone to concurrency errors. The mechanisms devised to strike a compromise between guaranteeing atomicity and optimizing performance make such components hard to verify.

In this thesis we define *abstract atomicity* to express clients' expectations from a concurrent component. It ensures that operations of the abstract data structure implemented by the component appear to run as if each operation runs atomically, although the entire

body of each operation may not be fully-synchronized. We distinguish between “abstract atomicity” and “atomicity” as used in [20, 51]. The latter, which is a more stringent criteria, ensures that for each interleaved execution of the component, there is an equivalent “reduced” version of this execution that accesses the program variables atomically and results in the same end state. Abstract atomicity for a data structure guarantees absence of common concurrency-related errors [77] in programs that use that data structure. Abstract atomic behavior of operations is a more thorough criterion for correctness than method-local assertions. It states a requirement about updates to the whole state of the component and involves a deeper analysis and tracking of what is expected to remain consistent in a concurrent environment¹.

Studies that investigate about similar criteria make use of specifications in different forms in order to specify sequential behaviors of concurrent programs. Atomicity, a widely-studied correctness criterion [20, 19] requires each implementation trace to be equivalent to some atomic execution *of the implementation*. The distinction between our approach and atomicity in what specifies the expected sequential behavior can be important for concurrent systems. Atomicity uses the implementation itself run sequentially as the reference for correct behavior. As a result, atomicity is sometimes too stringent and may declare certain acceptable system behavior as erroneous. Consider, for example, an implementation in which a method may terminate exceptionally because resource contention between concurrent threads prevents it from completing its job. If a specification that allows exceptional method terminations had been used for refinement checking, this execution would not have caused a refinement violation.

There have been several static and dynamic approaches to check atomicity-like criteria in order to detect concurrency errors. Concurrency bugs are notoriously difficult to detect and reproduce through testing. Model checking is a powerful technique to verify such complicated properties, but it is prone to state space explosion [29] due to large numbers of thread interleavings and the exponential dependency of the state space on the program size. As the refinement criterion requires most of the implementation details to be preserved to show correctness, abstracting the program to reduce the state space may cause false warnings.

¹Throughout this thesis we will refer to a concurrent environment as “environment”, unless otherwise specified.

Thread-modular verification [30, 31] aims to verify properties of individual threads with respect to the rest of the system specified as environment assumptions. The main difficulty in applying these techniques is deriving the environment assumptions modeling effects of the other concurrent threads. In this case the number of the assumptions can blow up for software of large sizes.

In this study, we investigate a correctness criterion that allows us to reason about abstract atomicity of a component with respect to the abstract data structure it implements. Our criterion requires a component implementation to conform to its sequential specification. The specification captures sequential behaviors of operations in which each method is guaranteed to be executed atomically. The necessary conditions to verify conformance can be expressed as a *refinement* relation [10] between the component implementation and its specification. Checking refinement consists of verifying that each execution trace of a concurrent implementation is “equivalent” to a trace of its specification. This criterion provides more thorough validation than checking method-local assertions during program execution. Thus, if an implementation refines a specification, each implementation trace in which portions of method executions are interleaved with others is equivalent to a *specification trace* with atomic method executions.

We define two notions of refinement, called I/O-refinement and view-refinement [39, 7]. These definitions are motivated by our experience with verifying an industrial-scale concurrent program: Boxwood, novel software for distributed storage infrastructure [24]. Components in Boxwood have highly concurrent method implementations that use intricate synchronization mechanisms and different types of caching to improve performance. For example, BLinkTree operations use a very relaxed locking scheme which allows operations from different methods to overtake each other. Interestingly, none of these methods can be shown to be atomic using simpler correctness criteria and analysis methods studied earlier such as reduction [20], commit atomicity [17], or purity [18].

I/O-refinement stipulates that for each execution of the implementation, there is an atomic, sequential run of the specification consisting of the same method calls (and arguments) and associated return values. The specification can be provided separately in a higher-level description language or a non-concurrent, “atomized” interpretation of the implementation can serve as the specification. This correctness criterion can be viewed as

a variant of linearizability [21] as elaborated on in Chapter 2. Our analysis requires that in every method execution, a unique action is marked as the “commit action”, and uses the order of occurrence of commit actions during execution to generate a “witness interleaving”. This novel use of commit actions distinguishes I/O-refinement from testing. Given the witness interleaving, we can determine exactly which return values are allowed for a particular invocation of a method. In the absence of this information, pure testing is forced to be overly permissive.

Our second correctness criterion, view-refinement, provides more thorough checking than I/O-refinement by means of more visibility into program state. View-refinement augments I/O-refinement by requiring a particular correspondence between the implementation and specification states when a commit action is taken. This correspondence is captured by a high-level abstraction of the component state called “view”. Hypothetical view variables view^I and view^S are added to the implementation and specification that represents the view of implementation and specification, respectively. Intuitively, the value of the view variable is a canonical representation of the abstract data structure state when the commit action is taken. View-refinement requires that the value of view^I matches view^S at commit points for each method execution. The programmer specifies how view^I and view^S are to be computed as a function of the implementation and specification state, respectively.

The visibility view provides into program state makes it possible the early detection of discrepancies of the implementation state from the specification state. These discrepancies are signs of concurrency errors that will appear in return values of subsequent method calls. In contrast, in testing and I/O-refinement, an implementation error is detected only if and when it causes a discrepancy in the return value of a method. In a particular run, a triggered error may lead to an observed discrepancy long after it occurs, or not at all, whereas, in a different run, the same error would have caused a more serious and easily observable outcome. **view** enables the detection of the error even in the former case.

We introduce a new scalable runtime analysis technique for checking refinement. The I/O-refinement and view-refinement criteria are defined so that they are appropriate for runtime checking. Our focus is on analyzing runtime information from test executions of the component. This technique is between testing and exhaustive verification but it provides a solution that strikes a compromise between advantages and disadvantages of

these approaches. Runtime checking of our simpler correctness criterion, I/O-refinement, requires very little instrumentation and logging while still providing a more rigorous check than pure testing. To enable runtime checking of I/O-refinement, the programmer annotates the implementation code so that in every method execution, a unique action is marked as the “commit action”. This strength of view refinement comes at the cost of more detailed instrumentation, logging, and programmer effort put into specifying how view^S and view^I should be computed.

We implemented I/O and view refinement checking as a verification tool called **VYRD** (VerifYing concurrent programs by Runtime Refinement-violation Detection) [7]. VYRD can perform refinement checking of industrial-scale concurrent data structure implementations. We used the Boxwood project to drive our development of the VYRD tool. VYRD is very effective in detecting concurrency errors. VYRD was also able to catch known concurrency bugs in `java.util.StringBuffer` and `java.util.Vector`. In addition to extensive experimental results about the effectiveness and computational cost of using VYRD, we report the issues identified and solutions implemented while verifying industrial-scale systems. Chief among these are minimizing performance and concurrency impact on the program being verified and incremental computation and comparison of view^I in order to avoid re-traversing the entire program state at each verification step.

1.1 Contributions

In this study, we have the following contributions:

1. We propose using refinement [10] as a correctness condition for concurrently-accessed components. A component that refines its sequential specification satisfies the property of abstract atomicity. A key contribution of this thesis is the definition and use of two variations, I/O-refinement and view-refinement, on the notion of refinement appropriate for runtime checking.
2. I/O-refinement stipulates that for each execution of the implementation, there is an atomic, sequential run of the specification consisting of the same method calls (and arguments) and associated return values. This novel use of commit actions distinguishes I/O-refinement from testing and provides practically-efficient checking of an

execution trace.

View-refinement augments I/O-refinement by requiring a particular correspondence between the implementation and specification states when commit actions are taken along an execution. This correspondence is captured by a high-level abstraction of the implementation and specification states. View-refinement provides more visibility than testing and I/O-refinement and makes possible early detection of errors.

3. We have chosen to check refinement at runtime by analyzing execution traces of the component. Our technique sacrifices completeness because of the computational cost and practical difficulty of exhaustively verifying concurrent software. Our approach is very scalable, requires relatively little manual effort, and provides significantly more thorough checking than testing.
4. Our technique has the ability to use specifications that more closely model real concurrent executions, e.g. with exceptional termination of the methods due to concurrency. This makes refinement a more appropriate correctness criterion for concurrent systems. Our technique accepts a separate specification written in a high-level description language or a non-concurrent, “atomized” interpretation of the implementation can serve as the specification. The latter choice allows the sequential verification and concurrency verification problems to be studied separately.
5. We implemented I/O and view refinement checking as a verification tool called **VYRD** (**VerifY**ing concurrent programs by **R**untime **R**efinement-violation **D**etection). VYRD can perform refinement checking of industrial-scale concurrent data structure implementations. VYRD comes with several improvements that minimize the performance and concurrency impact of checking on the program being verified and incremental computation and comparison of abstract states for view-refinement.

1.2 Organization

The organization of this thesis is as follows. We provide a survey of the literature on verification of concurrent programs in Chapter 2. We introduce different aspects of concurrent software components from our technique’s point of view and give an example component

that we will use throughout this thesis in Chapter 3. The refinement criteria including I/O-refinement and view-refinement are formalized in Chapter 4. Chapter 5 elaborates our methods to check refinement criteria at runtime. We present our tool VYRD that implements our runtime checking technique in Chapter 6. Experimental results from the application of VYRD to Boxwood and other programs are described in Chapter 7. Chapter 8, including the future work, concludes the thesis.

Chapter 2

SURVEY ON VERIFICATION OF CONCURRENT PROGRAMS

Since concurrent and parallel computation techniques and algorithms got interest of computer scientists, testing and verifying these systems has been exploited by people from both hardware and software literature extensively. This chapter gives a comprehensive survey on the theoretical and practical studies on analysis and verification of concurrent software systems.

2.1 Correctness for Concurrent Software

There have been studies on defining a correction criteria for concurrent systems. Most effort has been given to memory consistency models to hardware and software implementations with respect to different kinds of consistency models [55, 56]. A system that correctly implements a consistency model gives guarantees to programmers that work on top of this system, which helps programmers develop programs with little effort. Sequential consistency [48] and linearizability [21] are the most popular ones.

Sequential consistency was defined by Leslie Lamport in [48]. Sequential consistency assures the execution of a parallel program to appear as some interleaving of operations of the processes on a single sequential processor. In other words, the sequential consistency defines a correct execution as one whose results is the same as if all operations of all processors are interleaved in a single processor as a sequential order, and the order of the operations of an individual processor is the same as in its program order. Lamport also gave two implementation requirements which, if met, would enforce sequential consistency. (1) Each processor issues memory requests in the order specified by its program. (2) Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue. In sequential consistency a processor observes write operations of all processors but only sees its own read operations. There is no global time notion for the whole execution.

Linearizability [21] is more strict than sequential consistency in that it requires the history order between the methods to be preserved when deriving a corresponding sequential history of a concurrent history [64]. Our definition of sequential execution for methods is similar to the one in the theory of linearizability, with a difference that the sequential executions are parameterized by the termination condition of the methods that forces the sequential method to fail even the precondition does not permit.

To ensure that these conditions hold, strict synchronization mechanisms are applied by programmers but sequential consistency is mostly convenient by programmers to assume than other weaker consistency models [4] which increase concurrency and, thus performance.

Atomicity in the work of Flanagan and Qadeer [20], and Wang and Stoller [28] handle the correctness problem from a more abstract and software-centric point of view. Their criteria are based on reduction theory by Lipton [22] and require that non-atomic (interleaved) executions of the implementation be “equivalent” to a sequential run of the implementation obtained using reduction. While Flanagan and Qadeer uses a type system to statically check atomicity for code blocks, Wang and Stoller tracks at runtime specific patterns in action sequences to find non-atomic executions. [51] also introduces a dynamic atomicity checking algorithm that uses Lipton’s reduction theory and lockset algorithm of Eraser, in order to check atomicity violations dynamically. They try to infer the lockset of variables dynamically, so this can result in missing some erroneous states. Their algorithm detects the locksets of variables without requiring much annotation to the source code. They formalize the instrumented semantics that annotate an execution with necessary information (mover types of actions [20], pre- and post-commit types for states [51] etc.) that allows them to reason about the atomicity of standard semantics.

The works above are important because they use a very similar condition that seeks for an interleaved execution of the implementation an equivalent sequential behavior that validates the interleaved one. The distinction between what specifies the expected sequential behavior in atomicity and abstract atomicity in our approach can be important for concurrent systems. In reduction based approaches like atomicity, the sequential specification is defined based on the implementation and this makes atomicity unnecessarily restrictive in some cases. Consider, for example, an implementation in which a method may terminate exceptionally because resource contention between concurrent threads prevents it from

completing its job. In an atomic execution of this implementation, there is only one thread in the middle of a method execution at any given time and thus no resource contention between methods, therefore, method executions never terminate exceptionally. Therefore, executions of this implementation containing exceptional terminations of this method are not equivalent to any atomic execution of the system and will be declared erroneous according to the atomicity criterion. Since there is not a (possibly more permissive) specification separate from the implementation itself run atomically, atomicity is sometimes too stringent and may declare certain acceptable system behavior as erroneous. If a specification that allows exceptional method terminations had been used for refinement checking, this execution would not have caused a refinement violation. The ability to use specifications that more closely model concurrent executions makes refinement a more appropriate correctness criterion for concurrent systems.

The choice of the definitions for I/O-refinement and view-refinement were motivated by our experience with verifying two industrial-scale concurrent programs, neither of which could be proved by reduction (e.g. as in [20]), by the concept of “commit atomicity” as in [17], or by proper selection of “pure” code blocks as defined in [18]. The intuition behind this fact is illustrated by the following simple example inspired by Boxwood. Consider a tree implementation of a set data structure in which data is stored at the leaves only, while internal nodes (the “indexing structure”) enable search. In the following, let p denote a write to a shared variable that changes the abstract data structure contents, i.e., a write which changes the contents of a leaf. Let q denote a shared variable that represents an internal node of the tree. In the example below, writes to q re-arranges the shape of the data structure, for example, by re-balancing the tree. Suppose all writes to p and q are lock protected. Consider the interleaving of actions by two threads t_1 and t_2 concurrently executing two methods μ_1 and μ_2 respectively:

$$\begin{array}{ll} t_1: & W(p) \qquad \qquad \qquad W(q) \\ t_2: & \qquad W(p) \quad W(q) \end{array}$$

Because t_1 and t_2 perform multiple, lock-protected writes to the same set of shared variables that do not commute with each other, it is not possible to convert this execution using reduction or purity to a method-atomic execution of the implementation. However, since only the writes to p modify the abstract data structure state, refinement checking can be performed using the witness ordering of μ_1 followed by μ_2 and will not report any

refinement violations.

I/O-refinement can be viewed as a variant of linearizability, where the “sequential specification” for the data structure implementation is given separately in the form of an executable specification. The specification may contain non-determinism in order to allow termination scenarios not possible in a sequential execution. This is accomplished by giving the methods of the specification the result of the execution of the corresponding implementation methods (through method return values) and making the specification methods to consider this result while running. This way of having non-determinism in the specification allows I/O-refinement to rule out fewer components that has intricate implementations but satisfy abstract atomicity.

Checking refinement as a verification approach is well-studied (See [10, 25] among many others). Runtime checking of conformance to a state invariant derived from an object model has been investigated [16]. Runtime checking of property annotations inserted into implementation code has been studied [59]. Runtime analysis methods for correctness criteria such as atomicity have been developed [19, 28]. Refinement checking has recently been integrated with simulation-based validation of hardware designs [27]. The commit-atomicity work of Flanagan [17], published simultaneously with an earlier version of the work described in [39], is closely related but is a very restricted case of the work presented here. In commit atomicity, the specification is required to be the atomized version of the implementation, refinement checking is done only at clean points rather than at each commit point, and a complete match between specification and implementation states is required whereas we, by using view variables, can declare more intuitively equivalent program states as matches. Our work is the first general tool that can check at runtime that an industrial-scale concurrent data structure implementation refines a specification.

2.2 Verification Techniques

2.2.1 Testing-based Approaches

Some other work is dedicated to test input generation for multithreaded test programs of concurrently accessed components. Generating test cases focused on catching errors is important for verification techniques that use test programs to produce executions to be analyzed. Visser et.al. [66] showed how model checking and symbolic execution can be used

to generate test inputs for functional testing of complex data structures. They discussed aspects of three different test generation techniques, pure model checking, model checking used in a black-box fashion to generate all inputs up to a fixed size, and model checking used during white-box test input generation. They emphasized efficacy of white-box test input generation for code complex data structure like the red-black tree, considering complex method preconditions. They use generalized symbolic execution [67] to abstract data structures with relevant information, conservative preconditions to pick the appropriate objects, and Java PathFinder [81] to produce the intended data structure with lazy initialization and state space search.

2.2.2 Runtime Monitoring

There have been runtime monitoring techniques that analyze executions of concurrent Java programs. Although most of the runtime techniques are unable to cover the whole execution space of the program being checked, they rarely give false alarms because they reason about realized behaviors. False alarms can also be eliminated by manually examining the execution trace. Java PathExplorer [12] facilitates testing of execution traces of programs by checking conformance of the traces against user supplied high level formal specifications such as temporal logic formulas. It also performs error pattern analysis on the executions using error detection algorithms, like the ones for deadlocks or race conditions, to detect error-prone code fragments. They use byte code instrumentation to insert the relevant code to perform the check but it is observed that the monitoring via event extraction slows down the execution of the original program by affecting its concurrency characteristics.

Rosu and Sen [14] present an automated code instrumentation procedure that generates appropriate messages during execution of a Java program to be sent to an external observer that performs some elaborated system analysis. The observer extracts the state update information from those messages together with the relevant causality partial order among the updates. This partial order abstracts the behavior of the running program and allows an observer to analyze this partially ordered computations rather than just flat sequences of events. This helps in predicting errors from analyzing successful executions, errors which can occur under a different thread scheduling and can be hard to find by just testing.

There are also studies on testing multithreaded software leveraging state space explo-

ration techniques of model checking. Some of them focus on generating test scenarios that explores unique paths using state-less search techniques [80]. Stoller extended the idea for multithreaded and distributed Java programs. Both benefit from partial order reduction techniques to reduce the state space checked [32]. Gunter and Peled [33] proposed a technique which helps running multithreaded programs guided by a path condition. They use it for producing or replaying paths with intended properties, proving partial correctness of programs, exploring similar/related paths to an execution, and also stepwise debugging a program, directed by temporal formulas. Their method requires the program to be decomposed to a flowchart and the tracer to run over this flowchart guided by the path condition. Their tool allows the user to select her desired execution traces for both sequential and concurrent programs, and executes the selected path. It helps software debugging by translating the code to visual flowcharts for the user to choose some path throughout the code in visual mode. It computes a formula called path condition to determine in which condition the selected path is executed and starts the execution in the state that the path condition is satisfied. The computation of path conditions may be helpful to replay some executions of concurrent programs. Their method requires for the user to select the paths but the path condition can be calculated while the program is in progress, and then when the program exits, the online computed path condition can be used to replay the execution. However, this requires the analysis of the program code.

Wang and Stoller [46] present a runtime analysis method that uses temporal logic and determines whether a given trace of events satisfies the atomicity properties similar to the ones in [20]. Some specialized algorithms can alert the suspicious misbehaving traces as well as controlling the atomicity of traces. The concept of atomicity in that work is used for isolation in ACID (Atomicity, Consistency, Isolation, and Durability), which defines four properties of transaction systems. Concurrent transactions can be considered as concurrent calls of the methods of an abstract data type (ADT). For an ADT whose methods guarantee atomicity, set of concurrent method calls is equivalent to some serial executions of the methods.

Monitor-oriented programming [59] checks the conformance to specifications at runtime (see also Section 2.3). MoP does not modify the original programming language. It uses *annotations* to insert logic statements and a tool automatically generates code to check

the logics in target programming language. The annotations consist of two entities, atomic predicates and formula defined with predicates. Formal methods are not always practical because of state explosion and invariant discovery problems while testing is unable to guarantee correctness. MoP merges both approaches to achieve their benefits. However, the loss of coverage problem must be considered since MoP checks only single execution traces. It finds many but not all of the errors. Although MoP can be extended in order to check simple assertions for concurrent programs, refinement criteria, especially view-refinement, require more extensive checking that makes it hard to express these criteria using simple logic statements.

2.2.3 Static Analysis

There are also static techniques to verify some properties or invariants of system [71, 44, 72, 43, 52, 53] observing static structure of data and control paths to reason about the possible behaviors. They can find a variety of common error patterns and bugs. However, their capabilities are limited and inaccurate to obtain behavioral coverage. Imprecise static methods can exhibit many false alarms that would not occur at runtime. Another problem with static analysis is current approaches are unable to check thorough criteria like refinement. The interleaved behaviors are required to be analyzed extensively, i.e. through symbolic execution, which is hardly precise and costly compared to runtime analysis.

With the intensive use of multithreaded software systems, especially for internet servers, software verification world concentrated on verification of concurrent systems [34]. Model checking tools verify that a concurrent system satisfies a given logic formula, often in temporal logic [35, 36]. Some approaches are dedicated to verifying some properties on concurrent software, like race-free execution, deadlock detection, atomicity of relevant code blocks or invariants of systems [37, 46, 47, 42]. Most of them focus on the synchronization events and access to shared variables to apply their rules. This requires all the operations to shared data and synchronization objects to be specified and recorded to work on them.

Model Checking

Model checking have been one of the most preferred way of verifying concurrent systems. It uses computation and storage resources exhaustively, and subject to state explosion prob-

lem as systems become more complex and interactive. In order to reduce state space and resource usage, data and control path abstraction methods are used. Predicate abstraction [73] has been used widely for both systems in both formal languages and modern programming languages [74], even in object oriented languages [75]. Some of the studies generate on-the-fly a state graph of the system [73, 72], which is expressed in a formal way. BLAST [74] derives a control flow automaton of a source code using lazy abstraction and does model checking according to given temporal logic formulas. In a similar way, BANDERA [36] extracts a finite-state model from the original source code and translates it to a formal language (Promela), of a verification tool (SPIN [15]) by applying abstraction interpretation to extract relevant parts of the code and let the tool check the desired properties, and it requires user intervention to annotate the source code with desired properties. Both BLAST and BANDERA rely on the user to guide the abstraction of the code by specifying the relevant parts in abstract specification languages. By contrast, SLAM [45] does abstraction automatically, given a safety property, it can automatically abstract the relevant parts of the code. SLAM is a toolkit that checks temporal safety properties of sequential C programs. It does not require programmer annotations and uses counter-example driven abstraction refinement and error reporting. They use predicate abstraction and get a boolean program which expose only the transitions on interested program variables that effect the safety properties. SLAM toolkit proposes a technique on deriving a boolean program from a C program, refining the boolean program to a more concise version upon spurious errors. SLAM abstracts the concrete statements of the code to an abstract boolean program using predicate abstraction [54] and does verification directly on the abstract code checking safety properties. However, it can only deal with sequential programs and can not handle concurrent programs and dynamically created objects. Java PathFinder [60] either translates the source code to a formal language or does the verification by running the program on a customized virtual machine. It uses the whole virtual machine state to represent the states experienced. Verisoft [80] uses state-less search and partial order reduction to reduce the contents of a state representation. Verisoft introduces a state space exploration technique for actual code written in an implementation language like C and C++. They use a state space search algorithm, which makes partial order reduction, to reduce the state space searched. Verisoft works on concrete code like C or C++ and reduces the amount of space

allocated in the memory for states.

Model checking techniques make use of partial order techniques to reduce the number of interleavings of concurrent threads that need to be explored [78]. Roughly speaking, a pair of actions executed by two different threads are declared independent unless one writes to a shared variable that the other reads from or writes to. Commuting a pair of such actions is said to result in an equivalent execution and only one execution from an equivalence class defined in this way is explored during model checking. The set of equivalence classes that this give rise to could be seen as a coverage metric for concurrent programs. However, since the coverage state induced by this metric includes both data, control and the interleaving aspects of a program, it is prohibitively complex and does not result in a usable coverage metric for large, practical programs.

