A CONCURRENCY AND TIME CENTERED FRAMEWORK FOR CERTIFICATION OF AUTONOMOUS SPACE SYSTEMS

A Dissertation

by

DAMIAN DECHEV

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2009

Major Subject: Computer Science

A CONCURRENCY AND TIME CENTERED FRAMEWORK FOR CERTIFICATION OF AUTONOMOUS SPACE SYSTEMS

A Dissertation

by

DAMIAN DECHEV

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Bjarne Stroustrup
Committee Members,	Jaakko Järvi
	Rabi N. Mahapatra
	Raytcho Lazarov
Head of Department,	Valerie E. Taylor

December 2009

Major Subject: Computer Science

ABSTRACT

A Concurrency and Time Centered Framework for
Certification of Autonomous Space Systems. (December 2009)

Damian Dechev, B.S., University of Indianapolis;
M.S., University of Delaware
Chair of Advisory Committee: Dr. Bjarne Stroustrup

Future space missions, such as Mars Science Laboratory, suggest the engineering of some of the most complex man-rated autonomous software systems. The present process-oriented certification methodologies are becoming prohibitively expensive and do not reach the level of detail of providing guidelines for the development and validation of concurrent software. Time and concurrency are the most critical notions in an autonomous space system. In this work we present the design and implementation of the first *concurrency* and *time* centered framework for product-oriented software certification of autonomous space systems. To achieve fast and reliable concurrent interactions, we define and apply the notion of *Semantically Enhanced Containers (SEC)*. SECs are data structures that are designed to provide the flexibility and usability of the popular ISO C++ STL containers, while at the same time they are hand-crafted to guarantee domain-specific policies, such as conformance to a given concurrency model. The application of nonblocking programming techniques is critical to the implementation of our SEC containers. Lock-free algorithms help avoid the hazards of deadlock, livelock, and priority inversion, and at the same time deliver fast and scalable performance. Practical lock-free algorithms are notoriously difficult to design and implement and pose a number of hard problems such as ABA avoidance, high complexity, portability, and meeting the linearizability correctness requirements. This dissertation presents the design of the first lock-free dynamically resizable array. Our approach offers a set of practical, portable, lock-free, and linearizable STL vector operations and a fast and space efficient implementation when compared to the alternative lock- and STM-based techniques. Currently, the literature does not offer an explicit analysis of the ABA problem, its relation to the most commonly applied nonblocking programming techniques, and the possibilities for its detection and avoidance. Eliminating the hazards of ABA is left to the ingenuity of the software designer. We present a generic and practical solution to the fundamental ABA problem for lock-free descriptor-based designs. To enable our SEC container with the property of validating domain-specific invariants, we present Basic Query, our expression template-based library for statically extracting semantic information from C++ source code. The use of static analysis allows for a far more efficient implementation of our nonblocking containers than would have been otherwise possible when relying on the traditional run-time based techniques. Shared data in a real-time cyber-physical system can often be polymorphic (as is the case with a number of components part of the Mission Data System's Data Management Services). The use of dynamic cast is important in the design of autonomous real-time systems since the operation allows for a direct representation of the management and behavior of polymorphic data. To allow for the application of dynamic cast in mission critical code, we validate and improve a methodology for constant-time dynamic cast that shifts the complexity of the operation to the compiler's static checker. In a case study that demonstrates the applicability of the programming and validation techniques of our certification framework, we show the process of verification and semantic parallelization of the Mission Data System's (MDS) Goal Networks. MDS provides an experimental platform for testing and development of autonomous real-time flight applications.

To my parents, Tsvetanka and Decho, for their endless encouragement and love.

ACKNOWLEDGMENTS

I thank my dissertation advsior and mentor, Dr. Bjarne Stroustrup, for always encouraging free and independent creative thinking and promoting the search for practical and efficient engineering solutions. His patience, guidance, fostering of original ideas, resourceful and broad outlook, and respectful and friendly attitude created a unique and inspiring research environment in our research lab. I am grateful to my committee members, Dr. Jaakko Järvi, Dr. Rabi N. Mahapatra, and Dr. Raytcho Lazarov, for their guidance, helpful advice, patience, and service. I would like to thank my good colleague and friend, Peter Pirkelbauer, for being a great friend and good collaborator on a number of interesting research projects. I am thankful to my colleagues and mentors at the Jet Propulsion Laboratory, Dr. William K. Reinholtz, Dr. Nicolas Rouquette, Dr. David Wagner, and Dr. Garard Holzmann, for giving me the exciting chance to collaborate with them on a number of interesting projects, for their welcoming and friendly attitude, and for providing me with the unique perspective on the application of C++ for space missions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In this work we present the design and application of the first *concurrency* and *time* centered framework for product-oriented software certification of autonomous space systems. The process of software certification establishes the level of confidence in a software system in the context of its *functional* and *safety* requirements. A software certificate contains the evidence required for the system's independent assessment by an authority having minimal knowledge and trust in the technology and tools employed [1]. Providing such certification evidence may require the application of a number of software development, analysis, verification, and validation techniques [2]. The goal of our work is not to provide a grade or a rating of the software development process and the existing software for cyber-physical systems. Instead, we engineer a number of programming and validation techniques that play a critical role for the design and implementation of reliable real-time autonomous software.

This dissertation offers the following contributions: the design of the first lock-free dynamically resizable array, detailed analysis and generic solution to the fundamental ABA problem, a comparison study of the available state-of-the-art nonblocking techniques, the application of static analysis to deliver most efficient and reliable nonblocking designs than otherwise would have been possible, improved and verified constant-time dynamic cast operation for polymorphic data, suggests the scope and dimensions of product-oriented certification, a study on the applicability of lock-free designs in mission critical code, the design and implementation of a framework for formal verification and automatic parallelization of control modules for cyber-physical

This dissertation follows the style of *IEEE Transactions on Software Engineering.*

systems.

A. Goals

The goal of this work is to provide the definition, design, and implementation of the first *concurrency* and *time* centered framework for product-oriented software certification of autonomous space systems. To achieve fast and reliable concurrent interactions, we define and apply the notion of *Semantically Enhanced Containers (SEC)*. SECs are data structures designed to provide the flexibility and usability of the popular ISO C++ STL containers, while at the same time they are hand-crafted to guarantee domain-specific policies, such as the validity of given semantic invariants or the conformance to a specific concurrency model. In particular, to meet the challenges of engineering mission critical code, we require a SEC to provide the following: a) built-in safe concurrent synchronization, b) use of static analysis for enhanced safety and faster run-time execution, and c) syntactic interface and semantics similar to the widely applied and supported containers of the programming language used for the system implementation. As our experience with MDS demonstrates, shared data can often be polymorphic. To allow for the direct representation of the polymorphic behavior of such data in MDS, we describe and validate an improved constant-time dynamic cast operation. Such an approach achieves safe real-time application and low cost of the operation at the expense of some extra work performed by the compiler's static checker.

1. Semantically Enhanced Containers

To achieve *higher safety* and *faster performance*, we define the notion and propose the application of *Semantically Enhanced Containers* (SEC) for *lock-free synchroniza-*

tion. As defined by Herlihy [3], a concurrent object is *nonblocking* if it guarantees that *some* process in the system will make progress in a *finite* amount of steps. Nonblocking algorithms do not apply mutually exclusive locks and instead rely on a set of atomic primitives supported by the hardware architecture. The most ubiquitous and versatile data structure in the ISO C++ Standard Template Library [4] is *vector*, offering a combination of dynamic memory management and constant-time random access. A number of pivotal concurrent applications in the Mission Data System framework employ a shared STL vector protected by mutually exclusive locks, such as the Data Management Service containers [5], the Goal Checker — an application for monitoring the status of goals, and Elf — a framework for message passing and transportation. In this work we present and utilize the design of the first lock-free implementation of a SEC dynamically-resizable array in ISO C++. It provides fast linearizable operations, disjoint-access parallelism for random access reads and writes, lock-free memory allocation and management, and fast execution. To allow the validation of domain-specific concerns in SEC and achieve faster run-time execution, we utilize the Pivot framework for C++ program representation and static analysis [6] and introduce Basic Query (BQ), an expression-template based library for the definition of static queries. While eliminating the hazards associated with the application of locks, nonblocking programming techniques introduce a safety hazard on their own: the ABA problem [7], [8]. Our SEC approach directly addresses the ABA problem and offers a number of practical techniques for its avoidance. In addition, we define a generic condition for ABA safety, called the $\lambda\delta$ approach, that allows the elimination of the ABA hazard in a time and space efficient manner and with no reliance on complex atomic primitives. In an object-oriented design, the application of dynamic cast provides flexibility in the use of data management facilities. The traditional compiler implementations of dynamic cast do not provide the timing guarantees needed for hard real-time embedded systems. Because of the dynamic cast's important role in the MDS Data Management Services, we explore the model-based semantic enhancement of the dynamic cast operation that allows for its application in embedded autonomous space systems.

2. Verification and Semantic Parallelization of Real-Time C++ in the Mission Data System Platform

We rely on the notion of Semantically Enhanced Containers to design and implement a methodology for verification and semantic parallelization of real-time C++. Our notion of *semantic parallelization* implies the thread-safe concurrent execution of system algorithms that utilize shared data, based on the application's semantics and invariants. As a practical industrial-scale application, we demonstrate the parallelization and verification of the MDS Temporal Constraint Networks. A Temporal Constraint Network (TCN) defines the goal-oriented operation of a control system. The Temporal Constraint Networks Library is at the core of the Jet Propulsion Laboratory's Mission Data System (MDS) [9] state- and goal-oriented unified architecture for testing and development of mission software. The MDS framework and its associated system engineering processes and development tools have been successfully applied on a number of test platforms including the physical rovers Rocky 7 and Rocky 8 and a simulated Entry, Descent, and Landing (EDL) module for the Mars Science Laboratory mission.

B. Challenges for Mission Critical Autonomous Software

The dominant paradigms for software development, assurance, and management of autonomous flight applications rely on the principle "test-what-you-fly and fly-what-you-test." This methodology has been applied in a large number of robotic space missions at the Jet Propulsion Laboratory. For such missions, it has proven suitable in achieving adherence to some of the most stringent standards of man-rated certification, such as DO-178B [10], the Federal Aviation Administration's (FAA) software standard. Its Level A certification requirements demand 100% coverage of all high- and low-level assurance policies. Some future space exploration projects such as Mars Science Laboratory (MSL) [11], Project Constellation [12], and the development of the Crew Launch Vehicle (CLV) and the Crew Exploration Vehicle (CEV) [13] suggest the engineering of some of the most complex man-rated software systems. As stated in the Columbia Accident Investigation Board's Report [14], the inability to thoroughly apply the required certification protocols had been determined to be a contributing factor to the loss of STS-107, Space Shuttle Columbia. Schumann and Visser's discussion in [15] suggests that the current certification methodologies are prohibitively expensive for systems of such complexity. A detailed analysis by Lowry [2] indicates that at the present moment the certification cost of mission-critical space software exceeds its development cost. The challenges of certifying and re-certifying avionics software has led NASA to initiate a number of advanced experimental software development and testing platforms, such as Mission Data System (MDS) [9], as well as a number of program synthesis, modeling, analysis, and verification techniques and tools, such as JavaPathFinder [16], CLARAty project [17], Project Golden Gate [18], and New Millenium Architecture Prototype (NewMAAP) [19]. The high cost and demands of man-rated certification have motivated the experimental development of several accelerated testing platforms [20].

Perrow [21] studies the risk factors in the modern high technology systems. His work identifies two significant sources of complexity in modern systems: *interactions* and *coupling*. The systems most prone to accidents are those with *complex* interactions and *tight* coupling. With the increase of the size of a system, the number of functions it has to serve, as well as its interdependence with other systems, its interactions become more incomprehensible to human and machine analysis and this can cause unexpected and anomalous behavior. Tight coupling is defined by the presence of time-dependent processes, strict resource constraints, and little or no possible variance in the execution sequence. Perrow classifies space missions in the riskiest category since both hazard factors are present. The notions of *concurrency* and *time* are the most critical elements in the design and implementation of an embedded autonomous space system. According to a study on concurrent models of computation for embedded software by Lee and Neuendorffer [22], the major contributing factors to the development and design's complexity of cyber-physical systems are the underlying sequential memory models and the lack of first class representation of the notions of time and concurrency in the applied programming languages.

C. Parallelism and Complexity

ISO C++ [23] is widely used for parallel and multi-threaded software, despite the fact that the C++ Standard currently does not mention concurrency or thread-safety. In a parallel application, there are a number of challenges that are not known in sequential programming: most importantly to correctly manipulate data where multiple threads access it. The most commonly applied technique for controlling the interactions of concurrent processes is the application of mutual exclusion locks. A mutual exclusion lock guarantees thread-safety of a concurrent object by blocking all contending threads except the one holding the lock. This can seriously affect the performance of the system and diminish its parallelism. For the majority of applications, the problem with locks is one of difficulty of providing correctness more than one of performance.

The application of mutual exclusion locks poses significant safety hazards (such as livelock, deadlock, priority inversion, and convoying [24]) and incurs high complexity in the testing and validation of mission-critical software. Mutual exclusion locks can be optimized in some scenarios by utilizing fine-grained locks [25] or context-switching. Often due to the resource limitations of flight-qualified hardware, optimized lock mechanisms are not a desirable alternative [2]. Even for efficient locks, the interdependence of processes implied by the use of locks, introduces the dangers of deadlock, livelock, and priority inversion. The incorrect application of locks is hard to detect with the traditional testing procedures and a program can be deployed and used for a long period of time before the flaws trigger anomalous behavior [2].

1. The Mars Pathfinder Mission

As discussed by Lowry [2], in July 1997 The Mars Pathfinder mission experienced a number of anomalous system resets that caused operational delay and loss of scientific data. The follow-up study identified the presence of a priority inversion problem caused by the low-priority meteorological process blocking the high-priority bus management process. The investigation furthermore revealed that it would have been impossible to detect the problem with the black box testing applied at the time to derive the certification artifacts. A safer priority inversion inheritance algorithm had been ignored due to its frequency of execution, the real-time requirements imposed, and its high cost incurred on the slower flight-qualified computer hardware.

The subtle interactions in the concurrent applications of the modern aerospace autonomous systems are of critical importance to the system's safety and correct operation. Despite the challenges in debugging and verification of the system's concurrent components, the existing certification process [10] does not provide guidelines at the level of detail reaching the development, application, and testing of concurrent programs. This is largely due to the process-oriented nature of the current certification protocols and the complexity and high level of specialization of the aerospace autonomous embedded applications.

In the near future, NASA plans to deploy a number of diverse vehicles, habitats, and supporting facilities for its imminent missions to the Moon, Mars and beyond. The large array of complex tasks that these systems would have to perform implies their high level of autonomy. Rasmussen et al. [9] present the challenges for these systems' control as one of the most demanding tasks facing NASA's Exploration Systems Mission Directorate. Some of the most significant challenges that the authors identify are managing a large number of tightly-coupled components, performing operations in uncertain remote environments, ability to respond and recover from anomalies, guaranteeing the system's correctness and reliability, and the effective communication across the system's components.

D. Overview and Contribution

Here, we summarize the contributions of this dissertation and provide an overview of the remaining chapters:

- Chapter II presents in detail the theoretical background and the technical terms that lay at the foundation of this work. The chapter discusses *a.* the concept and requirements of Temporal Constraint Networks (TCN) and their application in the Mission Data System Platform, *b.* the foundations of nonblocking synchronization, and *c.* the challenges in the design and implementation of lock-free containers and in particular a dynamically resizable array.

- In Chapter III we describe our design and implementation of the first lock-free dynamically resizable array. Our approach offers a set of practical, portable, lock-free, and linearizable STL vector operations and a fast and space efficient implementation that incorporates nonblocking memory management and allocation libraries. The chapter presents nonblocking algorithms defining the following vector's operations: push_back, pop_back, reserve, read, write, and size. The chapter further analyzes the concurrent semantics of the operations and the implementation's correctness. Our performance analysis contrasts our lock-free vector implementation with *a.* the concurrent vector provided by Intel [25] and *b.* an STL vector protected by a lock. The chapter discusses the following set of experiments:

1. performance evaluation on a shared memory system: our experimental data shows that under contention the lock-free vector outperforms the alternative lock-based approaches by a factor of 10 or more,
2. evaluation of the vector's performance using two different garbage collection approaches,
3. performance evaluation on a system without shared L2 cache: the results demonstrate that in such systems the lock-free approach offers performance comparable to that of the best available lock-based alternatives.

Our performance analysis concludes that the presented implementation is portable, practical, fast, and space-efficient. Using the current implementation, a user has to avoid one particular ABA problem.

- In Chapter IV we study the application of the state-of-the-art nonblocking Software Transactional Memory libraries for the design of nonblocking containers. We demonstrate the use of the Rochester Software Transactional Memory (RSTM) [26] library for the construction of a nonblocking shared vector. Our

RSTM-based vector provides algorithms that implement the following basic vector operations: push_back, pop_back, read, and write. In our performance analysis we compare: *a.* the RSTM-based nonblocking vector, *b.* a variation of the RSTM-based vector using lock-based transactions, and *c.* our hand-crafted CAS-based design as presented in Chapter III. Our performance evaluation suggests that while hard to design and implement, CAS-based algorithms provide fast and scalable performance and outperform the nonblocking STM-based alternatives by a significant factor.

- In Chapter V we introduce the principles of our product-oriented certification framework founded on the concept of source code enhancement and analysis. The chapter offers a description of our classification of the certification artifact types, the development and validation tools and techniques used to implement a cyber-physical system, the application's domain-specific factors, and the levels of abstraction in the system's design and implementation.

- In Chapter VI we introduce the concept of Semantically Enhanced Containers (SEC). We restrict the notion of a SEC to a container that meets three core criteria: a) built-in safe concurrent synchronization suitable for real-time embedded applications, b) use of static analysis for enhanced safety such as the elimination of the ABA problem, and c) syntactic interface and semantics similar to the widely applied and supported ISO C++ STL containers. In Chapter VI we present a SEC vector engineered to ensure *safe and efficient concurrent synchronization* as well as offer the mechanisms to establish the validity of certain *user-defined semantic guarantees.* The chapter discusses our application of static analysis and the Pivot framework [6] that help us achieve more efficient run-time execution of the container's operations (when compared to the use of dynamic checks and garbage collection). The chapter further explains the design and application of Basic Query (BQ), our expression-template based library for extracting semantic information from C++ source code. BQ defines the programming techniques for specifying and statically checking domain-specific properties in code. We apply BQ to avoid the ABA problem in our lock-free vector implementation from Chapter III. According to our performance analysis the SEC approach offers a cost-effiective and flexible approach for the prevention of ABA hazards with only mild limitations on the uses of the lock-free vector.

- In Chapter VII we present a generic and practical solution to the ABA problem, called the $\lambda\delta$ approach, that can easily be adopted in any Descriptor-based lock-free design. Currently the literature *does not offer an explicit analysis of the ABA problem*, its relation to the most commonly applied nonblocking programming techniques, and the possibilities for its detection and avoidance. At the present moment of time, eliminating the hazards of ABA in a nonblocking algorithm is left to the ingenuity of the software designer. In Chapter VII we study in detail and define the conditions that lead to ABA. We investigate the relationship between the ABA hazards and the most commonly applied nonblocking programming techniques and correctness guarantees. Our performance evaluation establishes that the single word CAS-based $\lambda\delta$ approach delivers performance comparable to the use of the architecture-specific CAS2 [27] and offers considerable performance gains when compared to the use of garbage collection.