Predicate Abstraction and Symbolic Execution

Predicate abstraction based techniques try to reduce the state space examined in model checking by abstracting the concrete program to a boolean program in which the variables are represented by boolean predicates and each statement changes the value a set of predicates. Graf and H.Saïdi [73] motivated deriving an abstract state graph using predicate abstraction and model check it to prove some properties of the concrete software system. It also motivated people to check refinement and to reason about correctness of the refinement over abstract models of a specification and its refined version. They introduce a method, facilitated by theorem proving, that automatically generates an abstract state graph of an arbitrary system, with predicate abstraction. They use a parallel composition of sequential systems. The concrete states are represented by monomials over predicates and abstract states are represented by binary variables corresponding to the concrete predicates. They introduce the post and pre predicate transformers that let them derive the abstract transition relation by introducing first order implications that must be validated by the theorem prover. Using their abstract transition relation, they make the reachable abstract state graph more precise. [61] introduced the concept of inductive invariant, which is also used for concurrent system verification to prove the validity of an invariant. They use a theorem prover to check the validity of the implications that must be proved to assess the inductive property of the inductive invariant. They concentrate on safety properties of programs and

generate a set of first order logic formulas such that the validity of the formulas over the reachable set of states sufficiently prove that the model under consideration satisfies the invariant.

Lazy abstraction [74] generates the abstraction of some source code iteratively within three phases called “abstraction”, “verification” and “counterexample-driven refinement”. Lazy abstraction refines the current abstract model of the software on demand in case the current model has some invalid properties that can not be realized by the concrete software. They also explain and illustrate the approximation of pre- and post-conditions using predicate abstraction. With the help of iterative refinement of abstract interpretations, they prevent infeasible errors due to the over-approximations by backtracking and checking the preconditions versus the formulas of the states being passed.

However, predicate abstraction is limited to express more structural properties of composite data structures with links. In our work, view of BLinkTree, a data structure with links between elements, and some structural invariants that would be so hard to express in predicates. A novel approach on how to transform an arbitrary C program to a boolean program was described in [54]. The statements in the boolean program correspond to abstract state transitions applied to abstract states to get the next abstract state. It differs from the abstract state graph approach in that it defines how the abstract state will be represented but does not generate abstract states. The abstract program generates abstract states as the boolean program executes. However, predicate abstraction suffers from dynamically created objects, so more predicates must be derived for each object living in the system.

Dynamic predicate abstraction handles this case [75] but it can introduce too many predicates, which cause inefficiency and resource problems. Dynamic predicates study shows how informal abstraction criteria is formalized as predicates over the program variables, and translate the original source code to some annotated version to check the introduced predicates. Dynamic predicates allow dynamic construction of abstract state graph at runtime increasing the precision of the abstract interpretation of the concrete program. They compare the abstract state graph methods versus abstract transition generation methods, and static versus dynamic abstraction methods.

Symbolic execution uses symbolic types and symbolic objects that abstract the actual data types and the actual data structure preserving only relevant information, and the

composite structure with links [67]. Even though it uses lazy initialization, it must limit the size of data structures since it explores all possible structural compositions. Symbolic execution suffices to reason about many correctness conditions, but it is not precise enough in checking conditions that mostly appear in dynamic behaviors of the program. This is even more important for concurrent programs because symbolic analysis may not capture the interleaved behavior of the program or it requires rather big amount of computation and memory cost.

Thread-modular Verification

Thread model checking for verification of properties on loosely-coupled multithreaded software systems where threads communicate via shared objects called communication objects. These shared objects may be managed by other components. This is promising even for distributed systems, where threads communicate with other local threads using local shared objects and communicate with remote threads using network messages sent/received. It is crucial to verify each local system separately by incorporating environment assumptions that guarantees other remote thread's behaviors. They use assume-guarantee reasoning based thread-modular model checking, by checking each thread in the system separately using automatically inferred environment assumptions. Thread-modular model checking [31] improves the thread-modular state space exploration by generating the environment assumption automatically using the global information and the affects of other threads on that global information via guarantees. They also handle procedure calls using stack information of the threads and makes the model checker terminate in polynomial time.

Freund and Qadeer [40] introduced a suite of techniques for checking concise specifications defined for concurrent procedures with respect to environment assumptions. The techniques include checking access predicates of shared data objects expressed as “accessible_if” annotations and verifying that the implementation conforms to a formal specification expressed as “performs” annotations. They also use atomicity to simplify the specifications and reduction to check that statements declared atomic in specification are really atomic. The assume-guarantee method is used to include the effects of the concurrent environment formed by other threads. Environment assumptions are included in the translated codes of the procedures. The result is a technique that check that implementations of procedures

obey high level, concise specifications and access conditions of data objects. They also use reduction to check that the procedures realize the atomic statements in the specification correctly. The conditions to regular or exclusively access conditions for data objects are checked by injecting annotations to the code. They introduce how statements can be translated to some formal expressions that check access predicates, atomicity property via reduction and performs annotations. Witness annotations are leveraged to take the points in the code where the atomic statements of the specification are executed. It mimics also the atomizer algorithm, to verify the atomic blocks.

2.2.4 Coverage Metrics

One problem with using testing to generate execution traces is low coverage due to random inputs and thread interleavings. As a result, a very small part of the whole execution space of a test case may be analyzed although the test case has some execution that reveals a concurrency error. One direction in the future work of this thesis is to address this low coverage problem of testing.

Extensive research has been performed on test adequacy criteria for sequential programs, and coverage measurement and improvement tools have been developed. However, there is relatively little work on coverage metrics geared towards concurrency errors and measuring how well different interleavings of a multi-threaded program have been explored. A hierarchy of structural coverage metrics for testing multi-threaded programs was defined in [79]. However, the complexity of the metrics increase exponentially with the number of concurrent threads and makes them impractical for industrial-scale systems. Long et.al. [70, 76] propose the use of Petri-net models for representing Java synchronization primitives. Firings of these nets correspond to different actions that can be performed with the primitive and are correlated with likely low-level errors such as race conditions. Method control flow graphs are annotated with firings of the net and test sets are constructed manually in order to achieve coverage of the firings. Their analysis is focused on a single synchronized object, which requires a separate application for each shared variable in a concurrently-accessed component – a fact that makes their approach impractical for our purposes. Farchi et.al. [69] provide a classification of concurrency errors and testing heuristics that make it more likely that these errors will be exercised, but do not provide a way of measuring when and how

well testing has addressed possible errors of this sort.

2.3 Previous Work on Verifying Conformance to Specification

Verifying conformance between models of a system in different levels of detail has been interesting for many studies. The system implemented in a higher level generally indicates a correct description with respect to a set of properties of what is intended to be realized. Some of them on modular model checking use abstracted models of concrete implementations to verify some properties of compositions of those models [57, 30]. There are works for verifying conformance between a specification and an implementation, but most of them are devised for verifying sequential execution of implementations, and based either on the relations between pre- and post-conditions of the methods or the representation of the component in a formal way. In order to take the exceptional executions not existing in a sequential environment, these conditions must be written in a way that sometimes checks the post-condition independent of the pre-conditions. For example the method may encounter a resource contention although it satisfies its pre-condition that states sufficient amount of resources for the method to complete its job successfully.

Antoy and Hamlet do the check for sequential programs with specifications expressed as algebraic axioms [58]. Their study maps concrete implementation states to specification objects and makes the implementation check its conformance and consistency by itself. It requires the programmer to write the specification in formal way, and the abstraction function that does the mapping.

Crane and Dingel checks conformance for class specifications written in a special language called Alloy [49]. The check is done at specific points, by dumping the current objects in the system and mapping the object instances to types in the specification and verifying the conformance of them to the formulas in the specification. They check the conformance of the structure of the implementation to the object model, so in order to check the method executions, the user must specify the points where the conformance check will be done. Similarly, Chen and Rosu in [11] propose a generic framework that translates the formulas from a wide range of logics defined over the programs to runnable code in a modern programming language. The resulting code monitors the execution of the actual code and performs the checks specified in the formulas. There have been some effort to do conformance check in

.Net platform of Microsoft®, between the specification written in AsmL, and the implementation in any language supporting common language runtime [62]. The specification can be run independently for simulation and test, and it can be verified separately or be used to verify its implementation at runtime. The check is done at method boundaries by having the specification to verify the pre- and post-conditions of its implementation. However, that study handles only sequential programs.

A technique that uses a formal specification of the EV7 cache coherence protocol and verifies its implementation subject to some properties defined formally in the specification is introduced in [63]. They run TLC, a formal checker for TLA+ [6], in parallel with the implementation; the implementation triggers the execution of the checker at specific points. Those points are called clean points, at which no transactions in the middle of their operation exist. They also define an abstraction map to abstract out details of the implementation to have a clean part of it that can be compared to the specification. Our work improves the method by not requiring the checker to run in only clean states. Their abstraction map idea is used to infer “view definition” of data structures.

Monitor-oriented programming [59], MoP in short, is a light-weighted formal method and a design principle in software engineering to check conformance of implementation to specification at runtime. It is fundamental to increase safety, reliability and dependability of safety-critical software. MoP enables the designer to define monitoring logical frameworks on top of a specific programming language. There are three viewpoints for MoP: (1) It increases the reliability and dependability of a system by monitoring its execution. (2) It is an extension of current programming languages providing different types of logics. (3) It is a light-weighted formal method that complements theorem proving and model checking.

Our notion of computing and checking abstractions during the execution was adopted from the idea of an abstraction map [50, 63], which associates an implementation trace with a specification. Park and Dill [50] rely on an aggregation function. Aggregation function is a type of abstraction function that aggregates the steps of each transaction in the implementation into a single atomic transaction in the specification. The concept is called aggregation as the abstraction function assembles the transactions into atomic transactions. For the method to be used there are identifiable set of transactions each of which must have a unique commit point. At the commit point, a state change first

becomes visible to the specification. The main role of aggregation function is to complete transaction that has committed but not completed. Each transaction has a commit step in which the first set of assignments to specification variables takes place. The states during the transaction but before the commit step are pre-commit states and the states that are after the commit step are post-commit states. The clean state is the state that every committed transaction has completed. The commit step is also a transition from a pre-commit state to a post-commit state. Transaction mapping is a function that maps each implementation step to its corresponding transaction in the specification. In these studies, an implementation of cache coherence protocol was verified by using abstraction map and an aggregation function. The basic idea was to associate an atomic transaction with one or more commit points of a concurrent transaction on which a modification to the abstract state in specification was performed. At commit points, they triggered a checker, which completed or waited for the completion of all committed transactions to reach a clean state, and then did the verification on the abstracted concrete state. The specification language was TLA+ and TLC was used as the checker. In this thesis we try to write a kind of aggregation function for component methods by seeing methods as software transactions. The aggregation function for a component implementation computes a clean state for the specification that is reachable by a sequence of atomic method executions.

Abadi and Lamport [10] described refinement based on trace inclusion, stating that a component $\mathcal{C}_i$ refines $\mathcal{C}_j$ iff for each behavior of $\mathcal{C}_i$ there is a behavior of $\mathcal{C}_j$ with the same sequence of externally visible states, allowing stuttering. Tasiran et.al. defined runtime refinement checking for software first in [65] and used it on a concurrently accessed file system code. They adopted file and directory creation, reads/writes to a file as methods, and data block writes to the disk as actions. As for the specification, a file system code was used. The definition of commit-atomicity in [17] coincides with I/O-refinement checking when the specification they use is an atomized version of the implementation expressed in Promela language of SPIN [15]. It mentioned about atomicity of commit block rather than method bodies of data structures. Our work is more general since we can use looser specifications using the termination condition and our checking is more detailed by using a part of the whole global store abstracted specifically. All four works assumed that the system reaches a clean state in which the state of the verified component is fully consistent

and there is no running transaction on the component. However, this is not true for today's components, such as Internet servers, which are accessed all the time, without letting the component be in a clean state. Our approach complements these works focusing on the view, part of the component that must be kept clean and consistent with the execution order of commit actions of methods all the time, even if the other parts are dirty, due to concurrency, but not crucial for the correct execution of the methods.

Chapter 3

CONCURRENT SOFTWARE COMPONENTS

In this thesis we focus on *concurrent software components*. A component¹ is an implementation of an abstract data structure in an object-oriented language like Java or C#. The abstract data structure makes available a set of operations each of which is implemented as a *public method* of the component. Methods return a single value, and exceptional terminations for methods are modeled by special return values. We distinguish between methods that are not allowed to modify data structure state, called *observer methods*, and ones that make modifications on the data structure state, called *mutator methods*.

Components are accessed in a multithreaded environment by a set of client threads, each of which executes a sequence of public methods of the component. In addition, there may be worker threads that are initiated by the data structure itself and perform tasks internal to the implementation like balancing a binary-tree as nodes are inserted into the tree.

This chapter illustrates an example component and explains syntax and semantics of components as well as concurrent programs that create and run components.

3.1 Example: Multiset

Throughout the thesis, we will use a Multiset data structure as our running example and demonstrate the concepts and the techniques on Multiset. A Multiset is a set of integers with multiplicities; there can be more than one element with the same integer value in the set. Its interface supports four operations: **Insert**(*x*) to insert an integer *x* into the Multiset, **InsertPair**(*x*, *y*) to insert two integers *x* and *y* atomically into the Multiset, **LookUp**(*x*) to check if *x* is an element of the Multiset, and **Delete**(*x*) to delete an integer *x* from the Multiset if it is an element of the Multiset.

Multiset has a specification that defines the abstract meaning of the contents and the operations of Multiset and an implementation that defines the internal data structure that

¹Throughout this thesis, we will use the word *component* to refer to a “concurrent software component”.

<pre> INSERT(x, returnValue) 1 if (returnValue == success) 2 M = M U {x}; 3 return returnValue; INSERTPAIR(x, y, returnValue) 1 if (returnValue == success) 2 M = M U {x, y}; 3 return returnValue; </pre>	<pre> DELETE(x, returnValue) 1 if (returnValue == success) 2 if (x in M) 3 M = M \ {x}; 4 return success; 5 return failure; LOOKUP(x) 1 return (x in M) </pre>
---	---

Figure 3.1: Specifications of the Multiset operations

actually stores the contents and implements the operations over the internal structure. The specification is important in order to declare what is the correct behavior for the implementation. We will use the specification when defining the refinement relation in Chapter 4.

3.1.1 The Specification

The specification of Multiset defines an abstract data structure, which is a multiset, denoted by a state variable M , of integers with multiplicities. The specifications of the operations are presented in Figure 3.1. The specification describes the semantics of the **Insert**(x), **InsertPair**(x, y), **LookUp**(x) and **Delete**(x) operations on M in a high-level, abstract way. In fact, the specification defines the semantics of the methods from the clients' point of view.

The specification defines the behavior of the methods of Multiset as if each method operates in a sequential environment in which atomic execution of the methods is guaranteed. The abstract atomic behavior of the Multiset interface guarantees that a successful execution of **InsertPair** inserts both x and y , and a failed execution inserts none of them. **Delete** deletes an integer if it succeeds and does nothing if it fails. If **LookUp** returns **true**, it means there exists an x in the Multiset at a time between just after **LookUp** is called and

before it returns. A return value of **false** implies the opposite of the case for **true**.

The specification allows mutator methods to fail nondeterministically. Any invocation of the **InsertPair** and **Delete** operations is allowed to terminate successfully or exceptionally, but exceptionally-terminating operations are required to leave **M** unchanged. This is done by updating the method signatures of the specification in order to take as input a nondeterministic return value. To the specification this appears to be a nondeterministically selected input. While performing runtime verification, the value of this input will come from the corresponding method execution in the implementation trace. For example the specification of **InsertPair** takes **returnValue** as an argument and if **returnValue** is **success**, it does the necessary modifications on **M**. Otherwise it leaves **M** unchanged. This style of writing specifications allows us to model failing behaviors of implementations, as we shall see below, due to concurrency. A return value of **failure** does not mean an erroneous behavior but an exceptional termination of the method that leaves the component state unchanged.

3.1.2 The Implementation

The implementation of Multiset given in Figure 3.2 uses an array **A**[0..**n**-1] to store the Multiset elements. The field **A**[**i**].**elt** denotes the element stored in **A**[**i**], and initially **A**[**i**].**elt** = *null* for all **i**. The boolean field **A**[**i**].**valid** indicates whether the element **A**[**i**] should be considered a member of the Multiset.

Both **Insert** and **InsertPair** use the private subroutine **FindSlot** to reserve room for the integers to be inserted. **FindSlot** looks for an available slot in the array for a single element **x**. If it finds one, it reserves the slot by setting its **elt** field to **x** and returns its index (it does not update the **valid** field). Otherwise it returns **-1**.

The more interesting method of the implementation is **InsertPair**. **InsertPair** first makes sure that there is room for both **x** and **y**. To accomplish this, it uses **FindSlot** to reserve room for each element. If no room is found for at least one of **x** and **y**, it terminates exceptionally and frees space it may have allocated. Otherwise, the code block in lines 9-14 atomically adds **x** and **y** to the Multiset by setting their **valid** bits. When **InsertPair**(**x**,**y**) terminates successfully, it adds **x** and **y** to the Multiset. If it terminates exceptionally, the multiset state should remain unchanged. In particular, it will be considered an error if only one of **x** or **y** is inserted into the Multiset.

We present the **InsertPair** method for its two features that are representative of real concurrent systems. First, a variety of concurrent data structures, such as file systems and storage systems, implement operations that require multiple resources for successful completion. This adds a significant amount of complexity to the implementation. The **InsertPair** operation mimics such operations and is therefore particularly suitable for illustrating the power of our approach. Second, this example shows the generality of our approach compared to existing techniques based on reduction (e.g., [20, 19, 18]) which cannot prove the correctness of the implementation of **InsertPair**.

3.2 Syntax

The software components we deal with are written in an object oriented programming language. We assume that we have a simple language with the key features of languages like Java, C++ or C#. Figure 3.3 shows an example syntax that supports concurrent executions, which is similar to CONCURRENT JAVA in [41]. The syntax supports class definitions with fields and methods, object creation, thread creation, thread synchronization via monitors, as well as conditional statements such as **if** and **while** statements. Each component is implemented as a class. The class may define and make use of other classes internally.

A component $\mathcal{C}$ is a tuple $\mathcal{C} = \langle \mathcal{M}, \mathcal{V}_G, B, \mathcal{V}_L \rangle$. Each component exports to its environment a public interface that declares a set $\mathcal{M}$ of *public* methods, where each $\mu \in \mathcal{M}$ denotes a method. $\mathcal{V}_G$ is the set of global variables that form the internal data structure and are visible to all methods of $\mathcal{C}$. $\mathcal{V}_G$ of a component $\mathcal{C}$ instance consists of the fields of shared objects. The global variables participate in the current state of the component that is visible to all the threads in the system. The type of a global variable ν_g is denoted by $cn.fn$, where cn and fn are the name of a class and one of its fields, respectively. Class fields are either of primitive types of the underlying programming language or object references. B is the set of (public or internal) method implementations, called *method bodies*. Each method $\mu \in \mathcal{M}$ is implemented by a method body $\mathcal{B}_\mu \in B$ in the implementation language. Let $\mathcal{V}_L$ be the set of all local variables defined by the method bodies in B . The types of local variables in $\mathcal{V}_L$ can be either of primitive types or object references.

We distinguish *mutator* methods, which modify the state of the component and whose

only return value is the termination status, from *observer* methods, which do not modify the state of the component but respond queries about the current state of the component.

3.3 Intermediate Form with Atomic Constructs

To simplify the exposition of our method, we present an action-based intermediate language, called **ActLang**. Each method body can be translated to an action-based implementation in **ActLang**. Figure 3.4 gives the language constructs for **ActLang**. In **ActLang** each method body consists of a set of atomically executable operations, each called an *action*, controlled by sequential composition and a control flow structure formed by **if** and **while** statements. We do not limit the types of expressions, which should include at least boolean, arithmetic and object oriented expressions.

Atomic Expressions are the expressions in the source code that will be regarded as atomic. Types of atomic expressions are determined by the developer. How to interpret operations in the syntax of the implementation language as actions depends on the granularity of analysis the programmer desires: In fine-grained analysis she can associate each byte code instruction an action, which is performed atomically by the underlying runtime environment (like the Java Virtual Machine). If she wants to do a coarser-grained analysis she can treat some data-structure specific operation, like setting **elt** and **valid** fields of a element in Multiset, or even a private method call as an action. Other language constructs, some of which are defined in terms of atomic expressions, corresponding to one of the action types in **ActLang** given in Figure 3.4 as an action.

During the course of converting a method implementation from its original syntax to **ActLang**, a set of fresh local variables may be created along with the ones already declared in the original syntax to help with decomposing complex expressions to atomic counterparts. An *init action* $\alpha^{init}(\nu_l := e)$ creates a local variable ν_l and initializes its value to e . Each atomic local computation is expressed as a *local action* $\alpha^{local}(\nu_l := e)$ that assigns the value returned from the evaluation of e to ν_l .

In **ActLang**, modifications to global variables are done by associating each global variable ν_g with a fresh local variable ν_f and performing local operations on ν_f that are even-

tually reflected to ν_g . If thread $\mathbf{t}$ manipulates ν_g it follows the pattern below:

$$\cdots, r(\nu_f := \nu_g), \cdots, \alpha^{local}(\nu_f := e), \cdots, \omega(\nu_g := \nu_f), \cdots$$

- 1) $r(\nu_f := \nu_g)$ creates a fresh local variable ν_f that corresponds to ν_g (or uses the existing one) and copies the value of $\nu_g \in \mathcal{V}_G$ into ν_f .
- 2) After a sequence of local computations, $\alpha^{local}(\nu_f := e)$ updates the value of ν_f by assigning to it the value of the expression e .
- 3) The changes to ν_f are reflected to ν_g by $\omega(\nu_g := \nu_f)$, which copies the value of ν_f to ν_g .

$\mathbf{atomic}(s)$ marks a consecutive block of actions in s to be executed atomically. $\nu_f := \text{Call}(\mathbf{t}, \mu, \chi)$ is a *call action* that represents an invocation of the method $\mu \in \mathcal{M}$ by a thread $\mathbf{t}$ with a tuple of actual parameters χ . The return value of the method is assigned to a fresh variable ν_f in the scope of the caller. A *return action* is a tuple $\text{Ret} = (t, \mu, \rho)$, where t is the identifier of the thread performing the method return, $\mu \in \mathcal{M}$ is the returning public method, and ρ is the value returned by the method.

ActLang uses Java monitors [3] as the synchronization mechanism. The actions $\text{acq}(\nu_g)$ and $\text{rel}(\nu_g)$ acquire and release the lock that is associated with an object pointed by $\nu_g \in \mathcal{V}_G$.

We call actions in the implementations that access $\mathcal{V}_G$ *globally-visible* actions. Concurrent execution of globally-visible actions makes $\mathcal{V}_G$ prone to race-conditions. Actions over $\mathcal{V}_L$ are called *locally-visible* actions. r, ω, acq and rel are globally-visible actions, while α^{local} , Call and Ret are locally-visible.

3.4 Semantics

We use *state transition systems* as the formal semantics of components. The semantics of a component whose implementation is translated into **ActLang** is expressed as a state transition system, which is a tuple $\langle \mathcal{V}_G, S, s_0, \Delta, \Lambda \rangle$:

- $\mathcal{V}_G$ is the set of *global program variables*.
- S is the set of *states* defined over $\mathcal{V}_G$ as described below.
- $s_0 \in S$ is the *initial state*. The initial state is the state of the component when it is instantiated.

- $\Delta : S \times \Lambda \rightarrow S$ is the transition function, and Λ is the set of actions that label the transitions, such that $\Delta(s_i, \alpha) = s_j$ iff $s_i \xrightarrow{\alpha} s_j$.

Each *state* is an assignment of a value of the correct type to each variable in $\mathcal{V}_G$. $s[\nu_g]$ returns the value of the variable ν_g in the state s , and $s[x/\nu_g]$ is s except the value of ν_g is given by $x \in \text{Values}$ where *Values* is set of values x can have. In fact we mean by a “state” of the component the state that is visible to all the threads in the program excluding local states of the threads.

3.5 Concurrent Programs

Let $Tid = \{\mathbf{t}_i | 1 \leq i \leq n\}$ be the set of unique thread identifiers for a particular execution of a multi-threaded program with n threads using $\mathcal{C}$. Thread identifiers uniquely identify threads; $\mathbf{t}$ is not assigned to another thread after the current thread with identifier $\mathbf{t}$ terminates.

A particular invocation of a method body $\mathcal{B}_\mu \in B$ represented by the sequence of actions associated with a thread $\mathbf{t}$ and lying between the call and return actions for $\mathcal{B}_\mu$ is called an *execution* ξ of $\mathcal{B}_\mu$. Formally, $\xi = \langle \mu, \mathbf{t}, \sigma \rangle$ where $\mu \in \mathcal{M}$ is the method executed by the thread $\mathbf{t}$, and $\sigma = \langle \alpha_1, \alpha_2, \dots \rangle$ is the sequence of actions performed by $\mathbf{t}$ during that particular invocation. ξ is said to have a signature $Sign_\mu(\xi) = \langle t, \mu, \chi, \rho \rangle$, where $t \in Tid$ is the thread that executed method $\mu \in \mathcal{M}$, χ is the set of actual parameter values, ρ is the return value for this execution of $\mathcal{B}_\mu$.