- In Chapter VIII we present the design, implementation, and practical application of our framework for verification and semantic parallelization of real-time C++ within JPL's MDS framework. The nonblocking synchronizations techniques discussed in Chapters III, VI, and VII play a central role for the realization of our framework. The end goal of the framework is, given the implementation of the optimized iterative propagation scheme and the topology of a particular goal network, to establish the correctness of the core TCN semantic invariants and *automatically* derive a C++ implementation that can be executed concurrently on one of JPL's experimental testbeds for accelerated testing. We describe the architectural principles of the Mission Data System Platform in the context of our product-oriented certification [28] framework. Furthermore, the chapter presents an optimized algorithm for constraint propagation and proceeds by discussing our approach for modeling, formal verification, and automatic parallelization of the TCN propagation scheme. We show the Alloy formal models and the certification invariants applied. In addition, we use the certification framework introduced in Chapter V to analyze the process of model-based development of the parallel autonomic goals network. We identify seven critical certification artifacts in the process of model-driven development and validation of the MDS goal network. In the analysis of this process, we establish the relationship among the seven certification artifacts, the applied development and validation techniques and tools, and the levels of abstraction of system design and development. The analysis and performance evaluation of our approach show that the use of nonblocking synchronization is of significant importance in achieving reliability, efficiency, and better scalability in our parallel propagation algorithm.

- In Chapter IX we apply the principles of model-based analysis and certification and source code enhancement to the use of the C++ dynamic cast operation in MDS. The application of dynamic cast is considered hazardous for embedded real-time software due to the lack of constant time reply in its standard implementation. However, the dynamic cast allows flexibility in the design and use of data management facilities in object-oriented programs and has an important role in the implementation of the MDS Data Management Services Library. In Chapter IX we define and apply a co-simulation framework based on the SPIN model checker to formally verify and evaluate the G&S fast dynamic casting approach [29], an implementation strategy that guarantees fast constant-time execution of the dynamic cast operation. In the G&S scheme, a heuristic algorithm assigns an integer type ID at link time to each class. Our co-simulation framework consists of *a.* an abstract formal model of the G&S type ID assignment heuristics and *b.* a procedure for exhaustive search of the state space discovering the best type ID assignment. The chapter shows the pseudocode of our co-simulation approach and an algorithm for the discovery of a global minimum in the state space of the formal verification process. The analysis of the heuristics simulation performed in SPIN provides us with ideas of possible improvements to the G&S type ID assignment. The application of the co-simulation framework helped us implement optimizations to the G&S heuristics leading to the discovery of optimal type ID assignment in 85% of the class hierarchies, in contrast to 48% for the original G&S algorithm.

- Chapter X concludes this dissertation and provides directions for future research.

1. Overview of the Algorithms

Table 1 shows a list of the algorithms presented in this dissertation.

Table 1. List of Algorithms

Component	*Operation*	*Chapter*
Descriptor-based Lock-free Vector	push_back	Chapter III
Descriptor-based Lock-free Vector	pop_back	Chapter III
Descriptor-based Lock-free Vector	Allocate Memory Bucket	Chapter III
Descriptor-based Lock-free Vector	size	Chapter III
Descriptor-based Lock-free Vector	read	Chapter III
Descriptor-based Lock-free Vector	write	Chapter III
Descriptor-based Lock-free Vector	reserve	Chapter III
Descriptor-based Lock-free Vector	Complete Write	Chapter III
RSTM-based Vector	read	Chapter IV
RSTM-based Vector	write	Chapter IV
RSTM-based Vector	pop_back	Chapter IV
RSTM-based Vector	push_back	Chapter IV
Semantically Enhanced Containers	exclude push_back	Chapter VI
ABA-free Sync.	CAS-based speculation at L_i	Chapter VII
ABA-free Sync.	Two-step execution of a δ object	Chapter VII
ABA-free Sync.	Descriptor Object with obstruction-free semantics	Chapter VII
ABA-free Sync.	Implementing a $\lambda\delta$-modifying operation	Chapter VII
Automatic Parallelization Framework	TCN Propagation, Forward Pass	Chapter VIII
Automatic Parallelization Framework	TCN Propagation, Backward Pass	Chapter VIII
Automatic Parallelization Framework	Definition of Temporal Constraint and Time Point	Chapter VIII
Automatic Parallelization Framework	Definition of Time Phase and TP-based TCN	Chapter VIII
Automatic Parallelization Framework	Main TCN invariants	Chapter VIII
Automatic Parallelization Framework	Main TP-based TCN invariants	Chapter VIII
Fast Dynamic Cast	Co-simulation execution	Chapter IX
Fast Dynamic Cast	Finding the global minimum	Chapter IX

2. Overview of the Experiments

Table 2 presents a list of the core experiments executed and described in this dissertation.

Table 2. List of Experiments

Component	*Experiment*	*Chapter*
Descriptor-based Vector	Comp. w/ Lock-based Vector, Intel Core Duo	Ch. III, Figures 3, 4, 5, 6
Descriptor-based Vector	Comp. w/ Lock-based Vectorss, AMD 8-way Opteron	Ch. III, Figures 8, 9
Descriptor-based Vector	Comp. of Alternative Memory Management	Ch. III, Figure 7
RSTM-based Vector	Comp. w/ CAS-based vector	Ch. IV, Figures 10, 11, 12
SEC (Sem. Enh. Cont.)	SEC Performance Analysis	Ch. VI, Figure 14
ABA-free Sync.	Comp. w/ CAS2 and All-GC	Ch. VI, Figures 15, 16, 17, 18
Automatic Parallelization Frmk.	TCN Constraint Propagation	Ch. VIII, Figure 22
Fast Dynamic Cast	Co-simulation of the Seven Cases	Ch. IX, Table 16
Fast Dynamic Cast	Search Time for Type ID Assignment	Ch. IX, Figure 26

CHAPTER II

BACKGROUND AND PREVIOUS WORK

As opposed to process-oriented certification, the product-oriented methodology [1] relies on the application of safety concerns directly on implementation source code. The product-oriented approach is inherently more flexible by offering the developers the freedom to follow a variety of software development life-cycle paradigms. In addition, the certification authority itself has the ability to collect all required artifacts for the system's safety and quality assurance. Product-oriented certification has been approached by the application of a variety of formal verification [30], modeling ([31] and [32]), code synthesis [33], and static analysis techniques [34]. An example of a program synthesis technique is AutoFilter [35] that has been developed for the automatic generation of the safety-critical parts of flight software that estimates the position and altitude of the spacecraft. Since the correctness of the generated code is directly dependent on the correctness of the program synthesis tool, FAA regulates that such synthesis tools must meet the same certification criteria as the mission-critial software being generated. The significant implementation effort and the sophisticated design of such tools incur a prohibitive cost to the certification process in this approach. As demonstrated by Denney and Fisher's work [36], rewrite-based simplifications and other program transformations are often necessary in order to reduce the verification state space. In such methodologies, it is the developers' responsibility to certify and establish the fidelity of the formal models with respect to the source as well as the semantic derivation in the applied program transformations.

The rest of this chapter presents the background knowledge accumulated and needed to derive and apply the principles of our time and concurrency framework for product-oriented software certification.

A. Temporal Constraint Networks

A Temporal Constraint Network (TCN) defines the goal-oriented operation of a control system. The TCN library is at the core of the Jet Propulsion Laboratory's Mission Data System (MDS) [9] state-based and goal-oriented unified architecture for testing and development of mission software. A TCN consists of a set of temporal constraints (TCs) and a set of time points (TPs). In this model of goal-driven operation, a time point is defined as an interval of time when the configuration of the system is expected to satisfy a property predicate. The width of the interval corresponds to the temporal uncertainty inherent in the satisfaction of the predicate. Similarly, temporal constraints have an associated interval of time corresponding to the acceptable bounds on the interactions between the control system and the system under control during the performance of a specific activity. A TCN graph topology represents a snapshot at a given time of the known set of activities the control system has performed so far, is currently engaged in, and will be performing in the near future up to the horizon of the elaborate plan initially created as a solution for a set of goals. Figure 1 illustrates an example of a TCN topology with 14 time points. The topology of a temporal constraint network must satisfy a number of invariants.

(a) A TCN is a directed acyclic graph where the vertices represent the set of all time points ($\mathrm{S_{tps}}$) and the edges the set of all temporal constraints ($\mathrm{S_{tcs}}$).

(b) For each time point $\mathrm{TP_i} \in \mathrm{S_{tps}}$, there is a set of temporal constraints that are immediate successors ($\mathrm{S_{succ_i}}$) of $\mathrm{TP_i}$ and a set, $\mathrm{S_{pred_i}}$, consisting of all of $\mathrm{TP_i}$'s immediate predecessors.

(c) Each temporal constraint $\mathrm{TC_j} \in S_{tcs}$ has exactly one successor $\mathrm{TP_{succ_j}}$ and one predecessor $\mathrm{TP_{pred_j}}$.

(d) For each pair $\{\mathrm{TP_i}, \mathrm{TC_j}\}$, where $\mathrm{TP_i} \equiv \mathrm{TC_{succ_j}}$, $\mathrm{TC_j} \in \mathrm{S_{pred_i}}$ must hold. The reciprocal invariant must also be valid, namely for each pair of $\{\mathrm{TP_i}, \mathrm{TC_j}\}$ such that $\mathrm{TP_i} \equiv \mathrm{TC_{pred_j}}$, $\mathrm{TC_j} \in \mathrm{S_{succ_i}}$.

(e) The firing window of a time point $\mathrm{TP_i} \in \mathrm{S_{tps}}$ is represented by the pair of time instances $\{\mathrm{TP_{min_i}}, \mathrm{TP_{max_i}}\}$. Assuming that the current moment of time is represented by $\mathrm{T_{now}}$, then $\mathrm{TP_{min_i}} \leq \mathrm{T_{now}} \leq \mathrm{TP_{max_i}}$, for every $\mathrm{TP_i} \in \mathrm{S_{tps}}$.

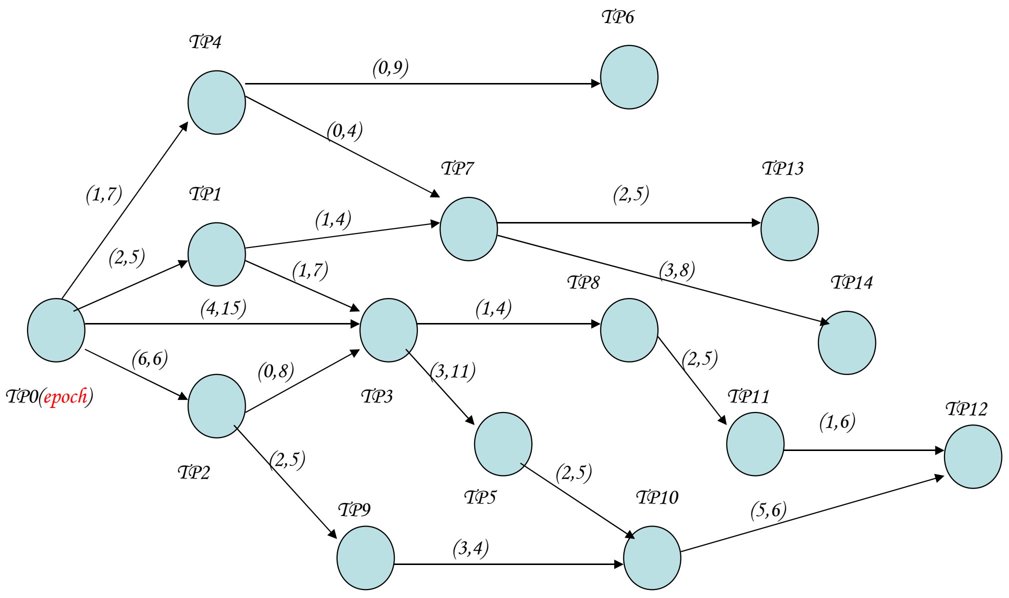


Fig. 1. An Example of a Temporal Constraint Network: A TCN Topology with 14 TPs

General-purpose programming languages lack the capabilities to formally specify and check domain-specific design constraints. Direct representation and verification of the TCN invariants in the implementation source code would likely result in a cumbersome and inefficient solution. However, any implementation (in C++, Java, or another programming language) must operate under the assumption that the basic TCN invariants are satisfied. Thus, prior to implementing a solution to the TCN constraint propagation problem, it is necessary to guarantee the correctness and consistency of the topology of the goal network. We further discuss these issues as well as demonstrate an approach for automatic semantic parallelization for accelerated testing of the TCN propagation approach in Chapter VIII.

B. Lock-Free Dynamically Resizable Arrays

In this section we examine the following topics:

(1) Briefly introduce the foundations of lock-free programming.

(2) Examine in details the challenges for the design and implementation of a concurrent dynamic array.

  (a) Discuss the possible consistency models and the assumed concurrent semantics.

  (b) Identify the most desirable characteristics of a nonblocking array, given the assumed semantics.

  (c) Analyze implementation issues related to:

    (-) ensuring portability,

    (-) meeting the requirements for linearizability,

    (-) coping with the ABA problem,

    (-) effectively incorporating nonblocking memory management techniques.

(3) Present a study of three state-of-the-art approaches for a nonblocking design of a concurrent dynamic array.

   (a) The hand-crafted approach: lock-free dynamically resizable arrays (as further discussed in Chapter III).

   (b) The Software Transactional Memory (STM) approach: the design based on the utilization of an STM library (Chapter IV).

   (c) Predictive Log Synchronization: a recent concept suggested by Shalev and Shavit [37].

1. Foundations of Lock-Free Programming

A concurrent object is *nonblocking* [8] if it guarantees that *some* process in the system will make progress in a *finite* number of steps. An object that guarantees that *each* process will make progress in a *finite* number of steps is defined as *wait-free.Obstruction-freedom* [38] is an alternative nonblocking condition that guarantees progress if a thread eventually executes in *isolation*. It is the weakest nonblocking property and obstruction-free objects require the support of a contention manager to prevent livelocking.

The lock-free, wait-free, and obstruction-free algorithms do not apply mutual exclusion locks. Instead, they rely on a set of atomic primitives such as the word-size CAS instruction [27]. Common CAS implementations [27], [8] require three arguments: a memory location, Mem, an old value, V_{old}, and a new value, V_{new}. The instruction atomically exchanges the value stored in Mem with V_{new}, provided that its current value equals V_{old}. The architecture ensures the atomicity of the operation by applying a fine-grained hardware lock such as a cache or a bus lock (e.g.: IA-32 [27]). The use of a hardware lock does not violate the nonblocking property as defined by Herlihy [8]. Common locking synchronization methods such as semaphores, mutexes, monitors, and critical sections utilize the same atomic primitives to manipulate a control token. Such applications of the atomic instructions introduce interdependencies of the contending processes. In the most common scenario, lock-free systems utilize CAS in order to implement a speculative manipulation of a shared object. Each contending process speculates by applying a set of writes on a local copy of the shared data and attempts to CAS the shared object with the updated copy (see Chapter III for further details on the application of CAS in nonblocking designs). This speculative execution guarantees that from within a set of contending processes, there is at least one that succeeds within a finite number of steps (thus the system is nonblocking). Linearizability [8] is an important correctness condition for concurrent objects: a concurrent operation is linearizable if it appears to execute instantaneously in a given point of time between the time τ_{inv} of its invocation and the time τ_{end} of its completion. The consistency model implied by the linearizability requirements is stronger than the widely applied Lamport's sequential consistency model [39]. According to Lamport's definition, sequential consistency requires that the results of a concurrent execution are equivalent to the results yielded by *some* sequential execution (given the fact that the operations performed by each individual processor appear in the sequential history in the order as defined by the program).

2. Practical Lock-Free Programming Techniques

The practical implementation of a hand-crafted lock-free container is notoriously difficult: in addition to addressing the hazards of race conditions, the developer must also find a way to incorporate nonblocking memory management and memory allocation schemes. As suggested by the authors in [40], [41], and [24], the use of only a single-word CAS operation makes the task of designing a practical non-trivial concurrent container very difficult and complex. A simpler and more efficient design of a nonblocking data container requires the atomic update of several memory locations. The use of a Double-Compare-And-Swap primitive (DCAS) has been suggest by Detlefs et al. [41], however such complex atomic operations are rarely supported by the hardware architecture.

Harris et al. describe [42] a software implementation of a multiple-compare-and-swap (MCAS) algorithm based on CAS. This software-based MCAS algorithm has been applied by Fraser in the implementation of a number of lock-free containers such as binary search trees and skip lists [43]. The cost of the MCAS operation is expensive requiring $2M + 1$ CAS instructions. Consequently, the direct application of the MCAS scheme is not an optimal approach for the design of lock-free algorithms. However, the MCAS implementation employs a number of techniques, such as pointer bit marking and the use of Barne's style announcements [44], that are useful for the design of practical lock-free systems. A Barne's style announcement is an object that allows an interrupting thread help an interrupted thread complete. The pointer bit marking technique exploits the last two bits of a pointer value, which are unused in a pointer representation, to store up to three additional binary states. Thus, a single CAS operation can atomically exchange the pointer and its state.

A number of advanced and recent Software Transactional Memory (STM) Libraries provide nonblocking transactions (typically obstruction-free) with linearizable operations [45]. Such transactions can be utilized for some designs of nonblocking containers. The high cost of the conflict detection and validation schemes in such systems would often not allow performance that is superior to that of a hand-crafted lock-free container which relies solely on the application of the portable atomic primitives. In addition, to prevent livelocking in an obstruction-free design, the implementor needs to apply a contention manager [38].

Predictive Log Synchronization (PLS) is an alternative paradigm suggested by Shalev and Shavit [37] that allows for simpler designs and less costly conflict detection and validation schemes. The core idea is to delegate all writes to a single thread that performs the data structure's modifications based on a log file protected by a single mutual exclusion lock. The design is nonblocking because once a thread finds the lock unavailable, it runs a speculative execution based on the description of the concurrent operations available in the log file. The presented approach is very recent and its implementation is still unavailable. PLS has been published simply as a proof-of-concept and thus has not been applied and extensively tested in the design of complex concurrent algorithms. The main drawbacks for its practical application are the weaker consistency model that it provides (Lamport's sequential consistency), its inefficiency in the scenario of an application performing a larger volume of concurrent writes, and the unbounded growth in the cost of its speculative routine in certain scenarios.

a. Lock-Free Data Containers

Recent research in the design of lock-free data structures includes linked-lists ([46] and [47]), double-ended queues ([48] and [49]), stacks [50], hash tables ([47] and [51]), and binary search trees [43]. The problems encountered include excessive copying, low parallelism, inefficiency, and high overhead. Despite the widespread use of the STL vector in real-world applications, we are aware of only one published work [7] that discusses the problem of the design and implementation of a lock-free dynamic array. The vector's random access, data locality, and dynamic memory management poses serious challenges for its nonblocking implementation.

3. Semantics

The semantics of a concurrent data container can be based on a number of assumptions. For the designs we study in this report, we assume that each processor can execute a number of the vector's operations. This establishes a *history* of invocations and responses and defines a *real-time order* between them. An operation O_1 is said to precede an operation O_2 if O_2*'s* invocation occurs after O_1*'s* response. Operations that do not have real-time ordering are defined as *concurrent.* A *sequential history* is one where all invocations have immediate responses. A *linearizable history* is one where:

(1) all invocations and responses can be reordered so that they are equivalent to a sequential history,

(2) the yielded sequential history must correspond to the semantic requirements of the sequential definition of the object,

(3) in case a given response precedes an invocation in the concurrent execution, then it must precede it in the derived sequential history.

It is the last requirement that differentiates the consistency model implied by the definition of linearizability with Lamport's sequential consistency model and makes linearizability stricter.

4. Design Goals

In this section we synthesize the most desirable characteristics of a shared nonblocking container:

(1) *thread-safety*: the data should be accessible to multiple processors at all times,

(2) *lock-freedom*: apply nonblocking techniques for the implementation,

(3) *portability*: do not rely on uncommon architecture-specific instructions,

(4) *easy-to-use interfaces*: offer the interfaces, functionality, and guarantees available in the sequential STL vector,

(5) *high level of parallelism*: concurrent completion of non-conflicting operations should be possible,

(6) *minimal overhead*: achieve lock-freedom without excessive copying, levels of indirection, and costly conflict detection and validation schemes, minimize the time spent on redundant and speculative computations and the number of calls to costly atomic primitives.