A multi-threaded execution $\mathcal{E}$ of $\mathcal{C}$ is obtained by interleaving a number of single-thread executions $\mathcal{T}_1, \dots, \mathcal{T}_n$ where $\mathcal{T}_i$ is for the thread $\mathbf{t}_i \in Tid$ a sequence of consecutive method executions. Threads $\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n$ form the environment $Env_{\mathbf{t}_i}$ of the thread $\mathbf{t}_i$ during $\mathcal{E}$. $\mathcal{E}$ can also be expressed as a sequence $\mathcal{E} = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots s_{n-1} \xrightarrow{\alpha_{n-1}} s_n$ of state transitions from an initial state s_0 . A *trace* π of $\mathcal{C}$ is the sequence $\langle \alpha_0, \alpha_1 \dots \alpha_{n-1} \rangle$ of actions performed during some execution $\mathcal{E}$. $\pi[i], 0 \leq i \leq n-1$ denotes the i^{th} action performed during $\mathcal{E}$.

The sequence $\langle s_0, s_1 \dots s_n \rangle$ of states observed during $\mathcal{E}$ is called a *run* r of $\mathcal{C}$. $r[i], 0 \leq i \leq n$ denotes the i^{th} state observed during $\mathcal{E}$. A state seen along a run is said to be *clean* if it does not lie between the call and return action of any method.

$\pi|_{\mathbf{t}}$ denotes the projection of the trace π to a particular thread $\mathbf{t}$ containing only the actions executed by $\mathbf{t}$. $\pi|_{\mathbf{t}}^{IO}$ is projection of $\pi|_{\mathbf{t}}$ to method call and return actions. We define the partial order $\triangleleft_{\pi}$ for a trace π such that method execution ξ_i comes before ξ_j in π , denoted $\xi_i \triangleleft_{\pi} \xi_j$, if the return action of ξ_i comes before the call action of ξ_j in π .

Λ -traces are defined by designating a subset $\Lambda \subseteq \text{Actions}$ of actions as *visible*. Call and return actions of public methods are required to be visible. Given a run r of the state transition system, the Λ -trace corresponding to r is the sequence of visible actions, i.e., actions belonging to Λ that take place during that run. When it is clear from the context, we simply refer to a trace and omit mention of Λ so that we always analyze projection of traces to Λ .

A sequence of actions by an application thread t is *well-formed* if

1. each call action $Call_p$ is eventually followed by a matching return action Ret_p , and
2. if $Call_q$ takes place after $Call_p$ but before Ret_p , then Ret_q takes place before Ret_p as well.

A trace τ is *well-formed* if for every application thread t $\tau|_t$ is well-formed. A run is *well-formed* if its corresponding trace is well-formed. In this thesis, we restrict our analysis to well-formed runs of state transition systems. A well-formed run is *method-atomic* if after every call action $Call(t, \mu, \chi)$ to a public method μ by an application thread t , no thread other than thread t performs an action until the corresponding return action $Ret(t, \mu, \rho)$ has occurred. A state transition system is *method-atomic* if all of its runs are method-atomic. Each portion of such a run associated with a particular method execution is called an *atomic fragment* of the run. A method-atomic state transition system is *deterministic* if whenever two atomic fragments of any two runs have the same starting state and the same method execution signature (with the same arguments and the same return value), they have the same final state as well. Thus, every trace of a method-atomic and deterministic state transition system is produced by a unique run. Note that this definition of determinism allows different possible return values for a method invocation at a given state. It only requires that given the return value, the final state be uniquely defined.

This chapter introduced the syntax and semantics of software components, the focus of in our verification study. Our aim is to verify that a component provides an abstract

atomic interface to its clients. The following chapter formalizes the conditions under which the implementation of the component has the abstract atomicity property.

<pre> FindSlot(x) 1 for (i=0; i<n; i++) 2 synchronized(A[i]) { 3 if (A[i].elt == null) { 4 A[i].elt = x; 5 return i; 6 } 7 } 8 return -1; Insert(x) 1 i = FindSlot(x); 2 if (i == -1) 3 return failure; 4 A[i].valid = true; 5 return success; Delete(x) 1 for (i=0; i<n; i++) 2 synchronized(A[i]) { 3 if ((A[i].elt == x) && (A[i].valid)) { 4 A[i].valid = false; 5 A[i].elt = null; 6 return success; 7 } 8 } 9 return failure; </pre>	<pre> InsertPair(x,y) 1 i = FindSlot(x); 2 if (i == -1) 3 return failure; 4 j = FindSlot(y); 5 if (j == -1) { 6 A[i].elt = null; 7 return failure; 8 } 9 synchronized (A[i]) { 10 synchronized (A[j]) { 11 A[i].valid = true; 12 A[j].valid = true; 13 } 14 } 15 return success; LookUp(x) 1 for (i=0; i<n; i++) 2 synchronized(A[i]) { 3 if ((A[i].elt == x) && (A[i].valid)) 4 return true; 5 } 6 return false; </pre>
--	--

Figure 3.2: Implementation of the Multiset operations

P	$::=$	$defn^* e$	(program)
$defn$	$::=$	$class\ cn\ body$	(class declaration)
$body$	$::=$	$extends\ c\ \{ field^* method^* \}$	(class body)
$field$	$::=$	$t\ fn = e$	(field declaration)
$method$	$::=$	$t\ mn(arg^*) \{e\}$	(method declaration)
arg	$::=$	$t\ x$	(variable declaration)
t	$::=$	$c \mid p$	
p	$::=$	$int \mid float \mid boolean \mid \dots$	(primitive type)
c	$::=$	$cn \mid Object$	(class type)
e	$::=$	$new\ c$	(allocate*)
		$\mid x$	(variable*)
		$\mid e.f$	(field access*)
		$\mid e.f := e$	(field assignment*)
		$\mid e == e \mid e < e \mid e >= e \dots$	(comparison expression*)
		$\mid e \vee e \mid e \wedge e \mid \neg e$	(boolean expression*)
		$\mid e.mn(e^*)$	(method call)
		$\mid synchronized\ e\ in\ e$	(synchronization)
		$\mid fork\ e$	(fork)
		$\mid if(Bexp?s : s)$	(if statement)
		$\mid while(Bexp, s)$	(while statement)
		$\mid atomic(s)$	(atomicity annotation)
cn	$\in$	$ClassNames$	
fn	$\in$	$FieldNames$	
mn	$\in$	$MethodNames$	
x, y	$\in$	$VariableNames$	

Figure 3.3: An example syntax for concurrent component implementations

$$\begin{aligned}
s &::= \Lambda \mid \mathbf{atomic}(s) \mid s; s \mid \mathbf{if}(\nu_l ? s : s) \mid \mathbf{while}(\nu_l, s) \\
e &::= c \mid \nu \mid \bar{e} \\
\Lambda &::= \alpha^{init}(\nu_l := e) \mid \alpha^{local}(\nu_l := e) \mid acq(\nu_g) \mid rel(\nu_g) \\
&\quad \mid r(\nu_f := \nu_g) \mid \omega(\nu_g := \nu_f) \mid Call(\mu, e^*) \mid Ret(\nu_l, Call(\mu, e^*)) \\
\nu &::= \nu_g \mid \nu_f \\
\nu_g &\in \mathcal{V}_G \\
\nu_l, \nu_f &\in \mathcal{V}_L \\
\bar{e} &\in \textit{Atomic Expressions} \\
c &\in \textit{Constants}
\end{aligned}$$

Figure 3.4: The constructs of **ActLang**

Chapter 4

REFINEMENT CRITERIA FOR CONCURRENT COMPONENTS

This chapter formally describes the refinement relation – the correctness criteria for component implementations. For each component $\mathcal{C}$ our technique requires an implementation of $\mathcal{C}$, denoted I , and a specification of $\mathcal{C}$, denoted S . I is the source code of the component written in a programming language like Java or C#. I is designed and implemented in a way to be run in a highly-concurrent environment and still provide guarantees required by refinement criteria. The aim of this work is to check that these guarantees are really provided by I . S specifies the behavior of the methods of $\mathcal{C}$ in a sequential environment. We have no restriction on how S is written. S can be given in any language ranging from a high-level language like TLA [6] to the same programming language as I is written. We require that S be executable. We require our specifications to be method-atomic and have a deterministic state transition system.

Abadi and Lamport [10] described refinement based on trace inclusion, stating that a component $\mathcal{C}_i$ refines $\mathcal{C}_j$ iff for each behavior of $\mathcal{C}_i$ there is a behavior of $\mathcal{C}_j$ with the same sequence of externally visible states, allowing stuttering. A behavior satisfies a property under stuttering if the property is satisfied for behaviors obtained from the current one by adding or deleting transitions that leave the state unchanged [10].

One of our key contributions is definition of two notions of refinement, I/O-refinement and view-refinement, which are expressed as practically checkable criteria for software components. These definitions were motivated by our practical experience with industrial-scale designs. The refinement relation gives the conditions for I to correctly implement S in a concurrent environment, thus, for I to satisfy abstract atomicity.

I/O-refinement and view-refinement are defined by considering the observation that the clients of $\mathcal{C}$ are unaware of the internals of $\mathcal{C}$ and query the data structure by only calling its methods. For I/O-refinement an observable behavior is a sequence of method call and return actions, which gives method parameters and return values, and defines visible states

in Abadi and Lamport's definition as the abstract states modified/observed by the clients calling the methods of the component. The criterion for I/O-refinement is defined over execution traces of I and S whose equivalence is stated in terms of matching call and return action sequences. Unlike I/O-refinement, view-refinement defines observable behaviors as abstract state transitions caused by executions of actions, and defines visible states as the abstracts states observed after each action execution.

We also emphasize that the distinction between what specifies the expected sequential behavior in atomicity [20, 19] and abstract atomicity in our approach can be important for concurrent systems and makes atomicity unnecessarily restrictive in some cases. It is possible that the implementation methods terminate exceptionally due to effects of concurrency, i.e., the resource contention among methods. In an atomic execution of this implementation, there is no resource contention between methods. Therefore, the concurrent execution containing the exceptional terminations of methods are not equivalent to any atomic execution of the implementation. Since it uses sequential executions of the same implementation, atomicity is sometimes too stringent and may declare certain acceptable system behavior as erroneous. If a specification that allows some method calls to terminate exceptionally but without modifying the current state of the component had been used for the analysis, the execution treated erroneous would be treated valid. The ability to use specifications that more closely model concurrent executions makes refinement a more appropriate correctness criterion for concurrent systems.

4.1 Refinement as a Correctness Criterion

We define a refinement relation between the state transition systems that define the semantics of the implementation I and specification S of a component, respectively. The relation is defined between Λ -traces of I and S so the general notion of the refinement is called Λ -refinement.

Definition 1 I Λ -refines S , denoted $I \preceq_{\Lambda} S$, if for every Λ -trace τ of I , there is a Λ -trace τ' of S such that

1. $\tau|_t = \tau'|_t$ for all $t \in Tid_{app}$, and
2. for all method executions ξ and ξ' , $\xi \triangleleft_{\tau} \xi'$ implies $\xi \triangleleft_{\tau'} \xi'$.

The set of actions in Λ gives different ways of selecting what portion of the state is externally visible. Λ -refinement is utilized to define various notions of refinement by declaring different sets Λ as visible that contain actions common to I and S traces. As a consequence, the same checking procedure for Λ -refinement can be used for all these notions of refinement (see Section 5.7). For example, clients of a component modify and query the state of the component by calling its methods. Call and return actions along an execution carry the information about what operations are performed on the component and what are the results of these operation, respectively. Clients are unable to observe the other actions between the call and return actions of a method execution. Thus, in our simplest notion of refinement, *I/O-refinement*, call and return actions are the only visible actions.

4.2 The Need for I/O-refinement

This section introduces the idea of I/O-refinement around the Multiset example. The idea of I/O-refinement is illustrated in Fig. 4.1. The actions that form a trace of the implementation are shown on the left half of the figure from top to bottom in the order they happen in time. The colors of the boxes representing actions indicate different threads performing them. The execution shows four method calls `LookUp(3)`, `Insert(3)`, `Insert(4)` and `Delete(3)` being concurrently executed by four different threads.

The execution is correct if there is any execution of the specification S of Multiset with the same method calls and return values. Since the four method executions overlap with each other, the corresponding methods in S could be serialized in any one of $4!$ ways. A simple but naive testing method for determining the correctness of the return value of `LookUp(3)` would require evaluating $4!$ possible executions of S and finding one that matches the current trace. Clearly, this method would not scale as the number of methods being executed concurrently increases.

Additionally, consider a verification thread consisting of calls to `LookUp` run after the termination of the threads in Figure 4.1. Since the executions of `Insert(3)` and `Delete(3)` overlap, the verification thread would have to consider `Insert(3)` and `Delete(3)` in both orders of execution. Thus both possible return values of `LookUp(3)` (`true` and `false`) are treated correct. However, `Delete(3)` completes its operation after `Insert(3)` inserts 3 so `LookUp(3)` should return `false`.

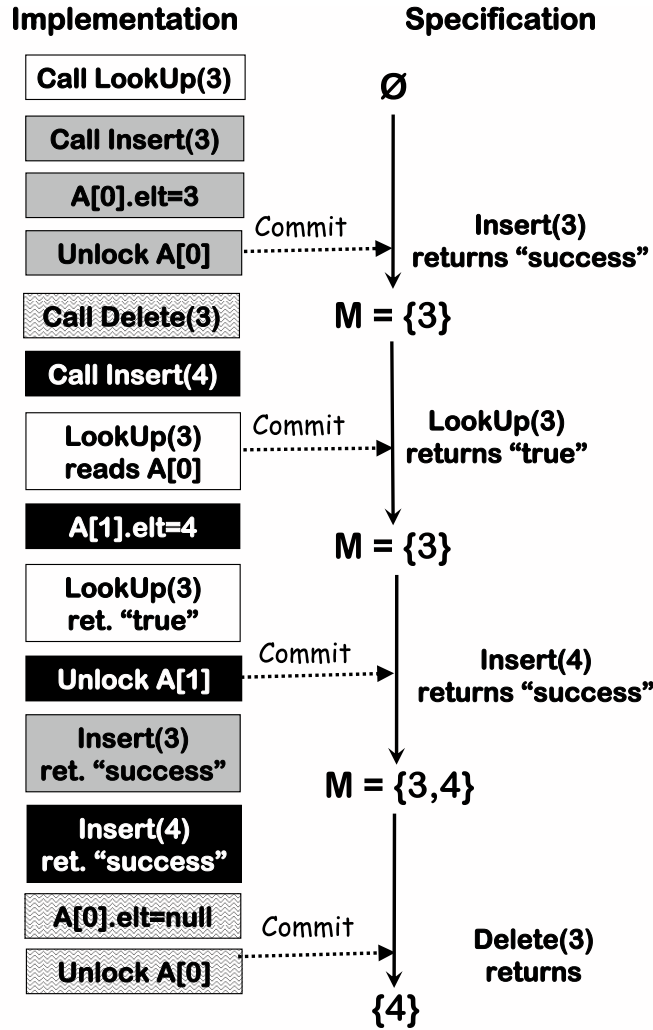


Figure 4.1: Refinement of Multiset.

Our solution to this problem has two parts. First, we require that the programmer specify a unique *commit action* for each method execution. The concurrent method executions are then deemed to have been serialized in the order of occurrence of the commit actions. This order of commit points gives us a *witness ordering* of methods in that execution. Second, we use the witness ordering in an execution of the implementation to drive the execution of the specification. Section 5.1 explains how commit points are derived and used in runtime refinement checking in detail.

Figure 4.1 also shows the state transitions of the specification in the right half. At each commit action, the method that is committing and its return value are used to execute the

specification. The return value may be derived by looking ahead in the implementation's execution, which can be done by waiting the associated method to terminate. From its current state the specification must take a transition associated with the committing method and its return value. If this is not possible, e.g., if a call to `LookUp(y)` method has a return value of “false” when the specification state indicates that `y` is in the Multiset, a refinement-violation is detected. Observe that, in the example in Figure 4.1, although the execution of `LookUp(3)` by the gray thread starts before the execution of `Insert(3)` and ends before the execution of `Insert(3)` ends, `LookUp(3)` returns “true” since its commit action comes after that of `Insert(3)`.

To see how the notion of refinement improves upon existing testing techniques for concurrent software, consider the scenario introduced above: `LookUp` being called after the termination of the threads in Figure 4.1. Using the order of commit points, we obtain a witness serialization. The witness serialization allows us to determine that `Delete(3)` occurs after `Insert(3)` and a `LookUp(3)` that occurs after the methods in Fig. 4.1 should return `false`.

4.3 I/O-refinement

I/O-refinement between I and S is denoted by $I \preceq_{I/O} S$. Let $\pi|_{\mathbf{t}}^{I/O}$ be projection of a trace π to only call and return actions that belong to thread $\mathbf{t}$. The formal definition of I/O-refinement is as follows:

Definition 2 I I/O-refines S , denoted $I \preceq_{I/O} S$, if for each execution $\mathcal{E}_I$ of I with associated trace π_I , there exists an execution $\mathcal{E}_S$ of S with associated trace π_S such that

1. $\pi_I|_{\mathbf{t}}^{I/O} = \pi_S|_{\mathbf{t}}^{I/O}$ for all $\mathbf{t} \in Tid$, and
2. for all method executions ξ and ξ' in π_I and π_S , $\xi \triangleleft_{\pi_I} \xi' \Leftrightarrow \xi \triangleleft_{\pi_S} \xi'$.

In other words, for an execution $\mathcal{E}_I$ and an equivalent execution $\mathcal{E}_S$, for each thread $\mathbf{t}$, π_I and π_S contain the same sequence of call and return actions and the happens-before relation between method executions are preserved. For each execution of a component that satisfies I/O-refinement, we can give an execution of the specification that runs sequentially and behaves in the same way in terms of method calls and returns.

4.4 The Need for View-refinement

```

FindSlot(x)
1 for (i=0; i<n; i++) {
2   if (A[i].elt == null) { // A[i] should have been locked
3     synchronized(A[i]) {
4       A[i].elt = x;
5       return i;
6     }
7   }
8   return -1;

```

Figure 4.2: Buggy implementation of FindSlot.

I/O-refinement guarantees the conformance of $\mathcal{E}_I$ to some $\mathcal{E}_S$ only at method boundaries using return values, and this decreases “observability”. Observability is an important issue because most of the mutator methods have no return values, and most of the information used to reason about I/O-refinement is obtained from observer method executions. Thus, if there is only a small number of observer methods in a trace, the observability suffers since the return values can agree even if the mutators modify the state of $\mathcal{C}$ in an inconsistent way; I/O-refinement does not check the consistency of states from both implementation and specification traces.

As an extreme example of the limited visibility provided by testing and I/O-refinement, consider a test program for the Multiset example that only makes calls to `InsertPair`. Since no observer methods are called, the runtime checks for I/O-refinement would pass trivially for all such tests since the call actions will trivially match and no return actions have return value. For useful testing or I/O-refinement checking, frequent calls to observer methods must be performed. But this might reduce the degree to which mutator methods are executed concurrently and make it less likely that an existing error will get triggered. Furthermore, calls to the observer methods might not get scheduled at the most interesting points. The following section presents view-refinement, a correctness condition for concurrent programs which augments I/O-refinement with more visibility into the data structure state. This

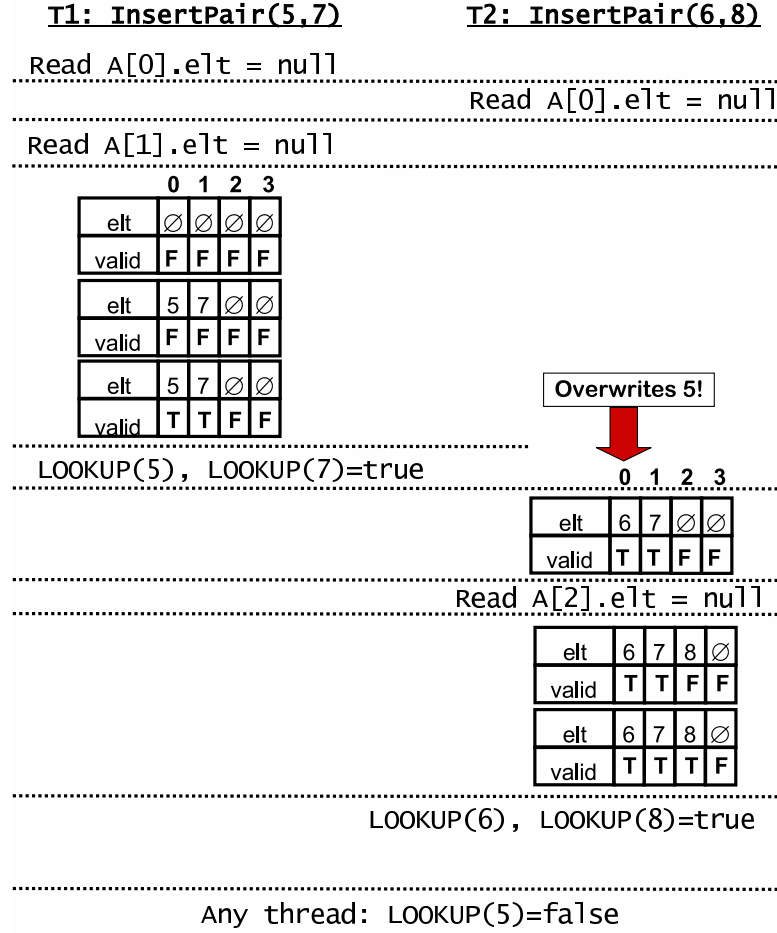


Figure 4.3: Refinement violations in the buggy version of Multiset.

extra visibility enables us to get more thorough checking from each test run.

To demonstrate the superiority of view-refinement over I/O-refinement, a buggy implementation of `FindSlot` is given in Figure 4.2. In this buggy implementation, `A[i]` is not locked while it is being read. Consequently it is possible for two concurrently executing `FindSlot` operations to both check `A[i].elt` at line 2 and conclude that the same index `i` is available, since neither have to hold the lock for `A[i]` in line 2.

Figure 4.3 shows a concurrency error that is caused by this bug. In the scenario depicted in the figure, there are two threads T1 and T2, each performing two `InsertPair` calls to the same Multiset instance. Because both threads reads the first slot `A[0]` of the array as empty, Thread T2 overwrites the value 5 that thread T1 inserted into `A[0]` beforehand. At

the end, the state of the component contains only three of the integers. Each thread makes calls to `LookUp` to check that if their `InsertPair` calls succeeded. Because `LookUp` methods get interleaved in the execution as showed in the figure, all of them return `true`. However, the last state of the component is inconsistent with the return values of the `LookUp` calls. If a test program had included a `LookUp(5)` that gets interleaved after both `InsertPair` operations complete, because the specification state would be $\{5,6,7,8\}$ at that point, the specification would require the return value be `true`, while, in the implementation, the return value would be `false`. Then I/O-refinement would catch the error by examining these inconsistent return values from the `LookUp(5)` calls. However, because the observer methods are not scheduled in this way, the error can not be detected by looking at only the return values of the observer methods and I/O-refinement misses the error. View-refinement solves this problem by analyzing the correspondence between implementation and specification states at each commit point along the execution traces. Even though I/O-refinement can catch the error by if calls to `LookUp` get interleaved at the proper places, view-refinement can identify errors exactly when they happen without need for any observer methods. For example in this case view-refinement is able to detect the error in the first implementation state that differs from what is required.

4.5 View-refinement

4.5.1 View Definition of A Component

The clients of $\mathcal{C}$ are under the impression that they access and manipulate an abstract data structure using the methods in $\mathcal{M}$. Conceptually, this data structure is represented by a *view variable* `view` and an abstract state of this data structure resulting from a valuation of `view` is called the *view* of $\mathcal{C}$. The key idea in view-refinement is to introduce hypothetical “view” variables `viewI` and `viewS` into the implementation and specification, respectively. Intuitively, the values of `viewI` and `viewS` extract the contents of the data structure from the implementation and specification states.

The view variables are computed as a function of the data structure state and abstract away information that is not relevant to what applications can observe. In other words, the view captures the most abstract state of $\mathcal{C}$ that can be observed by external clients through method calls. The abstract states are defined as $\tilde{\mathcal{S}} = \{\text{view} \mid \text{view} : \text{ViewVars} \rightarrow$

`ViewValues`}. A view variable can point to another (possibly more abstract) data structure whose state captures the view of the actual component. For example, consider a binary search tree that stores integers in it. It provides `Insert(int x)`, `Delete(int x)` and `LookUp(int x)` methods that allow its clients to add, remove and search for an integer. Although it contains internal structures to form a binary tree, it serves to the outside an integer set. Thus after abstracting away the extra details of the tree, view^I points to an integer set. A fully-synchronized version of `java.util.Vector` that contains instances of `java.lang.Integer` can be given as the specification to the binary tree. view^S for the specification points to a set of integers while the details that serve as an extendable vector are abstracted away.