5. Implementation Concerns

We provide a brief summary of the most important implementation concerns for the practical and portable design of a nonblocking dynamic array. The following sections discuss the implementation issues related to guaranteeing portability, meeting the requirements for linearizability, preventing race conditions, coping with the ABA problem, and incorporating nonblocking memory management and allocation schemes.

a. Portability

Virtually at the core of every known synchronization technique is the application of a number of hardware atomic primitives. The semantics of such primitives varies depending on the specific hardware platform. There are a number of architectures that offer the support of some hardware atomic instructions that provide greater flexibility (compared to a single-word CAS) such as the Load-Link/Store Conditional (LL/SC) supported by the PowerPC, Alpha, MIPS, and the ARM architectures or instructions that perform atomic writes to more than a single word in memory, such as the Double-Compare-And-Swap (DCAS) instruction [41]. The hardware support for such atomic instructions can vastly simplify the design of a nonblocking algorithm as well as offer immediate solutions to a number of challenging problems such as the ABA problem [52]. However, to maintain portability across a large number of hardware platforms, the design and implementation of a nonblocking algorithm cannot rely on the support of such atomic primitives. The most common atomic primitive that is supported by a large majority of hardware platforms is the single-word Compare-And-Swap (CAS) instruction.

b. Linearizability Requirements

In a CAS-based design, a major difficulty is meeting the linearizability requirements for operations that require the update of more than a single-word in the system's shared memory. To cope with this problem, it is possible to apply a combination of a number of known techniques:

(1) *Extra Level of Indirection:* Reference semantics can be used in case that the data being manipulated is larger than a memory word size.

(2) *Descriptor Object:* A *Descriptor Object* (see Chapter III) stores a record of a pending operation on a given memory location. It allows the interrupting threads help the interrupted thread complete an operation rather than wait for its completion.

(3) *Descriptive Log:* The Descriptive Log methodology lies at the core of virtually all Software Transactional Memory implementations. A Descriptive Log stores a

record of all pending reads and writes to the shared data. It is used for conflict detection, validation, and optimistic speculation.

(4) *Transactional Memory:* A duplicate memory copy used to perform speculative updates that are invisible to all other threads until the linearization point of the entire transaction.

(5) *Optimisitic Speculation:* A complex nonblocking operation employs optimistic speculative execution in order to carry out the memory updates on a local or duplicate memory copy and commit once there are no conflicts with interfering operations.

To illustrate the complexity of a CAS-based design of a dynamically resizable array, Table 3 provides an overview of the number of shared memory locations that need to be updated upon the execution of some of its basic operations.

Table 3. STL Vector — Number of Memory Locations to be Updated per Operation

	Operations	Memory Locations
push_back	$Vector \times Elem \rightarrow$ void	*2: element and size*
pop_back	$Vector \rightarrow Elem$	*1: size*
reserve	$Vector \times size\_t \rightarrow Vector$	*n: all elements*
read	$Vector \times size\_t \rightarrow Elem$	*none*
write	$Vector \times size\_t \times Elem \rightarrow Vector$	*1: element*
size	$Vector \rightarrow size\_t$	*none*

c. Interfaces of the Concurrent Operations

According to the ISO C++ Standard [23], the STL containers' interfaces are inherently sequential. The next ISO C++ Standard [53] is going to include a concurrent memory model [54] and possibly a blocking threading library. In Table 4 we show a brief overview of some of the basic operations of an STL vector according to the current standard of the C++ programming language. Consider the sequence of operations applied to an instance, vec, of the STL vector: vec[vec.size()-1]; vec.pop_back();. In an environment with *concurrent* operations, we cannot have the guarantee that the element being deleted by thepop_back is going to be the element that had been read earlier by the invocation of operator[]. Such a *sequential* history is just one of the several legal sequential histories that can be derived from the concurrent execution of the above operations. While the STL interfaces have proven to be efficient and flexible for a large number of applications [4], to preserve the semantic behavior implied by the sequential definitions of STL, one can either rely on a library with atomic transactions [45], [26] or alternatively define concurrent STL interfaces adequate with respect to the applied consistency model. In the example we have shown, it might be appropriate to modify the interface of the pop_back operation and return the element being deleted instead of the void return type specified in STL. Such an implementa-

Table 4. Interfaces of STL Vector

Operation	Description
size_type size() const	Number of elements in the vector
size_type capacity() const	Number of available memory slots
void reserve(*size_type n*)	Allocation of memory with capacity n
bool empty() const	true when size = 0
T* operator[] (*size_type n*) const	returns the element at position n
T* front()	returns the first element
T* back()	returns the last element
void push_back(const T&)	inserts a new element at the tail
void pop_back()	removes the element at the tail
void resize(n, t = T())	modifies the tail, making size = n

tion efficiently combines two operations: reading the element to be removed from the container and removing the element. The ISO C++ implementation of pop_back() returns void so that the operation is optimal (and does not perform extra work) in the most general case: the deletion of the tail element. Should we prefer to keep the STL standard interface of void pop_back() in a concurrent implementation, the task of obtaining the value of the removed element in a concurrent nonblocking execution might be quite costly and difficult to implement. Based on the shared containers' usage, observing the possibilities for such combinations can deliver better usability and performance advantages in a nonblocking implementation. Other possibly beneficial combinations of operations are 1) CAS-based read-modify-write at location L_i that unifies a random access read and write at location L_i and 2) the push_back of a block of tail elements.

d. The ABA Problem

The ABA problem [52] is fundamental to all CAS-based systems. A universal solution to the ABA problem is to associate a version counter to each element on platforms supporting Double-Compare-And-Swap or alternatively provide Load-Link/Store-Conditional (LL/SC) semantics. We cannot assume availability of these atomic primitives since they are specific to a limited number of hardware platforms.

There are two particular instances when the ABA problem can affect the correctness of the vector's operations:

(1) The user intends to store a memory address value A multiple times.

(2) The memory allocator reuses the address of an already freed object.

To eliminate the ABA problem of (2) (in the absence of CAS2 [27] or LL/SC), it is possible to incorporate a memory management scheme such as Herlihy et al.'s *Pass The Buck* algorithm [55] that utilizes a separate thread to periodically reclaim unguarded objects. The vector's vulnerability to (1) (in the absence of CAS2 or LL/SC), can be eliminated by requiring the data structure to copy all elements and store pointers to them. Such behavior complies with the STL value-semantics [4], however it can incur significant overhead in some cases due to the additional heap allocation and object construction. In a lock-free system, both the object construction and heap allocation can execute concurrently with other operations.

e. Memory Allocation and Management

A nonblocking algorithm needs to be able to acquire and safely release memory in an efficient, nonblocking manner. A garbage collected environment could significantly reduce the complexity of the implementation (by moving key implementation problems inside the GC implementation). However, we do not know of any available general lock-free garbage collector for C++ or Java.

Object Reclamation: it is possible to incorporate a reference counting technique as described by Michael and Scott [56]. The major drawback of such a scheme is that a timing window allows objects to be reclaimed while a different thread is about to increase the counter. Consequently, objects cannot be freed but only recycled. Alternatives such as Michael's Hazard Pointers [52] and Herlihy et al.'s Pass The Buck [55] overcome this problem.

Allocator: recent research by Michael [57] and Gidenstam [58] presents implementations of practical lock-free memory allocators.

CHAPTER III

LOCK-FREE VECTOR: DESIGN AND IMPLEMENTATION

In this chapter we provide an overview of our design and implementation of the first lock-free dynamically resizable array. The presented approach is based on a single-word atomic Compare-And-Swap (CAS) instruction. It provides a linearizable and highly parallelizable STL-like interface, lock-free memory allocation and management, and fast execution. Experiments on a dual-core Intel processor with shared L2 cache indicate that our lock-free vector outperforms its lock-based STL counterpart and the latest concurrent vector implementation provided by Intel by a large factor. The performance evaluation on a quad dual-core AMD system with non-shared L2 cache demonstrated timing results comparable to the best available lock-based techniques. The presented design implements the most common STL vector's interfaces, namely random access read and write, tail insertion and deletion, pre-allocation of memory, and query of the container's size. Using the current implementation, a user has to avoid one particular ABA problem. The lock-free vector's design and implementation provided follow the syntax and semantics of the ISO STL vector as defined in ISO C++ [23].

In the following sections we define a semantic model of the vector's operations, provide a description of the design and the applied implementation techniques, outline a correctness proof of the vector's lock-free semantics based on the adopted semantic model, address concerns related to memory management, and discuss some alternative solutions. The presented algorithms have been implemented in ISO C++ and designed for execution on an ordinary multi-threaded shared-memory system supporting only single-word read, write, and CAS instructions.

The major challenges of providing a lock-free vector implementation stem from the fact that key operations need to atomically modify two or more non-colocated words. For example, the critical vector operation push_back increases the size of the vector and stores the new element. Moreover, capacity-modifying operations such as reserve and push_back potentially allocate new storage and relocate all elements in case of a dynamic table [59] implementation. Element relocation must not block concurrent operations (such as write and push_back) and must guarantee that interfering updates will not compromise data consistency. Therefore, an update operation needs to modify up to four shared memory locations: size, capacity, storage, and a vector's element.

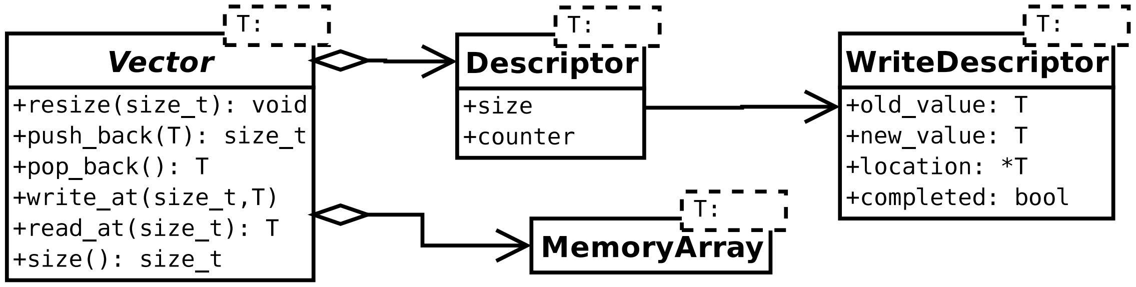


Fig. 2. Lock-free Shared Vector: UML Class Diagram

The UML diagram in Figure 2 presents the collaborating classes and their programming interfaces and data members. Each vector object contains the memory locations of the data storage of its elements as well as an object named Descriptor that encapsulates the container's size, a reference counter required by the applied memory management scheme (Section C) and an optional reference to a Write Descriptor. Our approach requires that data types bigger than word size are indirectly stored through pointers. Like Intel's concurrent vector [25], our implementation avoids storage relocation and its synchronization hazards by utilizing a two-level array. Whenever push_back exceeds the current capacity, a new memory block twice the size of the previous one is added.