In order to accomplish the mapping from the implementation state space to the abstract state space, the developer is asked to specify how the view variables are to be computed. We have found the definition of `view` to be an iterative process which leads to a better understanding of why the program guarantees atomicity of methods. The developer provides an abstraction function $\Theta : S \rightarrow \tilde{S}$ such that `view` = $\Theta(s)$ uses the current implementation state s to extract the current value of the view variable `view`. `view` then points to a data structure that captures the current view of the component.

We extend this definition to sequences of states as $\Theta(r)$ returns the ordered sequence of abstract states such that $\Theta(r) = \{\text{view}_i | \text{view}_i = \Theta(r[i]), 1 \leq i \leq |r|\}$. Assuming the formula ψ over $\mathcal{V}_G$ represents a subset of S , $\Theta(\psi)$ represents the corresponding set of views the component can have.

The 2-tuple $\langle \tilde{S}, \Theta \rangle$ forms a *view definition* for $\mathcal{C}$. Our technique requires the developer to give the view definition of the component. An abstraction function Θ is sufficient for the view definition. Syntactically, Θ is written as a regular method of $\mathcal{C}$ in Java, which runs over $\mathcal{V}_G$ and extracts a data structure representing the view. Naturally, the return type of Θ specifies the possible values the view variable can have, in other words the abstract state space of the component. For example the abstraction function for Multiset is given in Figure 4.4. Since it returns an object of type `Map<int, int>`, which maps each unique integer to the number of its appearances in Multiset. Thus, the states of `Map<int, int>` can represent an abstract state space of the view variable of Multiset.

The view of a component $\mathcal{C}$ is a canonical version of the abstract state of $\mathcal{C}$. The

```

Map<int,int> AbstractionFunction() {
// create an empty integer set
  Map<int, int> M = new Map<int, int>();
  for(int i=0; i<n; ++i){
    // include a slot if it is marked as valid
    if(A[i].valid) {
      // check the invariant
      Assert(A[i].elt != null);
      // include the content of the slot into the view
      int count = M.get(A[i].elt);
      M.put(A[i].elt, count + 1);
    }
  }
  return M;
}

```

Figure 4.4: Abstraction function of the Multiset example

abstraction function is written so that it canonizes the abstract state while extracting the parts relevant for the view. As an example consider the two vector-based representations and one tree-based representation for Multiset in Figure 4.5. Although their internal states are different the contents of the data structures that are abstracted as the view are the same: $M = \{1, 3, 5, 6\}$.

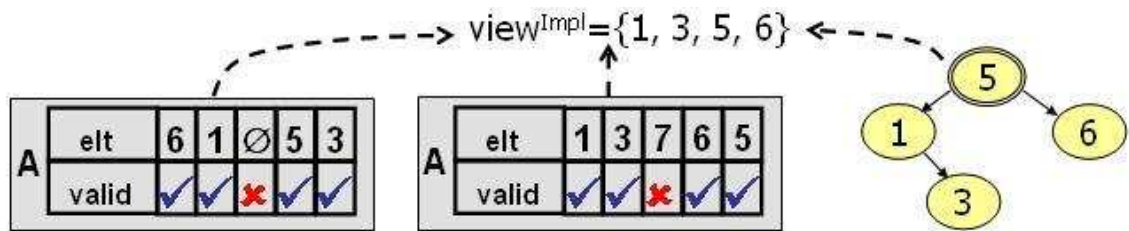


Figure 4.5: Three different representation for Multiset whose views are the same.

Although writing an abstraction function is straightforward for many cases, it may be difficult for some complex implementations to write one that works in dirty states, states in which at least one method of $\mathcal{C}$ is in progress. The result of an abstraction function

Θ written for only commit points may not be a valid abstract state because of partially modified contents of the underlying data structure by other threads at the commit point. In fact one could write Θ that will return a clean view for all the states of the component including the dirty ones, although this complicates the work that must be performed by Θ . Section 5.8 elaborates this problem and shows a method that allows us to make writing Θ more convenient for the developer at commit points.

4.5.2 Definition of View-refinement

View refinement between I and S is denoted by $I \preceq_{(\Theta_I, \Theta_S)} S$. Two runs r and r' are *equivalent under stuttering*, denoted $r \simeq r'$, if r and r' can be partitioned into k nonempty subsequences $r_1, \dots, r_k$ and $r'_1, \dots, r'_k$ such that the states in each subsequence pair r_i and r'_i are equivalent [10]. Let $\Theta_I : S_I \rightarrow \tilde{\mathcal{S}}$ and $\Theta_S : S_S \rightarrow \tilde{\mathcal{S}}$ be abstraction functions for two components I and S . *view-refinement* $\preceq_{(\Theta_I, \Theta_S)}$ is defined as follows:

Definition 3 I view-refines S , denoted $I \preceq_{(\Theta_I, \Theta_S)} S$, if there exist view definitions $\langle \tilde{\mathcal{S}}_I, \Theta_I \rangle$ and $\langle \tilde{\mathcal{S}}_S, \Theta_S \rangle$ such that, if for each execution $\mathcal{E}_I$ of I with associated traces π_I and r_I , there exists an execution $\mathcal{E}_S$ of S with associated π_S and r_S such that

1. $\pi_I|_{\mathbf{t}}^{IO} = \pi_S|_{\mathbf{t}}^{IO}$ for all $\mathbf{t} \in \text{Tid}$,
2. $\Theta_I(r_I) \simeq \Theta_S(r_S)$, and
3. for all method executions ξ and ξ' in π_I and π_S , $\xi \triangleleft_{\pi_I} \xi' \Leftrightarrow \xi \triangleleft_{\pi_S} \xi'$.

Note that the definition of view-refinement includes I/O-refinement as a first condition. The second condition increases observability by not requiring observer methods but seeking correspondence between views of the implementation and state specification along the whole execution.

To check view-refinement easily, we can express the criteria in Definition 3 as a special case of Λ -refinement by capturing the state information as annotations to commit actions. These annotations are values of view variables $\mathbf{view}^I$ and $\mathbf{view}^S$ computed using the abstraction functions of I and S , respectively. Along the trace π_I with associated run r_I , this is achieved by adding to a commit action α_i $\Theta_I(s_{i+1})$, the result of the abstraction function

on the next state $s_{i+1} = r_I[i + 1]$ after the commit action is performed. Let $\pi_I|_{\mathbf{t}}^{View}$ be the projection of a trace π_I to only call, return and commit actions that belong to thread $\mathbf{t}$. $\pi_S|_{\mathbf{t}}^{View}$ is obtained from π_S similarly using Θ_I . The following fact indicates that the criteria in Definition 3 is a special case of Λ -refinement:

Fact 4 *I view-refines S , denoted $I \preceq_{(\Theta_I, \Theta_S)} S$, if for each execution $\mathcal{E}_I$ of I with associated π_I , there exists an execution $\mathcal{E}_S$ of S with associated π_S such that*

1. $\pi_I|_{\mathbf{t}}^{View} = \pi_S|_{\mathbf{t}}^{View}$ for all $\mathbf{t} \in Tid$, and
2. for all method executions ξ and ξ' in π_I and π_S , $\xi \triangleleft_{\pi_I} \xi' \Leftrightarrow \xi \triangleleft_{\pi_S} \xi'$.

With the addition of the auxiliary variable $\mathbf{view}^I$ to the implementation, we get useful checking even with a test program that has no calls to `LookUp`. Tracking the views is more likely to expose errors and provide early warnings as the following examples demonstrate. Consider the bug scenario in Fig. 4.3. The first call to `LookUp(5)` by thread T1 returns the expected value of `true`. If a second call to `LookUp(5)` had not been performed at the end of the test program by any thread, the error would have gone undetected. This is a likely scenario since T2 is likely to only check the results of its own operations. By contrast, computing $\mathbf{view}^I = \{6, 7, 8\}$ and $\mathbf{view}^S = \{5, 6, 7, 8\}$ at the commit point of `InsertPair(6, 8)` and comparing them would have revealed the bug immediately.

Now consider another example in which a thread in the test program calls `InsertPair(x, x)` to insert two copies of $\mathbf{x}$ into the Multiset, but, because of the error in `FindSlot`, only one $\mathbf{x}$ gets inserted into the array $\mathbf{A}$. To expose the error through testing or I/O refinement checking, we need an execution that follows this call with `Delete(x)`, followed by `Lookup(x)` with no other calls between them to mutators with $\mathbf{x}$ as the argument. The probability of this during a test program not written to target this error is low. Even if such a test scenario were exercised, if the `Insert`, `Delete` and `LookUp` operations were far apart in the execution, by the time `Lookup(x)` returns an unexpected result, it would be difficult to locate the origin of this error. View refinement checking detects this error immediately at the commit action of `InsertPair(x, x)`.

This chapter formalized the refinement criteria that allows us to reason about abstract atomicity. The two notions of refinement, I/O-refinement and view-refinement, are defined

so that they can be checked at runtime for large complex software. We selected runtime verification due to computational cost of impreciseness of static verification, which makes it hard to apply industrial-scale programs. The following chapter explains our approach that checks refinement for each execution of a component, meaning that whether that execution conforms to an execution of the specification with respect to refinement criteria.

Chapter 5

CHECKING REFINEMENT AT RUNTIME

We check I/O-refinement and view-refinement criteria for components at runtime. The fundamental reason for doing runtime checking is to be able to handle industrial-scale software. Today's static techniques [71, 44, 72] are not good at doing precise analysis for complex programs. In particular, the refinement criteria requires a thorough static analysis of concurrent behaviors of the component, which has not been optimized sufficiently for large programs [13].

Runtime checking of refinement consists of collecting runtime information about a concurrent execution of a software component and analyzing it to check whether it satisfies refinement criteria. The runtime information is obtained by running a multithreaded test program that makes concurrent method calls to the component and recording the actions performed during the execution of the test. A trace, denoted π , is obtained from each execution ξ to be analyzed for I/O-refinement and view-refinement criteria by the refinement checker module. This chapter explains in detail the runtime checking procedure.

5.1 Commit Points

Using commit points are one of the strengths of our approach that makes it superior to pure testing. Suppose that the developer wants to apply a pure testing-based approach to check that an execution consisting of n concurrently running methods matches to some execution of the specification. She may have to try $n!$ specification traces formed by different serializations of the methods to find a matching method call sequence. This makes the verification time for a single execution big and it adds to the cost if checking one specification trace takes much time.

Commit points are crucial in that their order gives us the “witness interleaving” of the specification. Then an implementation trace is checked only with respect to the witness interleaving derived from that trace. In practice, the developer declares commit points by

<pre> InsertPair(x,y) 1 i = FindSlot(x); 2 if (i == -1) 3 return failure; 4 j = FindSlot(y); 5 if (j == -1) { 6 A[i].elt = null; 7 return failure; 8 } 9 synchronized (A[i]) { 10 synchronized (A[j]) { 11 A[i].valid = true; 12 A[j].valid = true; 13 } // commit point 14 } 15 return success; </pre>	<pre> Insert(x) 1 i = FindSlot(x); 2 if (i == -1) 3 return failure; // commit point 1 4 A[i].valid = true; // commit point 2 5 return success; Delete(x) 1 for (i=0; i<n; i++) { 2 synchronized(A[i]) { 3 if ((A[i].elt == x) && (A[i].valid)) { 4 A[i].elt = null; 5 return success; 6 } 7 } //commit point 1 (only after line 5 is run) 8 } 9 return failure; // commit point 2 </pre>
--	--

Figure 5.1: Commit points for Multiset operations

designating certain lines in the source code with specific annotations. During the execution of a method, running an action at a line annotated as a commit point brings that method to its commit point and makes that action the commit action for that invocation of the method. The programmer must make sure that, for each method, *exactly one* action is marked as the commit action for every execution path through the method code. This is accomplished by associating with each commit point annotation a condition under which execution of the associated action becomes a commit action. This condition can be typically expressed in terms of thread-local variables and then can be evaluated at runtime. The reader can see INSERT method in Figure 7.2 for an example to conditional commit point annotations.

Commit actions are hints to refinement checking tools by the programmer. There is no formal procedure for automatically selecting commit points. If a certain design pattern is followed, a method for selecting them can be devised. If the user has provided a view defini-

tion $\langle \tilde{\mathcal{S}}, \Theta \rangle$, the commit action in an execution of a mutator method by a thread is typically the first action which makes visible to other threads the modified abstract data structure state. The order of the commit actions in time must coincide with the application's view of how the state of the abstract data structure represented by the component transforms over time. Intuitively, a good guess for the commit action is the first lock release following the last modification to a variable in the support of view.

The code for the Multiset implementation in Figure 5.1 shows the commit points of the `Insert`, `InsertPair` and `Delete` methods. The commit point selection for these methods fits the description for appropriate commit points above:

- There are two possible termination condition for `InsertPair` method: either both `FindSlot` calls succeed to allocate slots for `x` and `y` and then the `valid` fields are set or at least one of the `FindSlot` calls fail to allocate a slot and the method fails to insert `x` and `y`. In the former case, the action that releases the lock of `A[i]` after the `valid` fields of `A[i]` and `A[j]` are set is the commit action (at commit point 1). This action is selected as the commit action because after that action any other method can see `x` and `y` as in the contents of the multiset. In the latter case, the action that returns `failure` is the commit action (commit point 2).
- There are two possible termination condition for `Insert` method: either `FindSlot` fails to find a free slot or it is able to find a free slot to put `x` in and validates the slot. In the former case, the return action that returns `failure` is the commit action (at commit point 1). In the latter case, the action that sets the `valid` field of the allocated slot is the commit action (commit point 2). This action is selected as the commit action because after that action any other method can see `x` inserted into the allocated slot.
- There are two possible termination condition for `Delete` method: either it finds a previously allocated slot with `elt` field set to `x` or it fails to find such a slot. In the former case, the action that releases the lock of `A[i]` after the slot is deallocated is the commit action (at commit point 1). This action is selected as the commit action because after that action any other method can see `A[i]` as a free slot. In the latter case, the action that returns `failure` is the commit action (commit point 2).

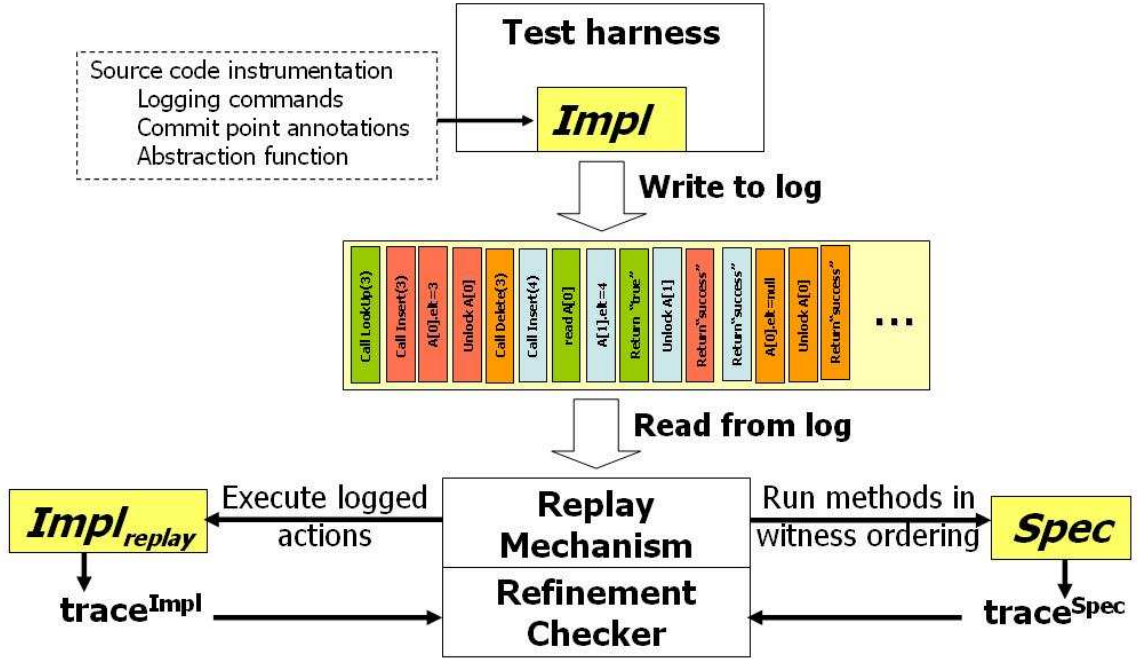


Figure 5.2: The communication between the test program and the checker through the log.

The runtime refinement check described could fail either (1) because the implementation truly does not refine the implementation or (2) because the witness interleaving obtained using the commit actions is wrong. Comparing the witness interleaving with the implementation trace reveals which one of these is the case. If the witness interleaving is wrong, meaning that the commit points are not correctly placed, the programmer must re-examine her selection of commit points. Then she must modify the commit points or must refine the conditions associated with those. Otherwise the programmer can decide on a real error be caught. We have found this iterative process very useful for debugging code that is in development and for improving our understanding of method implementations.

5.2 Separating the Program from the Checker via Logging

The first phase of the checking technique is collecting runtime information for executions of the component implementation I . Runtime information for an execution ξ consists of information about what actions, along with their input/output arguments, are performed during ξ from the initial state of program until the program terminates. It is desirable for

a runtime verification tool not to modify the concurrency characteristics of I significantly while simultaneously (1) collecting and (2) analyzing runtime information. This undesirable situation could happen if a large amount of instrumentation overhead is introduced or application threads block waiting for access to the instrumentation module. To interfere minimally with the running methods of I , we run refinement checking on a separate thread, we call the checker thread. The checker thread is informed about the implementation's trace through a *log*. Figure 5.2 depicts the communication between the test program that runs the component and the checker thread that checks the execution generated by testing. As the test program runs, the worker threads write entries to the log as they execute methods of I . The checker reads from the log the relevant information for refinement checking and behaves accordingly. Section 6.2 explains how logging is performed in practice by our refinement checking tool.

Formally, a log $\mathcal{L}$ for a trace π is a totally-ordered finite sequence of records $\mathcal{L} = rec_0, \dots, rec_n$ where rec_i holds information about the type and input/output parameter values of $\alpha_i = \pi[i]$. Type of α_i can be a globally visible mutator action on $\mathcal{V}_G$ (we do not record observer actions as explained in Section 5.4), a method call or a method return action. During the execution, some actions in the log are marked as commit actions to be handled specifically by the checker. For I/O-refinement checking, the log is required to contain all call and return actions, and the actions marked as commit actions. For view-refinement checking, all mutator actions are required in addition to the call, return and commit actions.

Working independently on the logged execution traces allows us to do online or offline check. In online checking, the verification thread runs in parallel with the test program, as the execution is communicated via log, which is both being written and read at the same time. The verification thread can run offline by reading the log saved after the test program terminates.

5.3 Verifying Mutator Methods

This section gives details of our method for checking refinement for mutator methods. The case for observer methods is explained in the next section. The two cases are studied separately because the derivation of witness ordering is performed differently for mutator and observer methods. A single, unique witness ordering can be obtained in a straightforward

manner for mutators, while it is expensive for observers as explained in the next section. This section and the following one introduce the checking procedure, and Section 5.7 gives the full algorithm in detail and explain it line by line.

Suppose that an execution of the test program generates an execution log of I containing method call and return actions, and mutator actions that modify $\mathcal{V}_G$ of I . Let π_I be the trace whose actions are in the log. The log contains for each thread $\mathbf{t}$ a sequence of pairs of matching call and return actions, and the mutator actions corresponding to that particular method execution between them. Some mutator actions are marked as commit actions of the method executions they belong to. The call and return actions in addition to the commit actions are visible to I/O-refinement checking procedure. By contrast, view-refinement checking procedure deals with all the mutator actions logged during the execution.

Replaying Actions from The Log

Recall that the communication between the test program and the checker thread is achieved using a log. Computing the value of view^I for view-refinement checking requires an atomic snapshot of the component state. Because the refinement check is performed independent of the test program, the checker is unable to access any intermediate state of the component seen during testing. Note that no state information is recorded in the log but the information about the actions performed. We use the log to solve the problem of taking this atomic snapshot and constructing the updated value of view^I . We keep a separate copy of the component, denoted I^* , which is under control of the checker. I^* is illustrated in Figure 5.2 by the component instance **Impl_{replay}**. The I^* starts from the same initial state as the initial state of I . As we read an action α_i from the log, we re-execute α_i on I^* . After that, the state s_{i+1} of I^* is the same as the instantaneous state s_{i+1} of I seen during testing and it is available to the checker for analysis. This state information is important for view-refinement checking.

In addition to the above, we construct a method-atomic interleaving by arranging these call-commit-return action triples in the order of their commit actions. To accomplish this, we simply execute the specification one method call at a time in the order given by the witness interleaving. Recall that the witness interleaving of specification methods is obtained from the order of commit actions of the corresponding method executions in π_I . This execution

of specification obtained in this way generates the trace π_S . Recall that since S is required to be atomic and deterministic, this ordered sequence of call-return pairs corresponds to a unique run, i.e., uniquely determines the sequence of states that the specification goes through.

The procedures that (1) re-execute actions of π_I on I^* and (2) executes witness-interleaved specification methods on S is called *the replaying mechanism*. Replaying mechanism decouples the program being checked from the verification process.

Replaying mechanism is crucial for view-refinement checking. The component implementation and its specification are not actually modified to perform view-refinement checking. Instead, the checking thread separately (possibly off-line) constructs the value of view^I by replaying the logged trace on I^* . The value of view^S is computed by running S driven by the witness interleaving. The variables view^I and view^S are initialized to the same value. Each mutator method updates view^I once, atomically with its commit action. view^S is updated once, atomically between the call and return of each mutator method of S . Observer methods do not modify the view variables. The verification for view-refinement consists of checking that the same sequence of updates are performed to view^I and view^S .

The Iterative Checking Procedure

Let π_I be the trace of a serialized execution of I , which has been logged. The checking procedure verifies that whether π_I conforms to any execution of S with the trace π_S . We perform the check per thread basis, comparing $\pi_I|_t$ and $\pi_S|_t$ for each thread $t \in \text{ThreadId}$ for (1) the method return values, if exist, and (2) views computed using abstraction function. This comparison is done at each commit point seen along π_I .

The checking procedure reads from log one record for an action α_i at a time and runs according to the type of α_i as follows:

- If α_i is a mutator action, the replaying mechanism re-executes α_i on I^* . This allows the checker to observe the state of I just after α_i is executed during testing if α_i is marked as commit action as explained below.
- If α_i is a call action for a method execution ξ , we create a method signature $\text{Sign}_\mu(\xi)$ whose method name, calling thread id and arguments are derived from α_i .

- If α_i is a return action for a method execution ξ , we put the return value included in α_i into the corresponding method signature $Sign_\mu(\xi)$.
- If α_i is marked as commit action for a method execution ξ , we check the corresponding method execution ξ with signature $Sign_\mu(\xi)$ against I/O-refinement and view-refinement criteria. As commit action may not be the return action, we require that the return value to be available in $Sign_\mu(\xi)$. Therefore the checker may need to look forward¹ in π_I to read the return value from the return action of ξ . After acquiring the return value for ξ , the replaying mechanism runs the corresponding specification method μ on S , using the return value in $Sign_\mu(\xi)$ if necessary. Then $view^I$ and $view^S$ are computed for the current states of I^* and S , respectively. If at that point, (1) it is not possible to execute the specification while conforming to the return action as specified in $Sign_\mu(\xi)$ or (2) $view^I$ and $view^S$ does not match at that point, the refinement check is said to fail.

5.4 Verifying Observer Methods

The checking procedure for observer methods runs differently. This is because deciding on which action of an observer method execution to be marked as the commit action is difficult for two reasons:

1. While the commit action of a mutator method is typically associated with an atomic write to a shared variable, the commit action of an observer method is typically associated with a shared variable read. A subset of shared variable reads performed by an observer method determine its return value, as is the case, for example, with a `LookUp` call that returns `true`. However, it is not known before the read is performed whether its outcome will determine the return value. Therefore, precisely marking the commit action of an observer method requires that almost all reads performed by it be logged. This would result in large overhead and significant impact on the concurrency among the operations.

¹Our tool VYRD accomplished this by using two separate threads, one reading the log and fills in the method signatures, and another that runs checking procedure. The latter thread may wait for the former one to read the return action of a method whose commit action is to be handled.