The semantics of the pop_back and push_back operations are guaranteed by the Descriptor object. The use of a Descriptor and Write Descriptor allows a thread-safe update of two memory locations thus eliminating the need for a DCAS instruction. An interrupting thread intending to change the descriptor needs to complete any pending operation. Not counting memory management overhead, push_back executes *two successful CAS* instructions to update *two memory locations*.

A. Operations

Table 5 illustrates the implemented operations as well as their signatures, descriptor modifications, and runtime guarantees.

Table 5. Shared Vector - Operations Description and Complexity

	Operations	Descriptor (Desc)	Complexity
push_back	$Vector \times Elem \rightarrow$ void	$\text{Desc}_t \rightarrow \text{Desc}_{t+1}$	$O(1) \times congestion$
pop_back	$Vector \rightarrow Elem$	$\text{Desc}_t \rightarrow \text{Desc}_{t+1}$	$O(1) \times congestion$
reserve	$Vector \times size\_t \rightarrow Vector$	$\text{Desc}_t \rightarrow \text{Desc}_t$	$O(1)$
read	$Vector \times size\_t \rightarrow Elem$	$\text{Desc}_t \rightarrow \text{Desc}_t$	$O(1)$
write	$Vector \times size\_t \times Elem \rightarrow Vector$	$\text{Desc}_t \rightarrow \text{Desc}_t$	$O(1)$
size	$Vector \rightarrow size\_t$	$\text{Desc}_t \rightarrow \text{Desc}_t$	$O(1)$

The remaining part of this section presents the generalized pseudo-code of the implementation. It omits code necessary for a particular memory management scheme. We use the symbols ^, &, and . to indicate pointer dereferencing, obtaining an object's address, and integrated pointer dereferencing and field access respectively. The function HighestBit returns the bit-number of the highest bit that is set in an integer value. On modern x86 architectures HighestBit corresponds to the BSR assembly instruction [27]. FBS is a constant representing the size of the first bucket and equals eight in our implementation.

Push_back (add one element to the end): The first step is to complete a pending operation that the current descriptor might hold. In case that the storage capacity has reached its limit, new memory is allocated for the next memory bucket. Then, push_back defines a new Descriptor object and announces the current write operation. Finally, push_back uses CAS to swap the previous Descriptor object with the new one. Should CAS fail, the routine is re-executed. After succeeding, push_back finishes by writing the element.

Pop_back (remove one element from end): Unlike push_back, pop_back does not utilize a Write Descriptor. The pop_back operation completes any pending operation of the current descriptor, reads the last element, defines a new descriptor, and attempts a CAS on the Descriptor object.

Non-bound checking read and write at position i: The random access read and write do not utilize the descriptor and their success is independent of the descriptor's value.

Reserve (increase allocated space): In the case of concurrently executing reserve operations, only one succeeds per bucket, while the others deallocate the acquired memory.

Size (read number of elements): The `size` operation returns the size stored in the descriptor minus a potential pending write operation at the end of the vector.

Algorithm 1 push_back *vector*, *elem*

```
1: repeat
2:     $\mathsf{desc}_{\mathsf{current}} \leftarrow$ vector.desc
3:     CompleteWrite(vector, $\mathsf{desc}_{\mathsf{current}}$.pending)
4:     bucket $\leftarrow$ HighestBit($\mathsf{desc}_{\mathsf{current}}$.size + FBS) $-$ HighestBit(FBS)
5:     if vector.memory[bucket] = NULL then
6:         AllocBucket(vector, bucket)
7:     writeop $\leftarrow$ new WriteDesc(At($\mathsf{desc}_{\mathsf{current}}$.size), elem, $\mathsf{desc}_{\mathsf{current}}$.size)
8:     $\mathsf{desc}_{\mathsf{next}} \leftarrow$ new Descriptor($\mathsf{desc}_{\mathsf{current}}$.size+1, writeop)
9: until CAS(&vector.desc, $\mathsf{desc}_{\mathsf{current}}$, $\mathsf{desc}_{\mathsf{next}}$)
10: CompleteWrite(vector, $\mathsf{desc}_{\mathsf{next}}$.pending)
```

Algorithm 2 AllocBucket *vector*, *bucket*

```
1: bucketsize $\leftarrow \mathsf{FBS}^{\mathsf{bucket}+1}$
2: mem $\leftarrow$ new T[bucketsize]
3: if not CAS(&vector.memory[bucket], NULL, mem) then
4:     Free(mem)
```

B. Semantics

The vector's operations are of two types: those whose progress depends on the vector's descriptor and those who are independent of it. We refer to the former as *descriptor-modifying* and to the latter as *non-descriptor modifying operations*. All of the vector's operations in the set of concurrent descriptor-modifying operations S_1 are thread-safe and lock-free. The non-descriptor modifying operations such as random access read and write are implemented through the direct application of atomic read and write instructions on the shared data. In the set of non-descriptor modifying operations S_2, all operations are thread-safe and wait-free.

Algorithm 3 size *vector*

```
1: desc ← vector.desc
2: size ← desc.size
3: if desc.writeop.pending then
4:     size ← size − 1
5: return size
```

Algorithm 4 read *vector, i*

```
1: return At(vector, i)^
```

Correctness: The main correctness requirement of the semantics of the shared vector's operations is linearizability [60]. A concurrent operation is linearizable if it appears to execute instantaneously in some moment of time between the time point τ_{inv} of its invocation and the time point τ_{end} of its response. Firstly, this definition implies that each concurrent history yields responses that are equivalent to the responses of some legal sequential history for the same requests. Secondly, the order of the operations within the sequential history must be consistent with the real-time order. Let us assume that there is an operation $o_i \in \mathrm{S}_{\text{vec}}$, where S_{vec} is the set of all the vector's operations. We assume that o_i can be executed concurrently with n other operations $\{o_1, o_2..., o_n\} \in \mathrm{S}_{\text{vec}}$. We outline a proof that operation o_i is linearizable.

Linearization Points: For all non-descriptor-modifying operations the linearization point is at the time instance τ_{a} when the atomic read (Algorithm 4, line 1) or write (Algorithm 5, line 1) of the element is executed. Assume o_i is a descriptor-modifying operation. It is carried out in two stages: modify the Descriptor object and then update the data structure's contents. Let time points τ_{desc} (Algorithm 1, line 10; Algorithm 6, line 6) and $\tau_{\text{writedesc}}$ (Algorithm 9, line 2) denote the instances

Algorithm 5 write $vector, i, elem$

```
1: At(vector, i)^ ← elem
```

Algorithm 6 pop_back $vector$

```
1: repeat
2:     $\text{desc}_\text{current} \leftarrow$ vector.desc
3:     CompleteWrite(vector, $\text{desc}_\text{current}$.pending)
4:     elem $\leftarrow$ At(vector, $\text{desc}_\text{current}$.size $- 1)$^
5:     $\text{desc}_\text{next} \leftarrow$ new Descriptor($\text{desc}_\text{current}$.size $- 1$, NULL)
6: until CAS(& vector.desc, $\text{desc}_\text{current}$, $\text{desc}_\text{next}$)
7: return elem
```

of time when o_i executes an atomic update to the vector's Descriptor object and when o_i's Write Descriptor is completed by o_i itself or another concurrent operation $o_c \in \{o_1, o_2 ..., o_n\}$, respectively. Similarly, time point τ_{readelem} (Algorithm 1, line 7; Algorithm 6, line 4) defines when o_i reads an element. o_i is either a pop_back or push_back operation. The linearization point is either τ_{readelem} or τ_{desc} for the former case and τ_{readelem}, τ_{desc}, or $\tau_{\text{writedesc}}$ for the latter case.

Sequential Semantics: Let $\mathrm{S_c}$ be the set of all concurrent operations $\{o_1, ..., o_n\}$ in a time interval $[\tau_\alpha, \tau_\beta]$. If $\forall o_i \in \mathrm{S_c}$, DescriptorModifying($o_i$), the linearization point for each operation is $\tau_{\text{desc}}(o_i)$. Similarly, if $\forall o_i \in S_c$, NonDescriptorModifying(o_i), the linearization point for each operation is $\tau_a(o_i)$. In these cases, the resulting sequential histories are directly derived from the temporal order of the linearization points. In the remaining cases, the derivation of a sequential history is significantly more complex. It is possible to transform all non-descriptor modifying operations into descriptor modifying in order to simplify the vector's sequential semantics. Given our current implementation, this can be achieved in a straightforward manner. We

Algorithm 7 reserve $vector, size$

```
1: $i \leftarrow$ HighestBit(vector.desc.size + FBS -1)-HighestBit(FBS)
2: if $i < 0$ then
3:     $i \leftarrow 0$
4: while $i <$ HighestBit(size + FBS $- 1$) $-$ HighestBit(FBS) do
5:     $i \leftarrow i + 1$
6:     AllocBucket(vector, i)
```

Algorithm 8 At $vector, i$

```
1: pos $\leftarrow i +$ FBS
2: hibit $\leftarrow$ HighestBit(pos)
3: idx $\leftarrow$ pos xor $2^{\text{hibit}}$
4: return &vector.memory[hibit $-$ HighestBit(FBS)][idx]
```

have chosen not to do so in order to preserve the efficiency and wait-freedom of the current non-descriptor modifying operations. Table 6 determines the linearization points for each pair of concurrent operations (o_1, o_2) where DescriptorModifying(o_1) and NonDescriptorModifying(o_2).