2. The commit point of an observer method may depend on other concurrently executing methods. This dependency is hard for the developer to specify. As an example, consider a concurrent execution of a `LookUp(x)` and a `Delete(x)` methods of a Multiset instance that contains `x` in `A[i]`.

<code>LookUp(x)</code>	<code>Call</code>	...	...	...	<code>if(A[i].elt == x)</code>	<code>Ret</code>	...
<code>Delete(x)</code>	...	<code>Call</code>	<code>A[i].elt = -1;...rel(A[i])</code>	...	...	...	<code>Ret</code>

Both methods start at the same time, and after `Delete(x)` removes `x` from the multiset, `LookUp(x)` checks the slot from which `x` is removed and returns `false` at the end. If we record all the reads `LookUp(x)` performs and choose its commit point before `Delete(x)` removes `x` (`rel(A[i])` is its commit action), this causes a refinement violation to be thrown as the corresponding specification method returns `true`. However, if we choose the commit point after `Delete(x)` removes `x`, the specification's `LookUp(x)` method returns `false`.

Instead, we obviate the need for annotating and logging the execution of observer methods as follows. We only record the call and return actions of observer methods into the log and perform a different check.

Actions in the log	α_0^{cmt}	...	<i>Call</i>	...	α_1^{cmt}	...	α_2^{cmt}	...	α_{n-1}^{cmt}	...	α_n^{cmt}	...	<i>Ret</i>				
Specification state	$\tilde{s}_0$				$\tilde{s}_1$				$\tilde{s}_2$				$\tilde{s}_{n-1}$			$\tilde{s}_n$	

Figure 5.3: Checking refinement for observer methods.

Figure 5.3 shows one example execution of an observer method. For such an execution ξ of an observer method μ , let `Call` and `Ret` be the call and return action entries in the log, and $\alpha_1^{cmt}, \alpha_2^{cmt}, \alpha_3^{cmt}, \dots, \alpha_n^{cmt}$ the commit actions of mutator methods that lie between `Call` and `Ret` in the log. Let α_n^{cmt} be the last commit action before `Call`. We denote by $\tilde{s}_i$ the specification state obtained after atomically executing the method associated with α_i^{cmt} for all $0 \leq i \leq n$. Let ρ be the return value ρ in ξ of the method μ . For each $\tilde{s}_i$ ($0 \leq i \leq n$), we make an *observation* by atomically executing the specification version of μ on $\tilde{s}_i$. If the return value ρ is consistent with the observation (what the specification allows as a valid return value for μ) at any of the states $\tilde{S}_\mu = \tilde{s}_0, \tilde{s}_1, \dots, \tilde{s}_n$, we declare ξ correct. This amounts to allowing the real commit action of ξ to be anywhere between (and

including) *Call* and *Ret*. Otherwise, a refinement violation is signaled since a return value of ρ is not consistent with any choice of a commit action for ξ . Observe that this check does not result in a combinatorial blow-up in the number of linearizations considered, since the witness interleaving provides a unique ordering of commit points of mutator methods and the choice of a commit point for an observer method does not depend on other observers. Because observer methods do not modify the data structure state, the choice of the commit action for an observer method does not influence the checks performed for other methods.

5.5 The Atomized Implementation as the Specifications

Our technique can be used in conjunction with specifications in different formats, ranging from full-blown programming languages like Java to high-level description languages like TLA [6]. If a separate specification is not provided by the developer, we can also use an “atomized version” of the same implementation code as the specification. The idea of using an atomized version as a specification for refinement verification was introduced earlier by Tasiran and Qadeer [39], and Flanagan [17] independently. In an atomized version of I , denoted by $\llbracket I \rrbracket$, the component is forced to have only method-atomic executions. This can be accomplished by, for instance, defining a global lock for each instance of the component and requiring that methods must acquire the lock before they can start execution, hold this lock during execution, and must release it upon termination. For a component written in Java, e.g., $\llbracket I \rrbracket$ can also be achieved by defining all the public methods of I as `synchronized`. Thus for $\llbracket I \rrbracket$, at any time, there is only one method of it executing, guaranteeing sequential execution.

We require that specifications also allow exceptional termination due to concurrency by ending without any change in the abstract state (the view) of the component. The atomized version’s methods are also modified so that, in addition to the original set of arguments, they take the return value as an additional argument and compute the new specification state, in a fashion similar to Figure 3.1. Figure 5.4 shows the instrumented version of Multiset code to be used as the specification.

An alternative way to view this approach is to decompose the refinement verification task into verifying that

1. the implementation refines its atomized version, which is the focus of this thesis, and
2. the atomized version, a sequential program, refines a higher-level specification.

The latter problem can then be addressed using methods developed for verifying sequential programs [45, 44] that checks sequential execution of methods of the component obeys the sequential specification given as method pre- and post-conditions. Throughout this thesis we assume that the component, in a sequential environment, exhibits correct behaviors with respect to its sequential specification and concentrate on the first problem of detecting errors due to concurrency.

5.6 Checking Invariants

In addition to I/O-refinement and view-refinement, our technique allows invariants for the internal data structure of the component to be checked. The programmer can require that an invariant is satisfied at each commit point seen along an execution. Thus a check is performed on the state of the component at the commit point.

The programmer should state carefully the property to be checked at the commit point because the data structure may be in an inconsistent state due to other threads executing concurrently. This also a concern for the abstraction function computation. A property satisfied at commit points helps the developer to write the abstraction function more conveniently by relying on the property. For the Multiset example the following invariant is checked at each commit point along a trace:

For each slot `A[i]` ($0 \leq i \leq n$), if `A[i].valid == true` then `A[i].elt != null`.

The invariant guarantees that a slot with a `null elt` field can not be seen by any method as an allocated slot. The abstraction function of Multiset given in Figure 4.4 relies on this guarantee to skip checking that `elt` field of a slot with a `valid` field set to `true` is not null.

As another example, consider two storage components, a cache and a persistence manager, that manage $(addr, data)$ pairs in memory or in a persistent storage, respectively. *addr* is an abstract pointer, which can be interpreted by both components, to the area in which

$data$ is stored. For a $(addr, data)$ pair that persistence manager has there may or may not be an entry in the cache. If it exists, the clients of this storage system first tries to read from the cached version of $data$. A cache item $(addr, data)$ can be clean or dirty depending on whether $data$ in it is the same as the version in the persistent storage. There is the following invariants for this system:

If there is a clean cache entry for $(addr, data_i)$, there is a pair $(addr, data_j)$ in the persistent storage where $data_i = data_j$.

The developer can desire for the implementation of the cache and the persistence manager to satisfy this invariant at commit points of methods. The abstraction function, which returns the list of $(addr, data)$ pairs managed by this storage system, can be written by considering the property above so that if a clean cache entry for $addr$ exists, the cache entry $(addr, data)$ is used for abstraction skipping the version of $data$ in the persistent storage.

In fact the bug we found in an earlier version of the Boxwood system, reported in Section 7.3.2, causes violation of a similar invariant. After the buggy scenario happens, there is a clean cache item and a corresponding persistent pair in Chunk Manager with different data.

5.7 The Refinement Checking Algorithm

We now elaborate on the details of our checking method. Figure 5.5 gives the high-level description of the algorithm that analyzes the trace for refinement checking.

In the actual implementation of the algorithm given below, there are two threads running on behalf of the checking algorithm. The first thread reads the log and constructs a list of actions in π_I . It also creates method signatures when it sees a call action and fills in the relevant fields of it as it sees commit and return actions. The second thread, in parallel, reads one action at a time from the action list constructed by the first thread and runs replaying and refinement checking procedures. For simplicity, the algorithm below merges the tasks performed by these threads, also skipping how π_I is obtained from $\mathcal{L}$.

The main loop of the verifier runs over a given implementation trace π_I . The following is the explanation of the algorithm line by line:

VERIFIER-LOOP:

Line 3: The next action α_i is read from π_I . The remainder of the algorithm runs depending on the type and contents of α_i .

Line 4: α_i is re-executed on I^* by the replaying mechanism. After that, the current state s_{i+1} of I^* is equivalent to the state $r_I[i + 1]$ of I after just after α_i is executed during testing.

Line 5-7: If α_i is a call action for some method execution ξ , a new method signature $Sign_\mu$ is created and associated with ξ . The fields of $Sign_\mu$ are filled in as other actions like commit and return actions are detected.

Line 8-11: If μ is an observer method, an observation is made for the call point of ξ by running the atomic version of μ on S as explained in Section 5.4.

Line 12-15: If α_i is a return action for some method execution ξ , the return value ρ contained in the return action is put in into the signature $Sign_\mu$.

Line 16-21: If μ is an observer method, an observation is made for the return point of ξ by running the atomic version of μ on S . At this point the I/O-refinement check for μ is done by checking whether at least one of the observations performed between the call and return points of ξ matches to the return value ρ in $Sign_\mu$.

Line 22-24: If α_i is a commit action, the DOCHECKREFINEMENT routine is called.

DOREFINEMENTCHECK:

Line 1-3: First the corresponding method signature $Sign_\mu$ is selected and it is checked if $Sign_\mu$ has the return value. For most cases when the commit and return actions for a method signature are different, ρ is not available at this point. Thus the procedure LOOKFORWARDFORRETURNACTION is called. The procedure reads the subsequent actions in π_I until it reaches the return action of ξ from which it obtains the return value ρ of ξ . Then ρ field of $Sign_\mu$ is filled ρ of ξ and the algorithm can continue.

Line 4: The corresponding method in S is called with the same arguments as in $Sign_\mu$. After this point, the algorithm has return values R_{impl} and R_{spec} from I and S method executions, respectively.

Line 5-8: For all observer method executions not completed yet, an observation is made by calling corresponding observer method in S .

Line 9-12: If μ is a mutator method, the I/O-refinement check for ξ consists of comparing the return values R_{impl} and R_{spec} .

Line 13-17: The view-refinement check consists of computing the view variables $view^I$ and $view^S$ using the abstraction functions and comparing equivalence of these. At this point $\Theta_I(I^*)$ and $\Theta_I(S)$ can be considered as $\Theta_I(r_I[i+1])$ and $\Theta_I(r_S[i+1])$, respectively.

Line 18-19: If any invariant is given by the developer, it is also checked whether the state of I^* satisfies the invariant.

5.8 Commit Blocks

The computation of $view^I$ given in Figure 4.4 for the Multiset example is a bit oversimplified. The abstraction function Θ assumes that the current program state is a clean state, no other method of Multiset is in progress. However, it can be called at a commit point when there are other methods manipulating the data structure. Because Θ must be run atomically by the checker at the commit point, it skips the locking discipline of the component so that all the changes by the other threads are visible to Θ .

Consider the scenario where a thread t_1 that is executing an `InsertPair(x1, y1)` operation completes line 11 (`A[i].valid = true`). Then, a context switch occurs to thread t_2 which is executing `InsertPair(x2, y2)`. Suppose that t_2 executes its commit action (line 13). At this point, the $view^I$ computation will include x_1 into $view^I$ but not y_1 , since its valid bit is not yet set. Figure 5.7 shows the `INSERTPAIR` method. Since lines 11 and 12 of the method are protected by locks, it is not possible for another thread to see this “dirty” state where x_1 is in the multiset but y_1 is not.

To handle such cases, intuitively, the computation of view^I needs to “roll back” some shared variable modifications that uncommitted operations have performed. However, undoing some operations may be expensive or impossible for some implementations, such as ones used in file systems or networking applications. To ease the task of defining and computing view^I , we introduce the concept of *commit blocks*. Lines 9 and 13 of the `InsertPair` method in Figure 5.7 indicate the beginning and the end of the commit block for `InsertPair`.

The intent of the commit block is that (i) it can be easily verified to be atomic, and (ii) the task of computing view^I is simplified by assuming that only the committing thread is in the process of executing a commit block. In an actual execution π , (ii) does not hold, but by using (i), the execution can easily be converted to an equivalent execution $\tilde{\pi}$ according to reduction as defined in [20] where (ii) holds. Then, view can be written under the assumption that, at any given commit point, only one thread is in the process of executing its commit block. The commit block must be selected such that its atomicity can either be verified easily, statically, using a reduction argument, or be checked at runtime using a tool based on reduction, such as Atomizer [19]. Conceptually, our refinement checking algorithm first computes $\tilde{\pi}$ from the actual execution τ described in the log, and then performs refinement checking on $\tilde{\pi}$.

In fact, relevant portions of $\tilde{\pi}$ can be constructed incrementally. Because a commit block is atomic, the actions in it does not conflict with any other action until the block ends [20]. Thus the replaying mechanism for I^* can execute all the actions in the commit block atomically at any point between the first and the last actions of the block. We execute the entire block when we reach the commit action of the associated method execution, which is also part of the commit block. This allows us to construct $\tilde{\pi}$ on-the-fly during the refinement check.

The choice of the commit block must strike a compromise between simplifying the computation for view^I and verifying that the commit block is atomic. Marking a commit block too large, for instance, making the entire method as the commit block may make it impossible or too expensive to statically verify that the block is atomic. Marking too small a block as the commit block, for instance only the commit action, as the commit block does not simplify the computation of view^I . In this case the developer must consider the modifications on the state by the other threads. Since these modifications may give an invalid

abstract state, the abstraction function may have to undo some variable updates that are not yet visible to the outside.

The granularity of checking done using view variables is intermediate between two extremes. At one extreme, one can require a state correspondence only at clean states of the implementation as in [26, 17]. A clean state is a state in which no thread is in the middle of execution. But most industrial-scale concurrent data structures are built to be used by large numbers of threads continuously and during any realistic execution, clean points are very rare. Checking only at these points might cause errors to be overwritten or to be discovered too late. At the other extreme, asking the programmer to write a refinement map that relates implementation and specification states after each action is impractical. Our choice of view variables that are updated only with commit actions strikes a good compromise: A check is performed for each method execution, and the refinement map is easier to write, particularly if commit blocks are used as described above.

We explained in this chapter the strategy we follow in checking that an execution of the implementation conforms to an execution of the specification. We implemented the techniques proposed in this chapter in a runtime verification tool called VYRD. The following chapter introduces our tool and describes how it implements different parts of our verification method.

<pre> synchronized InsertPair (x, y, returnValue) 1 if(returnValue == failure) { 2 return failure; 3 } 4 i = FindSlot(x); 5 if (i == -1) 6 return failure; 7 j = FindSlot(y); 8 if (j == -1) { 9 A[i].elt = null; 10 return failure; 11 } 12 synchronized (A[i]) { 13 synchronized (A[j]) { 14 A[i].valid = true; 15 A[j].valid = true; 16 } 17 } 18 return success; </pre>	<pre> synchronized Insert(x, returnValue) 1 if(returnValue == failure) { 2 return failure; 3 } 4 i = FindSlot(x); 5 if (i == -1) 6 return failure; 7 A[i].valid = true; 8 return success; synchronized Delete(x, returnValue) 1 if(returnValue == failure) { 2 return failure; 3 } 4 for (i=0; i<n; i++) 5 synchronized(A[i]) { 6 if ((A[i].elt == x) && (A[i].valid)) { 7 A[i].valid = false; 8 A[i].elt = null; 9 return success; 10 } 11 } 12 return failure; </pre>
--	--

Figure 5.4: An example Multiset specification derived from the implementation code.

```

VERIFIER-LOOP( $\pi_I, I^*, S$ )
1  while  $\pi_I$  has more actions
2      do
3           $\alpha_i \leftarrow \text{GET-NEXT-ACTION}(\pi_I)$ 
4           $\text{REPLAY-ATOMIC-ACTION}(\alpha_i, I^*)$ 
5          if  $\alpha_i$  is a Call
6              then
7                   $\text{Sign}_\mu\langle t, \mu, \chi, \rho \rangle \leftarrow \text{CREATE-METHOD-SIGNATURE}(\alpha_i)$ 
8                  if  $\mu$  is observer
9                      then
10                          $O_{impl} \leftarrow \text{REPLAY-ATOMIC-METHOD}(\mu, S)$ 
11                          $\text{ADD-OBSERVATION}(O_{impl}, \text{Sign}_\mu)$ 
12             elseif  $\alpha_i$  is a Ret
13                 then
14                      $\text{Sign}_\mu\langle t, \mu, \chi, \rho \rangle \leftarrow \text{GET-METHOD-SIGNATURE}(\alpha_i)$ 
15                      $\text{Sign}_\mu.\rho \leftarrow \text{GETRETURNVALUE}(\alpha_i)$ 
16                      $\triangleright$  Check I/O-refinement for observer methods
17                     if  $\mu$  is observer
18                         then
19                              $O_{impl} \leftarrow \text{REPLAY-ATOMIC-METHOD}(\mu, S)$ 
20                              $\text{ADD-OBSERVATION}(O_{impl}, \text{Sign}_\mu)$ 
21                              $\text{ASSERT}(\exists O_{impl} \in \mathcal{O}. [O_{impl} = R_{spec}])$ 
22             elseif  $\alpha_i$  is a  $\alpha^\gamma$ 
23                 then
24                      $\text{DOCHECKREFINEMENT}(\pi_I, I^*, S, \alpha_i)$ 

```

Figure 5.5: Runtime refinement checking algorithm

```

DOCHECKREFINEMENT( $\pi_I, I^*, S, \alpha_i$ )
1   $Sign_\mu \langle t, \mu, \chi, \rho \rangle \leftarrow \text{GET-METHOD-SIGNATURE}(\alpha_i)$ 
2   $Sign_\mu \cdot \rho \leftarrow \text{LOOKFORWARDFORRETURNACTION}(Sign_\mu, \pi_I)$ 
3   $R_{impl} \leftarrow Sign_\mu \cdot \rho$ 
4   $R_{spec} \leftarrow \text{REPLAY-ATOMIC-METHOD}(\mu, S, \chi, R_{impl})$ 
5  for all observer methods with signature  $Sign'_\mu \langle t', \mu', \chi', \rho' \rangle$  not completed yet
6      do
7           $O_{impl} \leftarrow \text{REPLAY-ATOMIC-METHOD}(\mu', S)$ 
8           $\text{ADD-OBSERVATION}(O_{impl}, Sign'_\mu)$ 
9   $\triangleright$  Check I/O-refinement for mutator methods
10 if  $\mu$  is mutator
11     then
12          $\text{ASSERT } (R_{impl} = R_{spec})$ 
13  $\triangleright$  Check view-refinement
14  $\triangleright$  Apply abstraction function to get view of  $I$  and  $S$ 
15  $\text{view}^I \leftarrow \Theta_I(I^*)$ 
16  $\text{view}^S \leftarrow \Theta_S(S)$ 
17  $\text{ASSERT } (\text{view}^I \equiv \text{view}^S)$ 
18  $\triangleright$  Check any invariant  $\tilde{\Gamma}$  for the internal data structure
19  $\text{ASSERT } (\tilde{\Gamma})$ 

```

Figure 5.6: The procedure that performs checks at each commit point

```
InsertPair(x,y)
1  i = FindSlot(x);
2  if (i == -1)
3    return failure;
4  j = FindSlot(y);
5  if (j == -1) {
6    A[i].elt = null;
7    return failure;
8  }
   // begin commit block
9  synchronized (A[i]) {
10   synchronized (A[j]) {
11     A[i].valid = true;
12     A[j].valid = true;
13   } // end commit block
14 }
15 return success;
```

Figure 5.7: The implementation of `InsertPair` with the commit block marked.

Chapter 6

VYRD: THE RUNTIME CHECKING TOOL

We implemented our runtime refinement checking techniques as a verification tool called **VYRD** (**VerifYing** concurrent programs by **R**untime **R**efinement-violation **D**etection) [7]. VYRD provides components for instrumenting implementation code for logging, and requires the developers to provide data structure specific instrumentation as will be explained in this chapter.

We used the Boxwood project to drive our development of the VYRD tool. VYRD is very effective in detecting concurrency errors. Using it, we were able to detect a subtle error in a cache module in Boxwood, as well as known concurrency bugs in `java.util.StringBuffer` and `java.util.Vector`. Chapter 7 presents our experience with these programs and report the result in detail.

This chapter describes the main features of VYRD as a reusable refinement checking tool. Additionally, we present the design issues that we faced while verifying industrial-scale systems, such as minimizing performance and concurrency impact on the test program, and solutions to them implemented in VYRD.

6.1 The Testing Strategy

To produce traces for refinement checking, we use test programs. Our test programs run highly concurrent test cases designed for examining concurrency behavior of different sets of methods in the component. Each test case aims to trigger a particular concurrency error in a target component.

A randomized test program for the Multiset example is shown in Figure 6.1. The test case works on a single Multiset instance. It first creates *numthreads* threads each of which runs **TESTER-THREAD** procedure. **TESTER-THREAD** calls *numCalls* methods of the Multiset instance. At each turn it chooses the method to be called randomly (by using the randomly selected integer *m*) and selects its parameters randomly from a pool of values. A test case

should cause methods to affect each other's execution or make contending operations on a common subset of variables. One approach to exploit this behavior is giving correlated input arguments to methods so that they operate on a small region of the data structure. This increases the possibility of methods' revealing concurrency errors. To accomplish this, each test program first generates a random pool of values to be shared by all threads as arguments for method calls. For example the test case for Multiset has a pool of integers from which arguments are chosen by `CHOOSEARGUMENTFROMPOOL`. The pool can be reduced gradually over time to focus more concurrent method calls on a smaller region of the data structure. In implementations with internal mechanisms performed by internal threads, the internal threads are either triggered automatically by the methods called during testing, or by the test program itself at the beginning of testing.

6.2 Logging Issues

During testing, the component writes all interesting actions to the log. To accomplish this, the implementation code is instrumented to save actions performed and related data to the log at runtime. We provide helper classes in a reusable library of VYRD that relieves the programmers from considerable programming effort for logging and restoring the implementation trace. In practice, the log can be in memory entirely, or it can be a file whose tail is kept in memory for faster access. The logging mechanism of VYRD uses the binary object serialization mechanism, which is supported by both the .Net and Java platforms. The log items are instantiated as record objects that contain information on actions performed. Then the log is written into a shared, sequentially accessed binary file. The log records are restored by the verification engine to obtain the execution trace. Serialization mechanism is relevant for our purposes since it relieves us from details of dealing with persisting objects, and fast enough to save and retrieve log entries to/from persistent storage.

In order to reduce the impact of logging on the concurrency characteristics of threads, each thread t produces a separate log $\mathcal{L}_t$ that contains its actions. The log entries are totally ordered by a single global counter that assigns a unique identifier to each entry. All threads synchronize to access this counter having separate logs (as in the FSE group's "multiplexer"). Before writing an entry to its own log, each thread is forced to atomically

increase the counter and get an identifier to the entry. The logs $\mathcal{L}_t$ for all $t \in Tid$ are merged into a single totally-ordered log $\mathcal{L}$ to be analyzed by the refinement checker. This approach provides benefits only if different counters are used for different domains.

Our technique also lets the developer to choose existing logging mechanism of the system being analyzed. For example in the Boxwood system the atomic operations performed are written to a sequential log, which is managed by a separate module in Boxwood. This log is for rolling back the previous operations upon system failures. The logging mechanism of Boxwood allowed us, in early phases of developing VYRD, to interpret log entries using a sequential memory model. However, in the latter phases, the common classes of VYRD was used to establish instrumentation as explained in Section 7.2.4.

6.3 Analysis at Different Granularities

It is possible to do refinement checking at different levels of granularity. The granularity of the analysis affects which type of operations encountered along a method body are treated as “actions”, and how they are logged and replayed. The developer has the option to do analysis with fine- or coarse-grained actions.

An example of fine-grained logging is to record all writes to shared variables, e.g., assignments to an integer-typed variable, inserts to a vector/hashtable, or locking/unlocking using a semaphore. In this case no knowledge of the data structure is necessary to re-construct data structure state using the log in this case. The replaying mechanism is simple and supported by the underlying language. The logging and replaying mechanisms for the actions can be reused for different components. The disadvantage of fine-grained actions is that logging an execution takes much space and replaying them may take much computation time.

Coarse-grained logging is data-structure specific. In this case a group of fine-grained operations are logged as an entry that corresponds to a single action. Such an action typically is a lower-level task than a public method, e.g., increasing or compacting the size of an array-based vector, re-distributing data between two tree nodes to balance a subtree, but higher-level than a single shared variable update. Since such tasks are data structure-specific, “replay” methods that re-construct data structure state from the log must be provided by the programmer for each such kind of log entry. In addition, the

logging a group of operations a single action must be performed by the developer, e.g., by defining new log entry classes and instrumenting the source code as needed. Analysis in this level reduces logging contention and overhead as the number of actions performed during a method execution is very low compared to the number of actions considered in fine-grained analysis.