Table 6. Linearization Points of o_1, o_2

$o_1 \backslash o_2$	read	write
push_back	$\tau_{\text{writedesc}}(o_1), \tau_a(o_2)$	$\tau_{\text{readelem}}(o_1), \tau_a(o_2)$
pop_back	$\tau_{\text{desc}}(o_1), \tau_a(o_2)$	$\tau_{\text{readelem}}(o_1), \tau_a(o_2)$

We emphasize that the presented ordering relations are not transitive. Consider an example with three operations o_1 (push_back), o_2 (write), and o_3 (read), which access the same element. We assume that time points $\tau_a(o_2), \tau_a(o_3)$ occur between

Algorithm 9 CompleteWrite $vector, writeop$

```
1: if writeop.pending then
2:     CAS(At(vector, writeop.pos), writeop.value$_{old}$, writeop.value$_{new}$)
3:     writeop.pending ← false
```

$\tau_{\text{readelem}}(o_1)$ and $\tau_{\text{writedesc}}(o_1)$ as well as that o_2 returns before the invocation of o_3. The resulting sequential history is o_1, o_2, o_3. It is derived from the real-time ordering between o_2 and o_3, and the pair-wise ordering relation between push_back and write in Table 6. A thorough linearizability proof for even the simplest data structure is non trivial and a further detailed elaboration is beyond the scope of this presentation.

Nonblocking: We prove the nonblocking property of our implementation by showing that out of n threads at least one makes progress. Since the progress of non-descriptor modifying operations is independent of all other concurrent operations, they are wait-free. Thus, it suffices to consider an operation o_1, where o_1 is either a push_back or pop_back. A Write Descriptor can be simultaneously read by n threads. While one of them will successfully perform the Write Descriptor's operation (o_2), the others will fail and not attempt it again. This failure is insignificant for the outcome of operation o_1. The first thread attempting to change the descriptor will succeed, which guarantees the progress of the system.

C. Memory Management

Our algorithms do not require the use of a particular memory management scheme. A garbage collected environment would have significantly reduced the complexity of the implementation (by moving key implementation problems inside the GC implementation). However, we do not know of any available general lock-free garbage collector for C++. Our concrete implementation uses reference counting as described by Michael and Scott [56]. Recent research by Michael [57] and Gidenstam [58] presents implementations of true lock-free memory allocators. Due to its availability and performance, we selected Gidenstam's allocator for our performance tests.

D. The ABA Problem and the Shared Vector

We have outlined the ABA hazards in the design of a lock-free vector in Chapter II, Section 5. In Chapters VI and VII we suggest new techniques for ABA prevention that can be applied to our lock-free vector.

E. Alternatives

In this section we discuss several alternative designs for lock-free vectors.

Copy on Write: Alexandrescu and Michael present a lock-free map, where every write operation creates a clone of the original map, which insulates modifications from concurrent operations [40]. Once completed, the pointer to the map's representation is redirected from the original to the new map. The same idea could be adopted to implement a vector. Since the complexity of any write operation deteriorates to $O(n)$ instead of $O(1)$, this scheme would be limited to applications exhibiting read-often-write-rarely access patterns.

Using Software DCAS: Harris et al. present a software Multi-Compare-And-Swap (MCAS) implementation based on a single-word CAS [42]. While convenient, the MCAS operation is expensive (requiring $2M + 1$ CAS instructions). Thus, it is not the best choice for an effective implementation.

Contiguous Storage: Techniques similar to the ones used in our vector implementation could be applied to achieve a vector with contiguous storage. The difference is that the storage area of the entire data structure can change over time.

This requires resize to move all elements to a new location. Hence, storage and its capacity should become members of the descriptor. Synchronization between write and resize operations is what makes this approach difficult. A straightforward solution is to apply descriptor-modifying semantics for all of the vector's operations as discussed in Section B.

We discussed the descriptor- and non-descriptor modifying writes in the context of the two-level array and the contiguous storage vector. However, these write properties are not inherent in these two approaches. In the two-level array, it is possible to make each write operation descriptor-modifying, thus ensure a write within bounds. In the contiguous storage approach, we could use pointer marking and element relocation could replace the elements with marked pointers to the new location. Every access to these marked pointers would get redirected to the new storage.

F. Performance Evaluation

We ran performance tests on an Intel IA-32 SMP machine with two 1.83GHz processor cores with 512 MB shared memory and 2 MB L2 shared cache running the MAC OS X operating system. In our performance analysis, we compare the lock-free approach (with its integrated lock-free memory management and memory allocation) with the concurrent vector provided by Intel [25] as well as an STL vector protected by a lock. For the latter scenario we applied different types of locking synchronizations — an operating system dependent mutex, a reader/writer lock, a spin lock, as well as a queuing lock. We used this variety of lock-based techniques to contrast our nonblocking implementation to the best available locking synchronization technique for a given distribution of operations. We utilize the locking synchronization provided by Intel [25].

Similarly to the evaluation of other lock-free concurrent containers [43] and [47], we have designed our experiments by generating a workload of various operations (push_back, pop_back, random access write, and read). In the experiments, we varied the number of threads, starting from 1 and exponentially increased their number to 32. Every active thread executed 500,000 operations on the shared vector. We measured the CPU time (in seconds) that all threads needed in order to complete. Each iteration of every thread executed an operation with a certain probability; push_back (`+`), pop_back (`-`), random access write (`w`), random access read (`r`). We use per-thread linear congruential random number generators where the seeds preserve the exact sequence of operations within a thread across all containers. We executed a number of tests with a variety of distributions and found that the differences in the containers' performances are generally preserved. Analysis presented by Fraser [43] establishes that in real-world concurrent applications read operations dominate and account to about 70% to 75% of all operations. For this reason we illustrate the performance of the concurrent vectors with a distribution of `+`:15%, `-`:5%, `w`:10%, `r`:70% on Figure 3. Figure 5 demonstrates the performance results with a distribution containing predominantly writes, `+`:30%, `-`:20%, `w`:20%, `r`:30%. The number of threads is plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis. Both axes use logarithmic scale.

The current release of Intel's concurrent vector does not offer pop_back or any alternative to it. To include its performance results in our analysis, we excluded the pop_back operation from a number of distributions. Figures 4 and 6 present two of these distributions. For clarity we do not depict the results from the QueuingLock and SpinLock implementations. According to our observations, the QueuingLock performance is consistently slower than the other lock-based approaches. As indicated in [25], SpinLocks are volatile, unfair, and not scalable. They showed fast execution

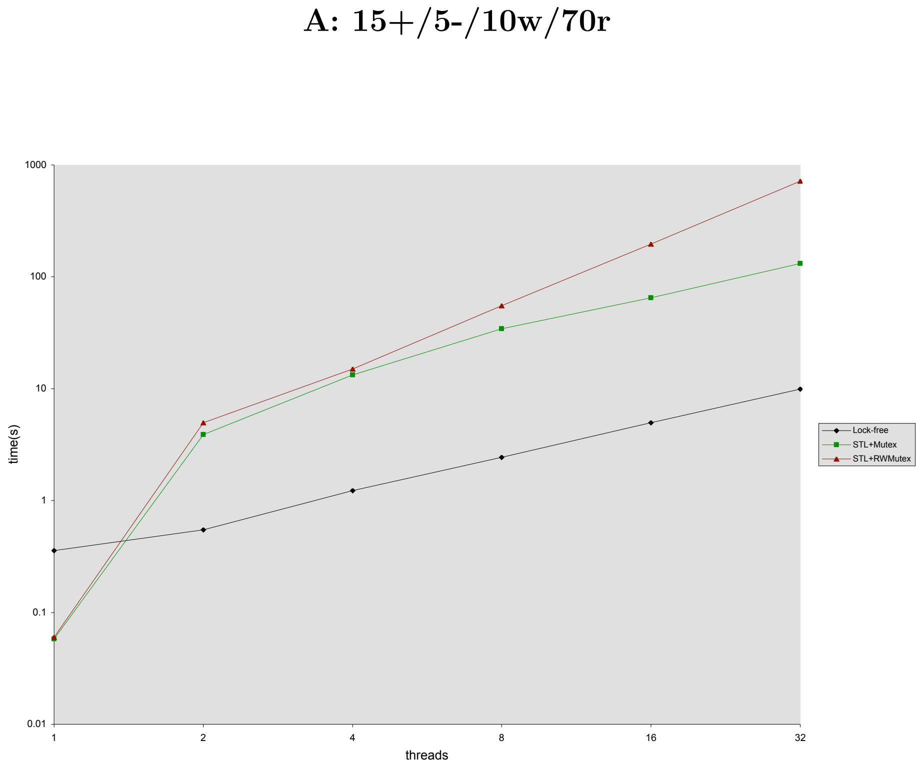


Fig. 3. Shared Vector Performance Results A - Intel Core Duo

for the experiments with 8 threads or lower, however their performance significantly deteriorated with the experiments conducted with 16 or more active threads. To find a lower bound for our experiments we timed the tests with a non-thread safe STL-vector with pre-allocated memory for all operations. For example, in the scenario described in Figure 6, the lower bound is about 10% of the performance numbers of the lock-free vector.

Under contention our nonblocking implementation consistently outperforms the alternative lock-based approaches in all possible operation mixes by a large factor.

B: 15+/0-/15w/70r

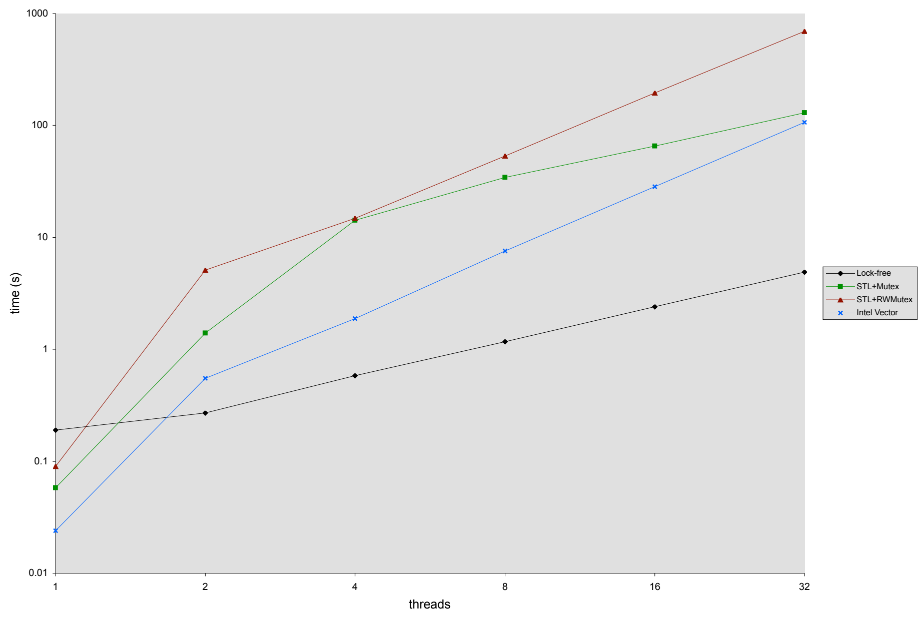


Fig. 4. Shared Vector Performance Results B - Intel Core Duo

Our lock-free design has also proved to be scalable as demonstrated by the performance analysis. Lock-free algorithms are particularly beneficial to shared data under high contention. It is expected that in a scenario with low contention, the performance gains will not be as considerable.