The property that an action must be executed atomically must be ensured by the developer. This is also true for fine-grained analysis: Reads/writes to variables of a primitive type or operations on a commonly-used data structure like a vector or a semaphore must be guaranteed to run atomically by the underlying programming language. However, the course-grained operations chosen as actions may not be guaranteed be atomic by the underlying system. If the developer wants to do the analysis in spite of this, she must make these operations execute atomically, for example, by globally synchronizing the group operations that form a single action with the other threads, e.g, using mutexes.

6.4 *Programmer's Responsibilities*

VYRD provides a common library of features. For the component to be properly checked, the developer must give extra classes that will allow VYRD to perform data structure specific tasks. We leave those to the programmer since we want our checking tool to be as independent as possible from the details of the data structure as the developer can use VYRD library for data-structure independent instrumentation of the code. This section introduces the functionality provided by the developer.

6.4.1 *Programmer Defined View Construction*

Defining a view for a concurrently accessed data structure requires knowledge of internals of the implementation and the relation between the method specifications and the internal structure. This stems from the fact that the methods provided by the components reflects an abstract structure (view) to the clients of the component. Whichever state the implementation is in due to multiple concurrent access, the results of the concurrent executions of the methods must guarantee a consistent modification or observation of the view. Therefore great care must be taken to view definition of the component and it makes the intervention of the programmer inevitable to write it.

Roughly speaking, the view definition $\langle \tilde{\mathcal{S}}, \Theta \rangle$ of a component must extract the abstract data structure contents by abstracting away the rest of the program state. How this abstraction is performed is specific to the application: The state space $\tilde{\mathcal{S}}$ is specific to the component and Θ is specific to the implementation. The developer needs to describe how **view** is to be computed from program state. Currently in VYRD, **view** is defined by writing a method called **AbstractionFunction** that implements Θ . **AbstractionFunction** computes the abstract state using the logged trace of I . It returns another data structure instance pointed by view^I . This data structure is implemented as a separate class that has methods for equivalence checking for its instances. The specification must also have a function that returns a data structure instance pointed by view^S . If the atomized version of the implementation is used as explained in Section 5.5, it is preferable to use the same **AbstractionFunction** for both the implementation and the specification. Note that for the view-refinement check to run properly, the data structures pointed by view^I and view^S must be either of the same type or be in a comparable format.

AbstractionFunction for the Multiset example is given in Figure 4.4. In many examples, including Multiset and Boxwood, **view** can be written using shared state defined by $\mathcal{V}_G$ only. First, the subset $\text{supp}(\text{view})$ of $\mathcal{V}_G$ in the support of view, the variables that reflect their value directly in the abstract state of the component, are determined. Given correspondence between the states of I and the abstract states, the set $\text{supp}(\text{view}^I)$ can be computed by a simple static analysis of the code of the implementation. In our current approach, the developer is responsible for determining $\text{supp}(\text{view})$ and deriving the view computation from an implementation state. $\text{supp}(\text{view}^I)$ for Multiset contains the variables $A[i].\text{elt}$ and $A[i].\text{valid}$ for all $0 \leq i \leq n$ as n is the size of the array A . If S is not the atomized version of I but supplied separately, the same procedure above is performed for it by the developer. Once we understood a data structure implementation, even for the industrial examples we studied, we found that it only took a few days of effort to determine the characteristics of the abstract state space and to write **AbstractionFunction**.

The developer is also asked to provide a method that checks equivalence of the data structures pointed by view^I and view^S . This method is written in the class implementing the data structure representing the view. For example, **AbstractionFunction** of Multiset in Figure 4.4 returns a map of type `Map<int, int>` that holds the abstract contents of a

multiset, by mapping each unique integer to its count in the set. `Map<int, int>` implements the `equals` method that checks that two integer sets has the same contents in both. For the Boxwood system, classes were written that represent the abstract contents of `BLinkTree` and `Cache`. The classes have methods that check if the two instances represent the canonical versions of equivalent `BLinkTree` and `Cache` states. In addition, if the implementation is supposed to satisfy an invariant at each commit point, as described in Section 5.6, the developer supplies additional code that traverses the program state at the commit point and checks if the invariant is satisfied. This code is embedded in `AbstractionFunction`. The abstraction function in Figure 4.4 contains the check as an assertion for the invariant of Multiset (For each slot `A[i]` ($0 \leq i \leq n$), if `A[i].valid == true` then `A[i].elt != null`). If either `viewI` and `viewS` are detected to be inequivalent or an invariant violation happens, VYRD is informed and it gives a warning.

6.4.2 Incremental View Computation & Comparison

The size of the program state for an industrial-scale concurrent data structure can be huge, e.g., the entire contents of a hard disk for a file system. Re-traversing the entire program state at each commit point to compute the abstraction function takes a lot of time while changes are restricted in to a small region of the data structure. To avoid re-traversing the entire state, `viewI` and `viewS` must be computed incrementally. Typically, since a view variable represents data structure contents, its value has a certain structure. Then one can speak of portions of that structure that are modified between two commit points. For example if the view of a component is represented as a linked-list, an action that manipulates the implementation state updates only a subset of linked-list entries in the view. As the replaying mechanism re-executes each action α , we perform a dependency analysis of the portions of the view updated to the variables in $\text{supp}(\text{view}^I)$ modified by α . We then only re-compute and compare only the relevant parts of the view. A similar analysis is performed for S after an atomic method is executed at a commit point.

Incremental view computation is a data structure-specific procedure. VYRD does not has a reusable mechanism for incremental view computation but the developer manually inserts methods that makes the dependency analysis and partially updates the view. The way this is plugged into the checking algorithm makes it look like VYRD has this functionality.

For the Multiset example, an action that updates `elt` or `valid` field of an element $A[i]$ ($0 \leq i \leq N$) makes a small change in one element in the view of Multiset. Then the change on view^I can be determined by only computing the modification in the abstraction of $A[i]$ and applying it to the view. Figure 6.2 shows a simplified, iterative version of the abstraction function that partially updates view^I . It takes as input the relevant information in order to figure out which portion of the view should be updated: the data structure pointed by view^I , the slot i updated, a boolean indicating whether the `valid` or `elt` field is updated and a boolean indicating whether the field is set to some value or it is reset to `null` or `false`. `PartialAbstractionFunction` adds or removes the content of the slot $A[i]$ from/to the set M depending on the effect of the action on $A[i]$.

6.4.3 The Specification

Along with the component to be verified, a specification must be supplied. The current version of the tool accepts atomized version of the same implementation code as the specification. This is done by creating another wrapper class that will act as the specification. The wrapper is written so that some methods can nondeterministically terminate exceptionally and the whole bodies of the methods are fully-synchronized as explained in Section 5.5. This atomic version is invoked at commit points during the replaying mechanism to construct the specification trace π_S .

Figure 6.3 shows a wrapper class to be used as Multiset specification. It contains a Multiset instance `multiset`. The wrapper first ensures that the return value coming from the implementation trace does not indicate an exceptional termination., and then it propagates the call to `multiset`.

6.4.4 Code Instrumentation for Logging and Replaying

The developer must instrument the implementation code in order to log actions when they are performed at runtime. The instrumentation can be done in either source code or compiled code level. For the current implementation of our tool, source code level instrumentation is used. The instrumented code must save a call action of a method with its actual parameters, and the return action with the return value. The logging library class has methods that allow call and return actions to be logged with their data. For other actions,

the developer is asked to write classes that extend the generic log item class. A log item subclass is instantiated when an action α of its associated action type is executed. Then the logger class is used to add the log item instance to the end of the log for the thread executed α . The logger class allows actions to be marked as commit actions by giving a particular flag to the logger methods. Our logging mechanism also supports commit actions to be determined during verification, which is useful when commit points are annotated with conditions. For this purpose, the developer inserts some points “precommit” log entries. When such an entry is read by the checker thread later, the following action by the same thread that wrote the precommit entry is marked as the commit action. For example, if the condition for a method to commit is satisfied at a point at runtime but the developer can not figure out what action to be executed after that point, she can insert a precommit entry to the log, which will make the next action to be executed the commit action of the regarding method execution.

In contrary to the specification trace, which is replayed with method calls, the implementation trace is replayed by re-executing individual actions read from the log. Replaying mechanism for generic fine-grained actions (reads/writes from/to primitive typed variables, insertions/deletions to/from common container classes like vector or hashtable, etc.) are supported by the implementation language. The implementation code must be added methods that can re-execute individual coarse-grained actions and data structure specific fine-grained actions on a component instance.

This chapter introduced how our runtime verification technique for refinement is implemented in our tool VYRD. Although runtime checking has additional costs due to replaying and abstraction mechanisms, optimizations in VYRD makes the technique tolerable for large examples. The experimental results reported in the next chapter shows that our technique has a moderate computational cost and it can find intricate concurrency errors in small time. Most of the mechanisms like logging and replaying that requires manual instrumentation effort can be automated by having an underlying virtual machine like the Java virtual machine (JVM (Java Virtual Machine)) in the Java PathFinder [81] model checker. Implementing the refinement checking on such a virtual machine is one of the future directions of this thesis (see Section 8.3.1).

```

TEST-MULTISET(size : int, numThreads : int, numCalls : int)
1  multiset ← CREATE-MULTISET(size)
2  for i ← 0 to numThreads
3      do
4          threads[i] ← CREATE-THREAD(TESTER-THREAD, multiset, numCalls)
5  for i ← 0 to numThreads
6      do
7          START-THREAD(threads[i])

TESTER-THREAD(multiset : Multiset, numCalls : int)
1  for i ← 0 to numCalls
2      do
3          m ← CHOOSERANDOMNUMBER(1, 4)
4          x ← CHOOSEARGUMENTFROMPOOL()
5          if m == 1
6              then
7                  multiset.INSERT(x)
8          elseif m == 2
9              then
10             y ← CHOOSEARGUMENTFROMPOOL()
11             multiset.INSERTPAIR(x, y)
12          elseif m == 3
13              then
14             multiset.DELETE(x)
15          elseif m == 4
16              then
17             multiset.LOOKUP(x)

```

Figure 6.1: Example test program for Multiset.

```

Map<int,int> PartialAbstractionFunction
    (Map<int,int> M, int i, bool isValidField, bool isSet) {
    if(isValidField) {
        if(isSet) {
            // invariant ensures that (A[i].elt != null)
            int count = M.get(A[i].elt);
            M.put(A[i].elt, count + 1);
        } else {
            if(M.Contains(A[i].elt)) {
                int count = M.get(A[i].elt);
                M.put(A[i].elt, count - 1);
            }
        }
    } else {
        if(isSet) {
            if(A[i].valid) {
                int count = M.get(A[i].elt);
                M.put(A[i].elt, count + 1);
            }
        } else {
            if(M.Contains(A[i].elt)) {
                int count = M.get(A[i].elt);
                M.put(A[i].elt, count - 1);
            }
        }
    }
    return M;
}

```

Figure 6.2: Incremental version of the abstraction function for Multiset

<pre> class MultisetSpec Multiset multiset; Multiset(Multiset currentset) { 1 multiset = currentset.clone(); } synchronized InsertPair (x, y, returnValue) { 1 if(returnValue == failure) { 2 return failure; 3 } 4 return multiset.InsertPair(x, y); } </pre>	<pre> synchronized Insert(x, returnValue) { 1 if(returnValue == failure) { 2 return failure; 3 } 4 return multiset.Insert(x); } synchronized Delete(x, returnValue) { 1 if(returnValue == failure) { 2 return failure; 3 } 4 return multiset.Delete(x); } synchronized LookUp(x) { 1 multiset.LookUp(x); } </pre>
--	--

Figure 6.3: The wrapper class implemented as the Multiset specification.

Chapter 7

EXPERIENCE

We tested VYRD on several programs to test its efficacy in catching errors. This chapter presents our experiments with components ranging from simple benchmarks to complex data structures in industrial-scale systems, and discusses the performance and effectiveness of our verification method.

7.1 The Boxwood Project

The Boxwood Project [24] by Microsoft® Research provides scalable storage infrastructure for applications. Boxwood brings the idea of abstracting a low level storage layer from the applications that have different data storage/retrieval requirements. Boxwood is developed to see how an infrastructure that abstracts the details of a storage system, like addressing, replication, recovery and synchronization, and makes those abstractions available for higher level data intensive applications.

Boxwood is written in C# and consists of roughly 30K lines of code (LOC). A block diagram for Boxwood is given in Figure 7.1. A distributed and concurrent implementation of a b-link tree module (BLinkTree) is built on Boxwood’s data store abstraction. In this abstraction, each shared variable is a byte-array identified by a unique handle and is stored and managed by the Chunk Manager module. Shared variables have version numbers that are incremented after each write. The Chunk Manager module implements a distributed storage that manages $(handle, byte-array)$ pairs where *handle* is a globally-unique pointer to an abstract storage area that contains *byte-array*. It provides its clients methods to allocate, read/write and deallocate byte-arrays. Read operation of Chunk Manager fetches a byte-array from the globally accessed storage into the local memory of a thread, and write operation of Chunk Manager writes a byte-array in the local memory of a thread to the globally accessed storage. The Cache module improves the performance of BLinkTree and is intermediate between BLinkTree and Chunk Manager. The contents of a BLinkTree is

distributed over multiple machines communicating over a network. Each machine creates a local copy of the tree and can access the whole tree distributed over the network. The copy on a machine uses a “local” Cache that forms another layer between the tree and the Chunk Manager, and Chunk Manager abstracts the distributed nature of the persistent storage of the data. Cache allows the trees on a local machine to access data blocks faster. BLinkTree communicates with Cache only to read or write any data block and it is Cache’s responsibility to send dirty data blocks to Chunk Manager to make the changes persistent. Cache accomplishes this by flushing dirty data. This involves intricate interaction between the locking and the logging mechanisms. The atomic operations performed by BLinkTree module are written to a sequential log, whose consistency is managed by Boxwood system and used for rolling back the previous operations upon system failures.

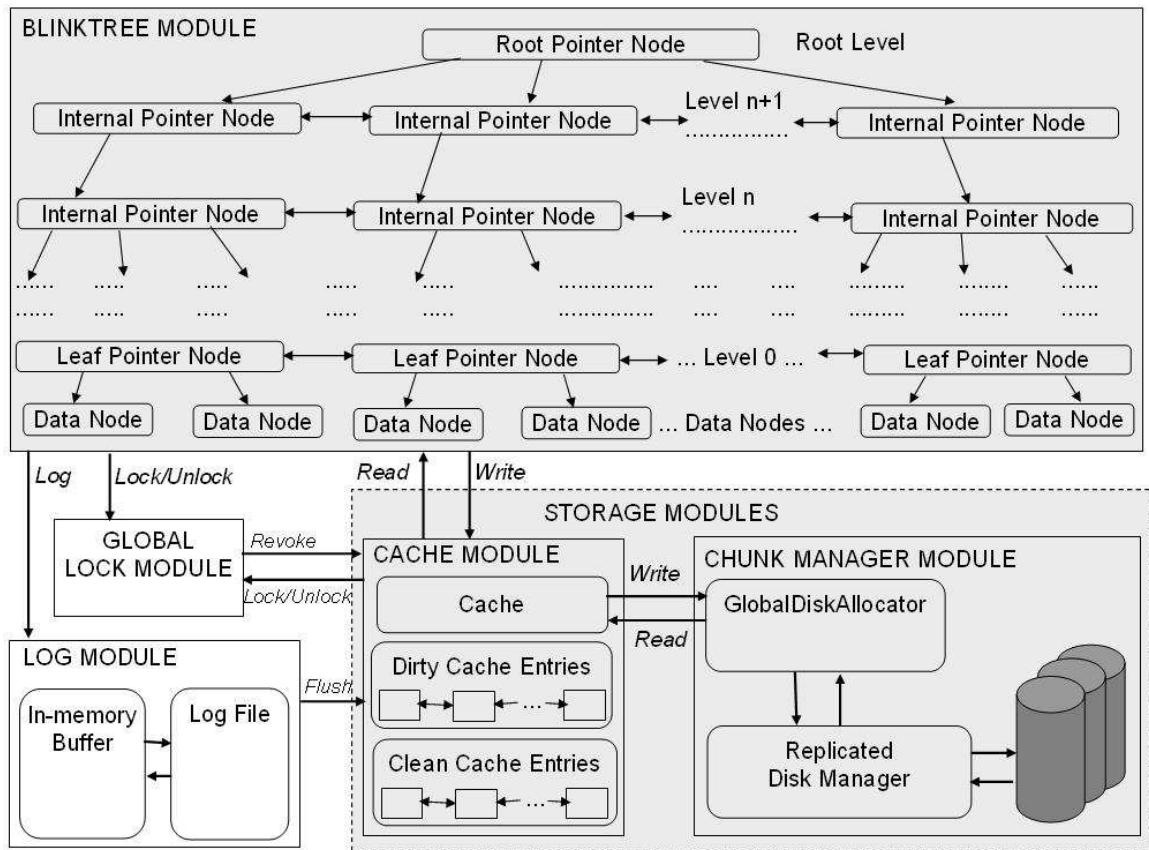


Figure 7.1: Boxwood block diagram

We verified the modules in Boxwood in two phases by considering its layered design and

dividing the verification task accordingly:

1. We assumed that the storage modules Cache and Chunk Manager are implemented correctly so that they satisfy abstract atomicity. Under this assumption, we ensured that all the methods of Cache and Chunk Manager run atomically by properly synchronizing them manually. Then we verified the refinement for the concurrent BLinkTree implementation with respect to its fully-synchronized version. Since Cache and Chunk Manager were guaranteed to be atomic, the calls to their methods were treated actions during fine-grained analysis of BLinkTree. In course-grained analysis, the b-link tree specific operations, like balancing the children of a root node of a subtree, were actions, keeping in mind that calls to Cache and Chunk Manager during these operations run atomically.
2. We then concentrated on verifying the Cache module, now assuming that Chunk Manager was implemented correctly so that it satisfies abstract atomicity. During the verification, method calls to Chunk Manager by Cache are treated actions. We treated Cache as a separate data structure that works in collaboration with Chunk Manager and has BLinkTree as its client.

7.2 The BLinkTree Module

7.2.1 The Concurrent Design of BLinkTree

The BLinkTree module in Boxwood is a highly concurrent implementation of the algorithms for a b-link tree data structure proposed by Sagiv [38]. A b-link tree provides its clients an interface that allows them to store/retrieve $(key, value)$ where key uniquely identify $value$. A b-link tree stores a set of $(key, value)$ pairs and provides three methods: $INSERT(key, value)$ inserts or overwrites the data to the tree associated with key , $DELETE(key)$ removes the data associated with key , and $LOOKUP(key)$ returns the data associated with key .

Figure 7.3 shows an example b-link tree. There are two types of nodes in the tree. Pointer nodes are used to traverse the tree towards the leaf pointer nodes. $(key, value)$ pairs are stored in data nodes pointed by a leaf pointer nodes. The non-data nodes form the indexing structure for the tree and are used for accessing the data nodes efficiently.

Concurrent execution of methods can change the indexing structure but the algorithms ensure that those modification on pointer nodes do not cause the data nodes at leaves to become unavailable from the root of the tree. Although the indexing structure is concurrently manipulated at different parts of it locally, all the data nodes are always kept visible and accessible to the methods. This is ensured by atomically updating the indexing structure and the leaf pointer nodes and the data nodes. The algorithms use an overtaking mechanism that allows all the effects of a mutator method to become visible to the other threads atomically. In this approach, the entire portion of the tree is copied and the method make the modifications on the nodes in that copy. Then the updated copy is swapped with the original part of the tree with the execution of a single action, as if the effect of the entire method is reflected to the data structure by that action. For example, splitting a pointer node $\mathcal{N}$ means making a copy of $\mathcal{N}$, transferring a subset of its children to the new node and updating parent of $\mathcal{N}$ to introduce the new node as its new child. This operation is performed by (1) making two copies of $\mathcal{N}$, and a copy of its parent node, (2) transferring the subset of $\mathcal{N}$'s children to the new node, (3) updating the copy of the parent node by introducing the new child, and finally (4) swapping the copy of the parent node with its original version in the tree. The last operation (4) is performed atomically and reflects all the updates through (1) to (3) atomically to the tree.

In addition to the public methods, there is also a private method, COMPRESS, that is run by a worker thread launched by the tree code itself. COMPRESS does compression to fix up empty or partially empty tree nodes. Compressing nodes involves removing the tree pointer nodes from the tree the number of whose children decreases below a threshold value (half of the capacity of a pointer node), and transferring its children to one of its neighbors. The internal compression thread works concurrently with data structure operations and re-arranges the tree structure without modifying the set of $(key, value)$ pairs. In addition to the data structure operations, we also checked that atomic updates that the compression thread performs to the program state do not modify the view.

The algorithms do not require the entire method body to be synchronized. Only some portions of the code is protected by holding locks on a few objects. Sagiv's algorithm for LOOKUP requires no locks to be held, while INSERT and DELETE require a single lock associated with a tree node to be held. In COMPRESS, up to three locks are held.

7.2.2 The Boxwood's Implementation of BLinkTree

In the Boxwood system, BLinkTree module implements the b-link tree introduced in the previous section. For Boxwood, each shared variable is a “byte-array”. Byte-arrays are stored and managed by Cache locally and Chunk Manager in a distributed manner. Chunk Manager maps abstract handles to byte-arrays, and allows the clients to allocate, deallocate, read and write byte-arrays to the abstract storage associating them with unique handles, whose contents are transparent to the clients. Since Chunk Manager distributes the entire abstract storage over a network, the contents of b-link tree represented by BLinkTree is naturally distributed in that network. BLinkTree is unaware of the distributed storage, though, as Chunk Manager abstracts the underlying distribution of data.

Every information used in BLinkTree is marshaled into a byte-array and stored in Chunk Manager. To increase performance, BLinkTree communicates Chunk Manager through the Cache module. Cache stores local copies of remote data for fast access. Since sequential consistency of the distributed data is guaranteed by Chunk Manager, which uses global lock module, BLinkTree uses locking schemes that synchronizes accesses of machine-local threads to the local copies of the variables.

7.2.3 Verifying the BLinkTree Module

This section elaborates how we verified BLinkTree module in detail. As the result of a long period of runtime checking, we found no errors in the original code for BLinkTree. However, some bugs, which are listed in Table 7.1, were inserted manually in order to collect experimental data for performance measurements.

Actions in BLinkTree Implementation

We verified the BLinkTree module for both coarse-grained and fine-grained actions. Our choice of coarse-grained actions are specific to BLinkTree, which are normally logged in a particular log format in the original Boxwood implementation. Some examples are adding a new level to the tree by inserting a new root node, merging two nodes into a new node and deleting the other one, and splitting a node into two nodes and inserting a pointer to one of these nodes. However the atomicity of operations that correspond to coarse-grained actions must also be ensured. Preferring a coarse-grained analysis over the fine-grained one

can allow us to catch a higher-level error without spending much computation time and space, while causing some intricate errors. Therefore, we use first coarse-grained actions, and then do a finer-grained analysis to make our method more accurate.

In fine-grained analysis, primitive operations on shared variables are the actions we take into consideration. As a shared variable for BLinkTree is represented by a byte-array stored in the abstract storage, method calls to Cache and Chunk Manager that manipulate $(handle, byte-array)$ pairs are treated actions. While using fine-grained ones, we synchronized the access to read/write methods of Cache and Chunk Manager during the concurrent execution of implementation to guarantee that they were really atomic. Checking Cache and Chunk Manager modules separately after finishing the verification of BLinkTree as explained in later sections verifies all the assumptions above.

Defining View for BLinkTree

Recall that $(key, data)$ pairs, which form the contents of a BLinkTree instance, are stored at data nodes pointed to by a leaf pointer node. The indexing structure is for only accessing the data nodes efficiently and it is kept invisible to the clients. Figure 7.3 shows the division of a b-link tree to the indexing structure formed by the pointer nodes and the leaf structure formed by the data nodes.

The view for BLinkTree component was defined to be the sorted list of all the $(key, data)$ pairs stored in the tree, along with their version numbers. The indexing structure is abstracted in the computation of $view^I$. The list of $(key, data)$ pairs was computed by a left to right traversal of the leaf pointer nodes and adding to the list the contents of each data node they point to.

Incremental View Computation for BLinkTree

During verification of BLinkTree, we updated the view for the tree incrementally. An action can modify a pointer node above the leaf pointer nodes, a leaf pointer node that points to data nodes or one data node. In first case, we do not update the view. For the last two cases, an incremental, update is performed. Let p be the pointer node that is modified or the parent of a data node modified. Then the contribution of p to the view computation, the sequence $(key, data)$ pairs contained in the data nodes pointed by p , is removed from

the view and the part of the view extracted from p is computed and included in as regular.

Figure 7.4 shows a modification on the leaf pointer node Y . Then the contribution of the data nodes pointed by Y is re-computed and only part of view through B to C is updated. The incremental view computation procedure is important for components managing large data structures in it for preventing the checker to traverse the entire state for each commit point.

Selecting Commit Points

The BLinkTree mutator methods described in [38] follow a pattern in which the effect of each method is reflected in the data structure state by a single write to a particular data or leaf node variable, while remaining writes re-structure the tree. For INSERT (See Figure 7.2) and DELETE, we identify this single write and designate it as the commit action. In INSERT, there are four possible commit points for different executions of INSERT depending on what exit path is taken through the method code. These paths are distinguished by whether the key being inserted was already in the tree (line 15), whether the leaf node that is being inserted into needs to be split into two nodes (lines 39 and 46), and whether the leaf node is also the root node of the tree (line 52). Each of these cases is identified using a boolean expression in terms of method-local variables. The appropriate leaf or data node write operation is then marked as the commit action.

7.2.4 Instrumenting the BLinkTree Module

In order to perform data structure specific logging and checking, the implementation source code is instrumented using the common library of the VYRD tool. This section explains the instrumentation of the BLinkTree module by listing the classes written to support VYRD for data structure specific tasks.

BTreeVyrdFactory: This factory class is used to instantiate the objects needed by VYRD to perform data specific tasks.

BTreeState: This class is used in view-refinement to represent the abstract state of BLinkTree. During view-refinement checking, the states of the implementation and specification are abstracted using their abstraction functions to produce BTreeState instances.

BTreeState implements the method **CheckEquality**, which compares two BTreeState.

BTreeSpec: This class represents the specification of BLinkTree and extends BLinkTree. It contains atomized methods that correspond to the public methods in the BLinkTree implementation. Its methods decorate, as described in Section 5.5, their counterparts by being synchronized and taking return values as input parameters. The BTreeSpec class also contains an abstraction function that computes abstract state (a BTreeState instance) of a BTreeSpec instance to be used by view-refinement checking.

BTreeImpl: Represents the implementation of BLinkTree. BTreeImpl keeps a BLinkTree instance whose state is updated after each action read from the log during verification. It contains methods that execute an action on the BLinkTree instance. It also has an abstraction function that computes the abstract state (a BTreeState instance) of the instance.

XAction Subclasses: For each BLinkTree method that is exported to its clients, there is subclass of the generic XAction class. XAction represents a method execution which is treated as a “transaction” on the data structure. Examples to these for BLinkTree are XAInsert, which, XADelete, XACompress, XALookUp. For Cache, there are subclasses XAWrite, XARead, XAFlush and XAREvoke. and XACache. XAction subclasses are responsible for managing the information that is kept in a method signature as described in the sections for the refinement checking. A XAction subclass also runs its corresponding method in the specification instance.

LogItem Subclasses: For each XAction subclass referred to above there is a LogItem subclass that contains the necessary information for the action to be replayed. The classes that extend XAction are responsible for replaying the actions they are associated with on a BLinkTree instance. Examples to those from the BLinkTree example is LogItem. There are also actions that correspond to specific method calls to underlying layers, e.g. Chunk Manager or Cache, such as LogItemCacheRead, LogItemCacheWrite, LogItemAllocRead and LogItemAllocWrite.

7.2.5 Inserting Instrumentation Code into The Existing Classes

XAction actions: XAction actions include call, commit and return actions, which are minimum requirements for the I/O-refinement check. Call actions are logged just after the method is called along with the actual parameters. Return actions are logged wherever the method returns back to the caller by either executing a *return* statement or throwing an

exception. Commit actions are actually the mutator actions that are logged depending on predefined conditions. One commit action is logged for each mutator method.

Mutator actions: For each atomic action execution, one log record must be saved to the log, some of which indicate commit actions. Since most of the primitive actions are also other method calls, the instrumentation is performed in those methods to centralize the action logging. For example, one action type is writing a byte array to the chunk manager. The action is performed by calling the method `Allocator.Write()`. Instead of inserting code at each line where a call to `Allocator.Write()` is made, the body of “`Allocator.Write()`” is instrumented, so that for each call to `Allocator.Write()`, the logging is automatically performed. For `BLinkTree`, this type of instrumentation is done for `Write`, `Allocate`, `Deallocate` methods and its constructor. To mark the commit points, `Insert` code is instrumented with four code lines, `Delete` code is instrumented with one code line, and `Compress` code is instrumented with three code lines.

Obviously, the instrumentation for logging requires so little effort that the number of code lines is much less compared to the program code in the method bodies. Centralizing action logging in the method body that represent the action performance relieves from inserting the logging code for each point at which the action is run.

7.3 The Storage Modules of Boxwood

7.3.1 Verification of Cache + Chunk Manager

Cache has public methods for reading and writing shared variables, flushing all dirty variables in the cache to Chunk Manager, and a “revoke” method for writing a single variable from Cache to Chunk Manager. Each variable accessed via Cache is addressed by the same unique handle that is also recognized by Chunk Manager. We verified the correctness of the abstract data store provided by the **Cache+Chunk Manager** combination assuming that Chunk Manager is implemented correctly.

We specified view^I for **Cache+Chunk Manager** as the set of $(\text{handle}, \text{byte-array})$ pairs stored in them. To extract view^I , for each *handle*, if there exists a cache entry associated with *handle*, *byte-array* is taken from the cache entry, otherwise, it is taken from Chunk Manager. We also verified the following invariants about Cache at runtime:

1. If a clean cache entry exists for *handle*, Cache and Chunk Manager must contain the same corresponding *byte-array*.
2. A cache entry must be in either the clean or dirty entries list.

For Cache we logged add and delete operations to hashtables and byte-array-copy operations from/to the cache entry buffers. Logging at this level of granularity was necessary for detecting the concurrency error described below.

7.3.2 The Bug Detected in the Cache Module

The error we found in Cache module is due to a method call (the call to `COPY-TO-CACHE(handle, buffer)` in line 23 of Figure 7.3) not being protected by the proper lock (`LOCK(clean)`). As a result, while a call to `WRITE(handle, buffer)` is executing and $data_i$ in the cache entry ($handle, data_i$) is being overwritten in-place (the `COPY-TO-CACHE` method), it is possible for a cache flush to be triggered. The cache flush on an entry for *handle* may be interleaved so that a corrupted version of *buffer* that contains part of old data from *data* and part of new data from *buffer* is written to Chunk Manager. At this point, the cache entry is marked “clean” since it is believed to have been written to Chunk Manager. The invariant 1 above is violated at this point, and a view refinement violation occurs. It is a lot harder for I/O-refinement and testing to detect this error: The corrupted state must lead to an error in the return value of a method as described in the following scenario: Before another write to this cache entry occurs, it is evicted from Cache but is not written back to Chunk Manager since it is marked “clean”. Then the block is read back into the cache, which retrieves the corrupted buffer from Chunk Manager. This bug had not been caught earlier by the Boxwood team. The code of Cache was updated so that the current version does not have the bug, though, the modification in the code was done without being aware of the buggy portion of the code.

7.4 Microbenchmarks

We evaluated performance and error detection capabilities of VYRD using small benchmark programs that either we wrote or are from standard class libraries. Among them are two implementations of the Multiset component, with `Vector`-based representation and

a binary search tree with data at leaves; and two classes `java.util.StringBuffer` and `java.util.Vector` from Java class library. The concurrency bugs were identified by previous studies as indicated below.

7.4.1 Multithreaded Java Class Libraries

We worked on the `java.util.Vector` and `java.util.StringBuffer` classes, and found concurrency errors that had been reported previously. Both examples consisted of approximately 1KLOC. Both I/O-refinement and view-refinement were able to easily detect the errors.

`java.util.StringBuffer`

This version of the `append` implementation in Figure 7.9 has a concurrency error due to the fact that the method argument, `sb`, is not locked throughout the method. This error was detected in [19]. During an execution of `append` by a thread t_1 , the state of `sb` can be modified by another thread t_2 between the invocation of the (synchronized) method `sb.length()` in line 3 of the `append` method and the invocation of `sb.getChars()` in line 7. For example, if `sb`'s length is more than 0 and t_2 executes `sb.setLength(0)` between these two invocations, an atomicity violation occurs.

We used a test case that created a set of `StringBuffer` instances and run different overloaded versions of `append` methods. An atomized version of `StringBuffer` was used as the specification. For the I/O-refinement checking, we run `toString` calls to the `StringBuffer` instances. Both I/O-refinement and view-refinement were successful in catching the error. Inequivalent return values from `toString` methods manifested the bug. Since the return point of `toString` methods that returned the inconsistent state of corrupted `StringBuffer`, debugging the execution trace took a long time to see that the error was the previously reported error. View-refinement was able to detect the error just after the `append` method that corrupted the `StringBuffer` instance committed. In fact I/O-refinement did a similar check since `toString` method return the abstract state of `StringBuffer`. However, because the execution of `toString` was not atomic, most `toString` calls missed the error even though their executions coincided with `append` methods that causes damage in `StringBuffer`. This revealed the strength of view-refinement in early detecting errors by

instantaneously looking at the part of `StringBuffer` that will appear in `toString` method, independent of any observer methods.

`java.util.Vector`

In Fig. 7.8, the code for this example is modified to contain one atomic action per line for illustration purposes. If line 3 in `lastIndexOf` is followed by line 5 in line 4 in `addElement`, this leads to an atomicity violation, which was discovered by [20]. In particular, if `elem == obj`, this would have led to an incorrect return value for `lastIndexOf`, which is a refinement violation.

We used a test case that created one instance of `Vector` and run `add`, `remove`, `indexOf` and `lastIndexOf` methods. A nondeterministic, atomized version of `Vector` was used as the specification. In this example, using a pool of values as explained in Section 6.1 made it easy to analyze the concurrency characteristic of methods running with the same arguments. As a result, the `lastIndexOf` calls to the `Vector` instance allowed I/O-refinement to detect the error. Concurrent execution of `add` and `lastIndexOf` methods with the same arguments revealed the error and I/O-refinement was able to see the error by looking at disagreeing return values from `lastIndexOf` methods in the implementation and the specification traces. Because the error in `Vector` did not cause any discrepancy in the state of `Vector` but manifest itself in the return value, this error could not be detected by comparing views of the implementation and the specification of `Vector` at commit points.

7.4.2 Multiset Implementations

We developed two different Java implementations of Multiset: one with an `Vector`-based representation (≈ 300 LOC), the other based on a binary search tree ($\approx 1\text{KLOC}$). The binary tree-based implementation stored the integers at leaf nodes, while the nodes above the leaves are used to traverse the tree down to the leaves fast. The tree-based implementation contained a complicated locking scheme that ensured atomic update of a subtree but also allowing multiple methods to run concurrently on different subtrees. Both implementations included compression mechanisms run by internal threads. Compression procedure for the vector-based implementation aims to gather the contents of the multiset scattered over the vector at the initial part of the vector to decrease execution time of `LookUp` and `Delete`

methods. Compression procedure for the tree-based implementation aims to reduce the size of the binary tree by removing the leaf nodes with empty content, and balance the subtrees rooted at particular nodes. VYRD was a valuable debugging aid during the development of the Multiset implementations. The implementations contained manually inserted bugs to evaluate the I/O-refinement and view-refinement checking procedures.

We used two test cases, each created one instance from a different representation of `Multiset` referred to above. They run `InsertPair`, `Insert`, `Delete` and `LookUp` methods. A nondeterministic, atomized version of `Multiset` was used as the specification for verifying both representations. Both I/O-refinement and view-refinement were successful in detecting the errors. I/O-refinement was able to catch the errors by analyzing the `LookUp` calls as regular. However, analyzing the trace after I/O-refinement signaled the error took considerably more time as the `LookUp` calls with wrong return values returned rather later than the point where the buggy scenario happened. view-refinement analyzed the abstract state of `Multiset` and was much faster in finding the errors.

7.5 Evaluating VYRD

7.5.1 Error Catching Capability

To compare the effectiveness of I/O-refinement and view-refinement checking, on the examples described before, we recorded the total number of methods executed by all the threads before the first error was detected by each technique. Table 7.1 reports the average of results from large numbers of repetitions of the same experiment. The last column presents the ratio of the CPU time required for running VYRD in view-refinement mode to that of running it in I/O-refinement only mode on the same trace. View-refinement’s superiority in early detection of error is apparent from these results. For example the view-refinement check catches the error in `Cache` after analyzing 14 method calls, while the I/O-refinement check detects it after 521 method calls. The additional cost of checking view-refinement must be traded off with its ability to detect errors much earlier in the trace. In our experiments, both I/O-refinement and view-refinement detected the error explained in Section 7.3.2, but I/O refinement required a much longer test run.

The only error in `java.util.Vector` was in an observer method and typically manifests itself without corrupting data structure state. Therefore view refinement was no better at

Program	Error Caught	Number of Threads	I/O-refinement	View-refinement	Run-time ratio
Multiset-Vector	Moving acquire in FindSlot	4	1308	25	1.03
		8	773	21	
		16	758	10	
		32	820	6	
Multiset-BinaryTree	Unlocking parent before insertion	4	3648	736	1.38
		8	930	217	
		16	330	76	
		32	262	78	
java.util.-Vector	Taking length non-atomically in lastIndexOf()	4	219	219	2.83
		8	58	58	
		16	52	52	
		32	25	25	
java.util.-StringBuffer	Copying from an unprotected StringBuffer	4	195	90	3.46
		8	152	63	
		16	124	19	
		32	29	17	
BLinkTree	Allowing duplicated data nodes	2	2198	405	5.68
		4	4450	483	
		8	3332	611	
		10	2763	342	
		16	1069	301	
		25	3692	515	
		32	2111	715	
Cache	Writing an unprotected dirty cache entry	4	521	14	1.87
		8	805	8	
		10	599	10	
		16	302	29	
		25	539	26	
		32	311	34	

Table 7.1: Time to detection of errors

detecting it than I/O refinement in our tests.

7.5.2 Computational Overhead Measurements

Table 7.2 compares the logging overhead, the total time spent for logging the execution trace, at runtime only with the runtime of the unmodified program, assuming I/O or view-refinement will be run off-line using the saved log later. All numbers reported are CPU seconds on a 2.4 GHz Intel® Pentium® PC with 1GB of RAM. In the first three examples and in Cache, in which the mutator methods perform a large number of shared variable writes, the overhead of logging information required for view refinement is significantly larger than that for I/O refinement. This can be seen from example from the column for Multiset examples: the ratio of logging for view-refinement to logging for I/O-refinement is around 60 for vector representation, as it is 2.6 for binary tree representation. This is because these view-refinement requires logging all the actions performed as well as call, return and commit actions. For the programs for which actions are selected in fine granularity, the difference between logging for view-refinement and I/O-refinement increases due to the large number of actions performed along a method execution. However, in `java.util.Vector`, `java.util.StringBuffer` and `BLinkTree`, the difference between logging overhead for I/O-refinement and view-refinement is much less. This is obvious from the results for `BLinkTree` example: the ratio of logging overhead for I/O-refinement over the runtime of the test program runtime is 17%, and the ratio for view-refinement is 20%.

Table 7.3 presents results on the computational cost of checking view-refinement and

Program	Runtime without logging	Logging for I/O-refinement	Logging for View-refinement
Multiset-Vector	2.14	0.01	0.6
Multiset-BinaryTree	0.3	0.03	0.08
Vector	0.3	0.12	0.14
StringBuffer	0.95	0.08	0.11
BLinkTree	16	2.8	3.2
Cache	2.2	1.25	1.7

Table 7.2: Overhead of logging for I/O- and view-refinement

Program	#Threads/ #Methods	Program runtime standalone	Runtime with logging	Runtime with logging & VYRD	VYRD alone (off-line)
Multiset-Vector	10/50	1.36	2.14	3.68	1.96
Multiset-BinaryTree	10/50	0.22	0.3	0.5	0.26
Vector	20/200	0.15	0.3	1.3	1.2
StringBuffer	10/30	0.08	0.95	1.85	1.12
BLinkTree	10/600	12.8	16	38	23.4
Cache	10/700	1.2	2.2	3.4	1.4

Table 7.3: Running time breakdown

compares it with the computational cost of running the program and logging information. All running times are CPU seconds on the same machine. Checking view-refinement requires replaying the implementation trace once more in addition to executing corresponding methods in the specification. This increases the computational cost compared to the I/O-refinement checking. For example the ration of the runtime for checking I/O-refinement over the runtime for testing is 27%, as this ratio is 137% for view-refinement. However the computation time is still tolerable, especially for the industrial examples. The impact of instrumentation and logging is low for coarse-grained analysis (**StringBuffer** and **BLinkTree**, e.g.), as it is also moderate for the programs analyzed in fine granularity (**Multiset** and **Cache**, e.g.).

This chapter reported the achievements from our tool VYRD. Results show that our tool is scalable so that it can be applied to large implementations conveniently. The following chapter concludes this thesis by giving future directions of this study that takes refinement checking further by generalizing the current verification method and proposing a new static approach.

```

INSERT(key, data)
1  ▷ ptr is the pointer to the current parent node of the the current node pointed by p
2  completed ← false
3  stack ← MOVE-DOWN-AND-STACK()
4  repeat
5      ptr ← POP(stack)
6      repeat
7          found ← true
8          LOCK(ptr)
9          A ← READ-NODE(ptr)
10         if A is a leaf pointer node and key is in A
11             then
12                 OVERWRITE(A, data)
13                 UNLOCK(ptr) ▷ Commit point 1
14                 RETURN
15             if key > HIGHVALUE(A)
16                 then
17                     UNLOCK(ptr)
18                     found ← false
19                     ptr ← GET-RIGHT-POINTER(A)
20             elseif key < LOWVALUE(A)
21                 then
22                     UNLOCK(ptr)
23                     found ← false
24                     ptr ← GET-LEFT-POINTER(A)
25         until found
26     if p = null
27         then
28             p ← CREATE-DATA-NODE(key, data)
29     if A is safe to insert d
30         then
31             p ← INSERT-INTO-SAFE(p, A)
32             If A is a leaf pointer node then ▷ Commit point 2
33             completed ← true
34     elseif A is not the root
35         then
36             p ← INSERT-INTO-UNSAFE(p, A)
37             If A is a leaf pointer node then ▷ Commit point 3
38     else
39         p ← INSERT-INTO-UNSAFE-ROOT(p, A)
40         completed ← true
41         If A is a leaf pointer node then ▷ Commit point 4
42 until completed

```

Figure 7.2: The **Insert** subroutine with the conditional commit point annotations.

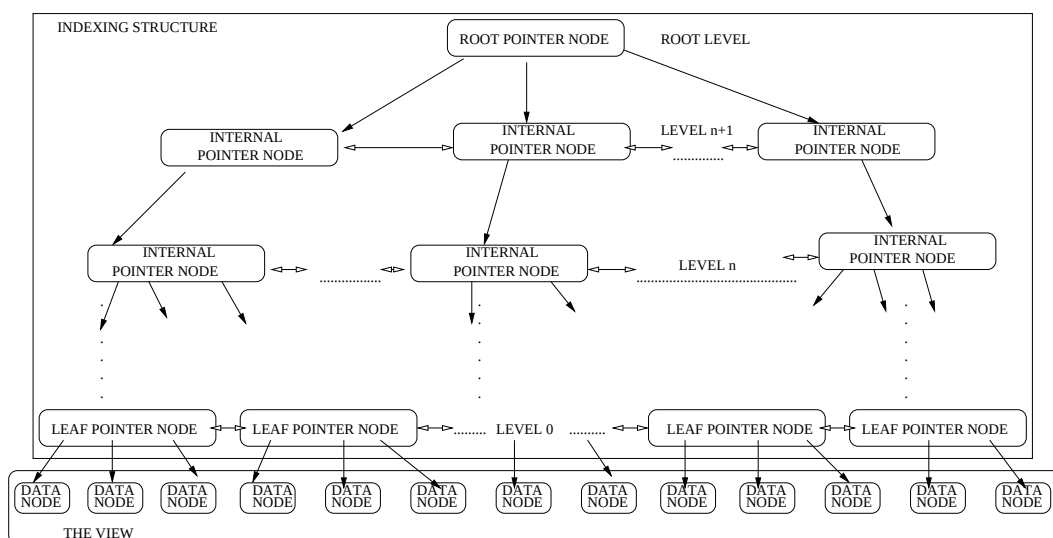


Figure 7.3: Decomposition of a b-link tree to indexing structure and leaf structure.

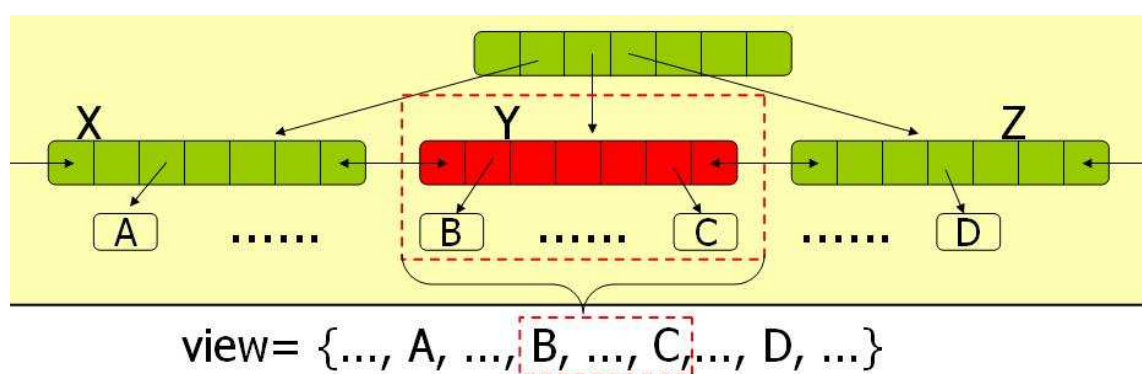


Figure 7.4: Part of the view updated incrementally for a leaf pointer node.

```

WRITE(handle, buffer)
1  RECLAIMLOCK.BEGINREAD()
2  LOCK(clean)
3  ce ← GET-CLEAN-ENTRY(handle)
4  de ← GET-DIRTY-ENTRY(handle)
5  UNLOCK(clean)
6  if ce = null and de = null
7      then
8          RECLAIMLOCK.ENDREAD()
9          te ← MAKE-NEW-ENTRY(handle)
10         RECLAIMLOCK.BEGINREAD()
11         COPY-TO-CACHE(buffer, te)
12         LOCK(clean)
13         ADD-TO-DIRTY-LIST(handle, te) ▷ Commit point 1
14         UNLOCK(clean)
15     elseif ce = null
16         then
17             LOCK(clean)
18             REMOVE-FROM-CLEAN-LIST(handle)
19             COPY-TO-CACHE(buffer, ce)
20             ADD-TO-DIRTY-LIST(handle, ce) ▷ Commit point 2
21             UNLOCK(clean)
22     else
23         COPY-TO-CACHE(buffer, de) ▷ Commit point 3
24  RECLAIMLOCK.ENDREAD()

```

Figure 7.5: The pseudo code for WRITE method of Cache

```

COPY-TO-CACHE(buffer, entry)
1  for i ← 0 to LENGTH-OF(buffer) − 1
2      do
3          entry.data[i] ← buffer[i]

```

Figure 7.6: The pseudo code for COPY-TO-CACHE method of Cache

```

FLUSH()
1  LOCK(clean)
2  while more entry in dirty list
3      do
4           $te \leftarrow \text{NEXT-DIRTY-ENTRY}()$ 
5          if  $te$  is old enough to flush
6              then
7                  BOXWOOD-ALLOCATOR-WRITE( $te.handle, te.data$ )
8                  ADD-TO-VICTIMS-LIST( $te$ )
9  while more entry in victims list
10     do
11          $te \leftarrow \text{NEXT-VICTIM-ENTRY}()$ 
12         REMOVE-FROM-DIRTY-LIST( $te$ )
13         ADD-TO-CLEAN-LIST( $handle, te$ )
14  UNLOCK(clean)  $\triangleright$  Commit point

```

Figure 7.7: The pseudo code for FLUSH method of Cache

```

public synchronized void addElement(Object obj) {
1   modCount++;
2   ensureCapacityHelper(elementCount + 1);
3   elementData[elementCount] = obj;
4   elementCount++;
5 }

public int lastIndexOf(Object elem) {
1   int count = Read(elementCount) - 1;
2   return lastIndexOf(elem, count);
3 }

```

Figure 7.8: Code fragments illustrating the error in `java.util.Vector`

```
public synchronized StringBuffer append(StringBuffer sb) {  
1  if (sb == null)  
2      sb = NULL;  
3  int len = sb.length();  
4  int newcount = count + len;  
5  if (newcount > value.length)  
6      expandCapacity(newcount);  
7  sb.getChars(0, len, value, count);  
8  count = newcount;  
9  return this;  
10 }
```

Figure 7.9: Code fragments illustrating the error in `StringBuffer`

Chapter 8

CONCLUSIONS**8.1 State of the Art**

Components implement abstract data structures that are building blocks of many concurrent software applications including file systems and database management systems. A functionally correct component provides to the rest of the system a reliable data structure that is free of concurrency errors. Functional correctness for these systems is one of the fundamental necessities for keeping data without any damage. Verifying industrial-scale concurrent software is particularly a hard problem today because of the added complexity in the behavior of the system due to different possible interleavings of the threads that run the application.

Clients of a component expect its methods to run as if each operation on the abstract data structure appears to be performed atomically. Abstract atomicity for a component guarantees absence of common concurrency-related errors and reduces the concurrency errors in the higher levels of the system. Performance needs force developers to use intricate synchronization mechanisms, which makes it hard to reason about abstract atomicity of components.

There have been several static and dynamic approaches to check atomicity-like criteria in order to detect concurrency errors. Concurrency bugs are notoriously difficult to detect and reproduce through testing. Model checking using abstractions is good at verifying local properties. However, it suffers from state space explosion problem. Additionally more global specifications such as refinement require most implementation details to be included in the implementation model, abstraction techniques do not achieve much reduction in the state space in this case. Thread-modular verification aims to verify properties of individual threads with respect to the rest of the system specified as environment assumptions. The main difficulty to apply these techniques is deriving the environment assumptions modeling concurrent effects of the other threads, which can blow up for complex software. Pure test-

ing, while computationally practical, suffers from low coverage and limited observability.

8.2 Summary

In this thesis, we presented two notions of refinement, I/O-refinement and view-refinement, that define correctness criteria for reasoning about concurrency characteristics of components in order to find concurrency errors. Refinement formalizes the requirement that method invocations by concurrent threads appear to be executed atomically in a linear order and are consistent with an executable specification. In other words, each execution of the implementation must be equivalent to an execution of the specification. The specification can be given as a separate program or be obtained by constructing an atomized version of the implementation. I/O-refinement requires a particular correspondence between the method execution signatures consisting of arguments and return values, while view-refinement seeks correspondence between the implementation and specification states, which enables detection of concurrency errors as soon as they cause a discrepancy in the implementation state.

We introduced a runtime technique for analyzing execution traces of the component for refinement criteria. We use a log to communicate the test program to the refinement checker, which minimizes most of concurrency impact on the testing. Although our technique can be seen between testing and exhaustive verification, it adds precision and observability to pure testing that allows us to find intricate bugs.

The main benefit of our checking strategy is it can check the refinement at runtime using the “original implementation code” of the data structure without any abstraction. This allows us to check even much more complex industrial software including millions of lines of code. We applied these techniques to some microbenchmarks including some utility classes in Java class library as well as some industrial scale systems including the Boxwood project. We presented experimental results demonstrating the efficacy of and the trade-offs offered by our techniques. Experimental results on industrial-scale designs showed that the added observability of view refinement makes the early detection of concurrency errors much more likely. Run-time checking of refinement promises to be a powerful verification approach with reasonable computational cost.

8.3 Future Directions

8.3.1 VyrdMC: Verifying Refinement During Model Checking

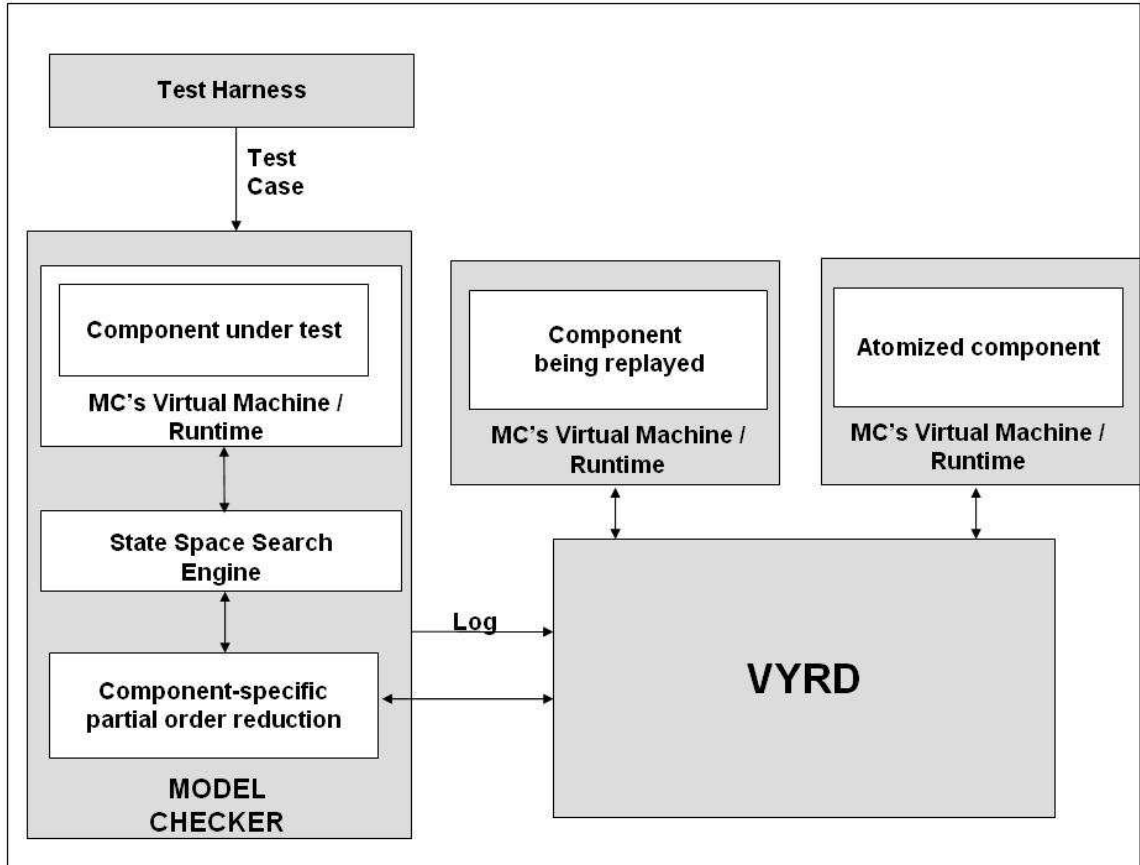


Figure 8.1: The architecture of VyrdMC

While runtime checking refinement as explained in the previous chapters provides significantly better verification than testing, it also suffers from low coverage. If concurrency errors are not triggered by a test program, refinement checking is no better than testing. In our experience, to trigger a particular concurrency error, tests with many concurrent threads, each performing a large number of method calls, were often needed, while the same error could have been triggered with many fewer threads and method calls if the correct (but unknown a priori) thread interleaving were enforced. This suggested that very small test programs might be able to reveal concurrency errors if all thread interleavings are systematically explored and view refinement checking is performed.

This observation motivates an improved technique proposed as a verification framework, called VyrdMC [9], in which an execution-based model checker guides the testing of concurrent software by exploring all distinct thread interleavings for a given fixed test program. We focus on model checkers like Java Pathfinder [81] that execute programs as is, without performing any abstraction. Clearly, such model checkers could be used to drive programs with the purpose of performing any runtime check. What distinguishes our approach (depicted in Figure 8.1) is the combination of the strengths of the following three approaches:

1. Testing is used to drive VYRD to non-trivial, sufficiently large and complicated “anchor” states. At anchor states, the specification and implementation are consistent. Starting at the anchor state, the test program issues a very small number of threads with few operations each. In each test case, the operations and their arguments are chosen so that they focus on the same portion of the state of the program and are intuitively likely to contend for access to resources and to trigger concurrency errors.
2. Our refinement checking tool VYRD provides improved observability, which makes it possible to detect errors triggered by the small test cases starting from the anchor state. During the course of such short tests, errors are unlikely to propagate to a method’s return value.
3. Model checking tackles the coverage problem by generating all qualitatively distinct execution traces starting from the anchor state. The model checker needs to only explore different thread interleavings since everything else is fixed. This very restricted application of model checking and the choice of very small test programs makes our technique computationally feasible even for industrial-scale programs.

A side benefit of using a model checker such as Java PathFinder is the re-use of its runtime environment – essentially a virtual machine (VM) – to automate monitoring and logging tasks (see Section 5.2) that are required for refinement checking and that had to be performed manually previously. An example is the detection and recording of updates to shared variables when a fine-granular analysis is done. Automating this phase makes refinement checking a lot more easily usable. VYRD can also use the runtime environment

of the model checker for its replaying mechanism, i.e., driving independent instances of the implementation and the specification for verification purposes.

8.3.2 *Measuring Coverage for Concurrency*

We are exploring several hybrid techniques in which coverage metrics serve as the link between model checking and testing tools and enable us to (i) quantify the adequacy testing/verification performed, (ii) communicate partial results and testing/verification goals between tools, (iii) direct testing effort towards unexplored, quantitatively distinct executions of a program that are interesting for a particular purpose. Consequently we proposed a coverage metric, called “location pairs” (LP) [8] focused on the concurrency aspects of test executions. The LP metric was inspired by a pattern common to atomicity and refinement violations that were found in industrial software examples and in the literature [39, 69, 19]. The LP metric can express as a coverage goal for many common error-prone scenarios that are described in the category “Code Assumed to Be Protected” of [69], which are particularly relevant for atomicity and refinement violations.

The fact that the LP metric seems to correspond well with certain types of concurrency errors makes it a promising tool for guiding manual validation effort and catching errors beyond the reach of methods based on state-space traversal. The LP metric captures errors caused by such unforeseen interleavings. The oversight typically occurs because of erroneous assumptions about the environment or the belief that a synchronization mechanism makes a certain interleaving impossible. Given an unexamined but apparently reachable location pair as a target, the programmer can reason about conditions that need to be set up to reach the target or provide the coverage analysis tool hints that help it prove that the target location pair is unreachable. In this way, the LP metric helps test writers explore qualitatively distinct and error-prone scenarios.

8.3.3 *Static Methods for Verifying Refinement*

We are also investigating statically checkable verification conditions for the view definition of a component that will allow us to show view-refinement with respect to a sequential specification. Under these conditions we can show that view-refinement implies I/O-refinement, the essential verification goal for concurrent components to ensure abstract atomicity. These

conditions express the relationship between the view definition of the component and the specifications of its public methods, and aid the developer to reason about correctness of her method implementation from a more abstract point of view. We are working on a static checking algorithm that, given a view definition for the component in the form of a formally written abstraction function, checks the necessary conditions for the view of the component to be observed/updated atomically, by making use of symbolic execution and a type and effect system.

In this thesis, we present a runtime technique for checking that a concurrently-accessed component conforms to an executable specification that contains an atomic method per data structure operation. The conditions for conformance are formalized as two notions of refinement, I/O-refinement and view-refinement. The refinement criteria is used to reason about abstract atomicity of components that ensure functional correctness for the rest of the system that are built these components. We paid special attention to the general applicability and scalability of the techniques and to minimizing their concurrency and performance impact. The result is a lightweight runtime verification method that provides a significant improvement over testing for concurrent programs. The main benefit of our checking strategy is it can check the refinement at runtime using the original implementation code of the data structure without any abstraction. This allows us to check even much more complex industrial software including millions of lines of code. VYRD is very effective, with modest computational cost, in detecting concurrency errors and promising for future verification methods that will be facilitated by the observations and solutions to runtime refinement checking problem developed in this study.

BIBLIOGRAPHY

- [1] Java concurrent class library (java.util.concurrent)
<http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-summary.html>
- [2] N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C. In B. Magnusson, editor, Proc. of ECOOP02, volume 2374 of LNCS, pages 415-440. Springer, 2002.
- [3] Doug Lea. Concurrent Programming in Java(TM): Design Principles and Pattern (2nd Edition). Addison-Wesley Press. 1999.
- [4] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy and M. D. Hill. Specifying System Requirements for Memory Consistency Models. Technical Report CSL-TR-93-594, Stanford University, December 1993. Also available as Computer Sciences Technical Report 1199, University of Wisconsin, Madison, December 1993.
- [5] Burns A., Davis G. Concurrent Programming. Addison-Wesley, 1993.
- [6] L. Lamport. The temporal logic of actions. ACM TOPLAS, 16(3):872– 923, March 1994.
- [7] T. Elmas, S. Tasiran, S. Qadeer. VYRD: VerifYing Concurrent Programs by Runtime Refinement-Violation Detection. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI), Chicago, Illinois, USA, June 12-15, 2005.
- [8] S. Tasiran, T. Elmas, G. Bolukbasi, M. E. Keremoglu A Novel Test Coverage Metric for Concurrently-Accessed Software Components Fifth International Workshop on Formal Approaches to Testing of Software (FATES 2005) University of Edinburgh, Scotland, UK, July 11, 2005.

-
- [9] T. Elmas, S. Tasiran. VyrdMC: Driving Runtime Refinement Checking with Model Checkers. Fifth Workshop on Runtime Verification (RV'05). The University of Edinburgh, Scotland, UK. July 12, 2005.
 - [10] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proc. 3rd Annual Symposium on Logic in Computer Science*, pp. 165–175. IEEE Computer Society Press, 1988.
 - [11] F. Chen and G. Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Electronic Notes in Theoretical Computer Science*, Vol. 89. Elsevier, 2003.
 - [12] Klaus Havelund and Grigore Rosu. Monitoring Java programs with Java PathExplorer. In *Proceedings First Workshop on Runtime Verification, RV'01*, Paris, France, July 23
 - [13] Owicki, S. and Gries, D. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM* 19(5), 279–285. 1976.
 - [14] Grigore Rosu, Koushik Sen. An Instrumentation Technique for Online Analysis of Multithreaded Programs: 18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 16, 2004.
 - [15] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Softw. Eng.*, 23(5):279–295, May 1997.
 - [16] M. L. Crane and J. Dingel. Runtime conformance checking of objects using Alloy. In *Electronic Notes in Theoretical Computer Science*, Vol. 89. Elsevier, 2003.
 - [17] C. Flanagan. Verifying Commit-Atomicity Using Model-Checking. In *SPIN '04: The SPIN Workshop on Model Checking of Software*. Springer-Verlag, 2004.
 - [18] C. Flanagan, S. Freund, and S. Qadeer. Exploiting purity for atomicity. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*. ACM Press, 2004.

-
- [19] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multi-threaded programs. In *Proc. 31st ACM Symposium on Principles of Programming Languages*, pp. 256–267, 2004.
 - [20] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. ACM SIGPLAN 2003 Conf. on Programming language design and implementation*, pages 338–349. ACM Press, 2003.
 - [21] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.
 - [22] Lipton, R. J. Reduction: a method of proving properties of parallel programs. *Commun. ACM* 18, 12 (Dec. 1975), 717–721.
 - [23] M. Ji and E. Felten. Scan-based scheduling and layout in a reliable write-optimized file system. Technical Report TR-661-02, Princeton University, Department of Computer Science, 2002.
 - [24] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *(To appear) Proceedings of the Sixth Symposium on Operating Systems Design and Implementation, OSDI 2004*. <http://research.microsoft.com/research/sv/Boxwood>
 - [25] S. Park and D. L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, 1998.
 - [26] S. Tasiran, A. Bogdanov, and M. Ji. Detecting concurrency errors in file systems by runtime refinement checking. Technical Report HPL-2004-177, HP Laboratories, 2004.
 - [27] S. Tasiran, Y. Yu, and B. Batson. Using a formal specification and a model checker to monitor and guide simulation. In *Proceedings of the 40th Design Automation Conference*, pages 356–361. ACM, 2003.
 - [28] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Electronic Notes in Theoretical Computer Science*, Vol. 89. Elsevier, 2003.

-
- [29] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead*, volume
 - [30] C. Flanagan, S. Qadeer, and S. A. Seshia. A modular checker for multithreaded programs. Technical Note 2002-001, Compaq Systems Research Center, 2002.
 - [31] C. Flanagan and S. Qadeer. Thread-modular model checking. *Proceedings of the SPIN Workshop on Software Verification*, 2003.
 - [32] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. Technical Report 536, Computer Science Dept., Indiana University, 2000.
 - [33] Elsa Gunter and Doron Peled. Tracing the executions of concurrent programs. *Electronic Notes in Theoretical Computer Science*, Vol. 70, Issue.4, 2002.
 - [34] K. Havelund, S. Stoller, and S. Ur. Benchmark and Framework for Encouraging Research on Multi-Threaded Testing Tools. *PADTAD'03, Parallel and Distributed Systems: Testing and Debugging* Nice, France, April 22-26, 2003.
 - [35] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
 - [36] John Hatcliff and Matthew Dwyer. Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software. In the *Proceedings of CONCUR'01*. June, 2001.
 - [37] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.
 - [38] Y. Sagiv. Concurrent operations on B-trees with overtaking. *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, p.28-37, March 25-27, 1985, Portland, Oregon, United States

-
- [39] Serdar Tasiran, Shaz Qadeer. Runtime Refinement Checking of Concurrent Data Structures. In Proc. RV'04 - Fourth Workshop on Runtime Verification, April 3, 2004, Barcelona, Spain.
 - [40] S.N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. Proceedings of the Fifth ECOOP Workshop on Formal Techniques for Java-like Programs, 2003.
 - [41] C. Flanagan and S. Freund. Type-based race detection for java. In Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation, Vancouver, Canada, June 2000.
 - [42] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages and Applications, Tampa Bay, FL, Oct. 2001.
 - [43] ESP: Path-Sensitive Program Verification in Polynomial Time. Manuvir Das, Sorin Lerner, and Mark Seigle. PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, June 2002.
 - [44] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Research Report 159, Compaq Systems Research Center, December, 1998.
 - [45] Thomas Ball, Sriram K. Rajamani, The SLAM Project: Debugging System Software via Static Analysis, POPL 2002, January 2002, pages 1-3.
 - [46] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In O. Sokolsky and M. Viswanathan, editors, Electronic Notes in Theoretical Computer Science, volume 89. Elsevier, 2003.
 - [47] C. Flanagan and S. Qadeer. Types for atomicity. In Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation, pages 1-12. ACM Press, 2003.

-
- [48] L. Lamport, How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Trans. on Computers* C-28, 9 (September 1979), 690-691.
 - [49] Michelle L. Crane and Juergen Dingel, Runtime Conformance Checking of Objects Using Alloy, *Electronic Notes in Theoretical Computer Science*, Vol. 89 (2) (2003)
 - [50] S. Park, D. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, volume 31(4), pages 355-376, July 1998.
 - [51] Cormac Flanagan, Stephen N. Freund: Atomizer: a dynamic atomicity checker for multithreaded programs. *POPL 2004*: 256-267.
 - [52] F. Nielson and H. R. Nielson. Type and effect systems. In E. R. Olderog and B. Steffen, editors, *Correct System Design*, number 1710 in *Lecture Notes in Computer Science*, pages 114–136. Springer-Verlag, 1999.
 - [53] William R. Bush, Jonathan D. Pincus and David J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software Practice and Experience* 2000, 30:775-802.
 - [54] Thomas Ball, Rupak Majumdar, Todd Millstein, Sriram K. Rajamani, Automatic Predicate Abstraction of C Programs, *PLDI 2001*, *SIGPLAN Notices* 36(5), pp. 203-213.
 - [55] H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch. Shared Memory Consistency Conditions for Nonsequential Execution: Definitions and Programming Strategies. *SIAM J. Computing*, Vol. 27, No. 1, pages 65–89, February 1998.
 - [56] S. V. Adve. Designing Memory Consistency Models for Shared-Memory Multiprocessors. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, December 1993.
 - [57] Thomas Ball, Sagar Chaki, Sriram K. Rajamani. Parameterized Verification of Multithreaded Software Libraries. *TACAS 2001*, *LNCS 2031*, April 2001, pp. 158-173.

-
- [58] Sergio Antoy , Dick Hamlet. Automatically Checking an Implementation against Its Formal Specification. *IEEE Transactions on Software Engineering*, v.26 n.1, p.55-69, January 2000.
- [59] Feng Chen and Grigore Rosu.. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In Oleg Sokolsky and Mahesh Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.
- [60] Guillaume Brat, Klaus Havelund, Seung-Joon Park, and Willem Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, September 2000.
- [61] S.Graf, H.Saïdi. Verifying Invariants Using Theorem Proving. In *Proceedings of the 8th Conference on Computer-Aided Verification (CAV'96)* , Rutgers University, New Jersey, July 1996.
- [62] Barnett, M., Schulte, W. Contracts, components and their runtime verification on the .NET platform. Technical Report MSR-TR-2002-38, Microsoft Research.
- [63] S. Tasiran, Y. Yu, and B. Batson. Using a formal specification and a model checker to monitor and guide simulation. In *Proceedings of the 40th Design Automation Conference*, pages 356-361. ACM, 2003.
- [64] H. Attiya and J. L. Welch. Sequential Consistency versus Linearizability. *ACM Trans. on Computer Systems*, Vol. 12, No. 2, pp. 99–122. 1994.
- [65] S. Tasiran, A. Bogdanov, and M. Ji. Detecting Concurrency Errors in Filesystems by Runtime Refinement Checking. Technical Report HPL-2004-177, HP Laboratories, 2004.
- [66] W. Visser, C. Pasareanu, S. Khurshid. Test Input Generation with Java PathFinder. *Proceedings of ISSTA 2004 Boston, MA, July 2004*.

-
- [67] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In Proc. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), Warsaw, Poland, April 2003.
 - [68] C. Artho, K. Havelund, and A. Biere. High-Level Data Races. In VVEIS '03, Int'l Workshop on Verification and Validation of Enterprise Information Systems.
 - [69] Eitan Farchi, Yarden Nir, Shmuel Ur. Concurrent Bug Patterns and How to Test Them. 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, page 286.
 - [70] B. Long and P. A. Strooper. A classification of concurrency failures in java components. 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France., page 287.
 - [71] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. Data Flow Analysis for Checking Properties of Concurrent Java Programs. Proceedings of the 21st International Conference on Software Engineering, pp. 399 – 410, May 1999
 - [72] H. Saïdi. Modular and incremental analysis of concurrent software systems. In 14th IEEE International Conference on Automated Software Engineering, pages 92-101. IEEE Computer Society Press, 1999.
 - [73] S.Graf, H.Saïdi. Construction of abstract state graphs with PVS. In Proceedings of the 9th Conference on Computer-Aided Verification (CAV'97), Haifa, Israel, June 1997.
 - [74] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy Abstraction. Proceedings of the 29th Annual Symposium on Principles of Programming Languages, ACM Press, pp. 58-70, 2002.
 - [75] W. Visser, S. Park, J. Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In Third Workshop on Formal Methods in Software Practice, August 2000.

-
- [76] B. Long, P. Strooper, and L. Wildman. A method for verifying concurrent java components based on an analysis of concurrency failures. *Concurrency and Computation: Practice and Experience*, 00(1-7):1–13, 2005.
 - [77] B. Long and P. A. Strooper. A classificator.of concurrency failures in java components. 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France., page 287.
 - [78] P.Godefroid. Partial order methods for the verification of concurrent systems-an approach to the state-explosion problem. In *Lecture Notes in Computer Science*, volume 1032. Springer-Verlag, 1996.
 - [79] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 18(3):206–215, 1992.
 - [80] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In Proceedings of the 24th ACM Symposium on Principles of Programming Languages, pages 174–186, Paris, January 1997.
 - [81] Guillaume Brat, Klaus Havelund, Seung-Joon Park, and Willem Visser. Model Checking Programs. In IEEE International Conference on Automated Software Engineering (ASE), September 2000.

VITA

Tayfun Elmas was born in Fethiye (Muğla), Turkey on November 6, 1981. He received his B.Sc. degree in Computer Engineering from EGE University, İzmir, in 2003. From September 2003 to December 2005, he worked as a teaching and research assistant at Center for Advanced Design Technologies (CADT) at KOÇ University, Turkey. His project on “verifying concurrent object-oriented software” was sponsored by TÜBİTAK and Microsoft® Research. He has published several papers about verification of concurrent software components for the following conferences PLDI’05 (Chicago, U.S.) and UYMS’05 (Ankara, Turkey), and workshops RV’05 (Scotland) and FATES’05 (Scotland). He worked for Microsoft® Research in Redmond as a research intern to improve Zing model checker during summer 2005. He plans to pursue a PhD study at KOÇ University in the future.