As discussed in Section C, we have incorporated two different memory management approaches with our lock-free implementation, namely Michael and Scott's reference counting scheme (RefCount) and Herlihy et al.'s Pass The Buck technique (PTB). We have evaluated the vector's performance using these two different memory

C: 30+/20-/20w/30r

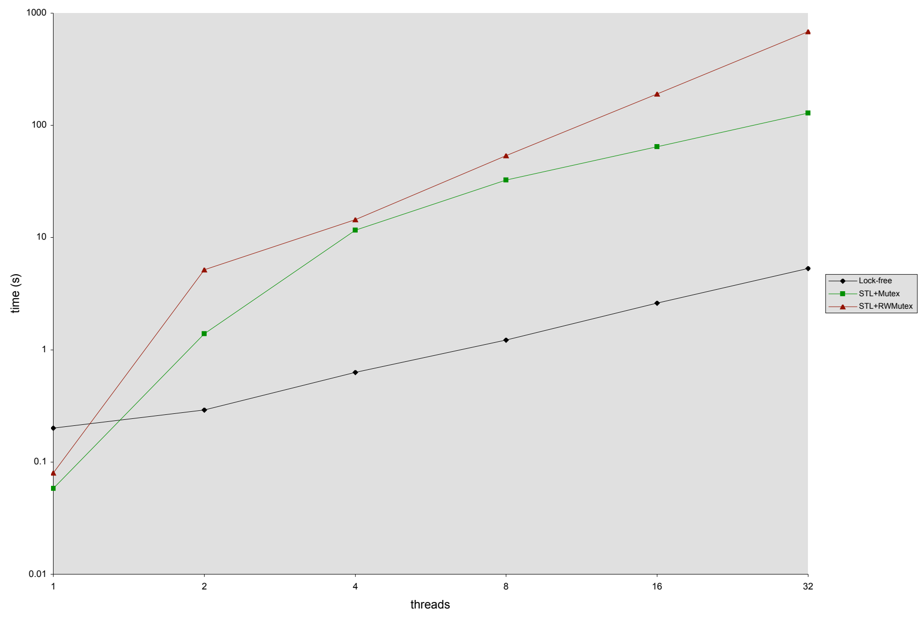


Fig. 5. Shared Vector Performance Results C - Intel Core Duo

management schemes (Figure 7).

On systems without shared L2 cache, shared data structures suffer from performance degradation due to cache coherency problems. To test the applicability of our approach on such architecture we have performed the same experiments on an AMD 2.2GHz quad dual core Opteron architecture with 1 MB L2 cache and 4GB shared RAM running the MS Windows 2003 operating system (Figure 8 and Figure 9). The applied lock-free memory allocation scheme is not available for MS Windows. For the sake of our performance evaluation we applied a regular lock-based memory allocator.

D: 35+/0-/35w/30r

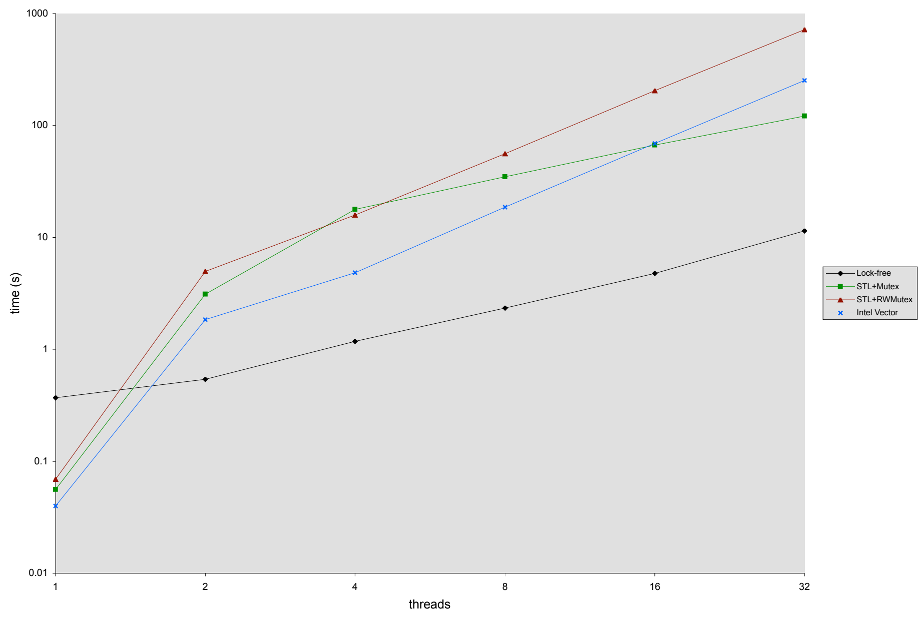
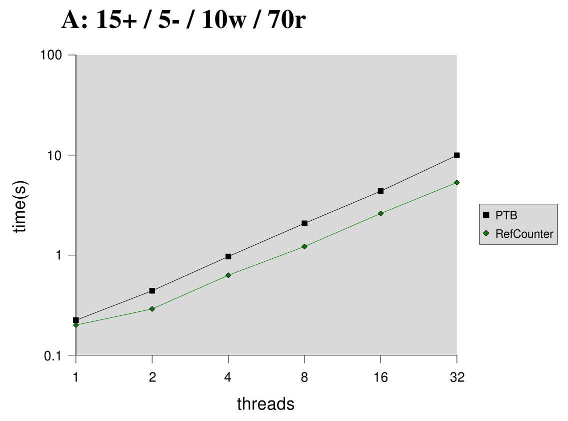


Fig. 6. Shared Vector Performance Results D - Intel Core Duo

The experimental results on this architecture lack the impressive performance gains we have observed on the dual-core L2 shared-cache system. However, the graphs (Figure 8 and Figure 9) demonstrate that the performance of our lock-free approach on such architectures is comparable to the performance of the best lock-based alternatives.

We presented the first practical and portable design and implementation of a lock-free dynamically resizable array. We developed an efficient algorithm that supports disjoint-access parallelism and incurs minimal overhead. To provide a practi-



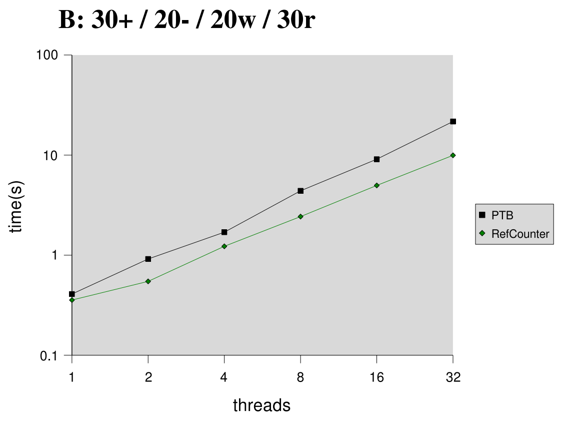


Fig. 7. Shared Vector Performance Results - Alternative Memory Management

cal implementation, our approach integrates nonblocking memory management and memory allocation schemes. We compared our implementation to the best available concurrent lock-based vectors on a dual-core system and have observed an overall speed-up of a factor of ten.

A: 15+/5-/10w/70r

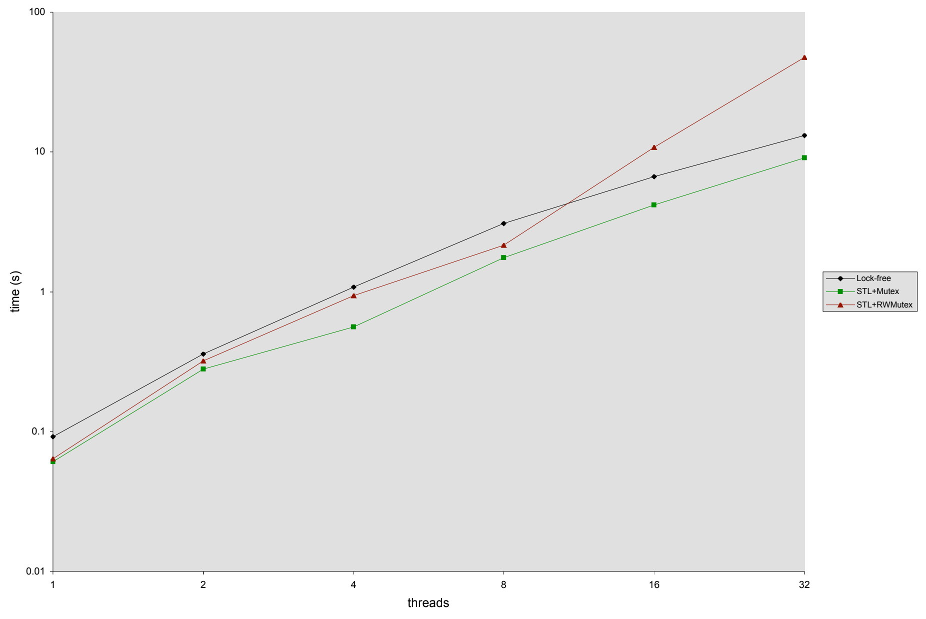


Fig. 8. Shared Vector Performance Results A - AMD 8-way Opteron

B: 15+/0-/15w/70r

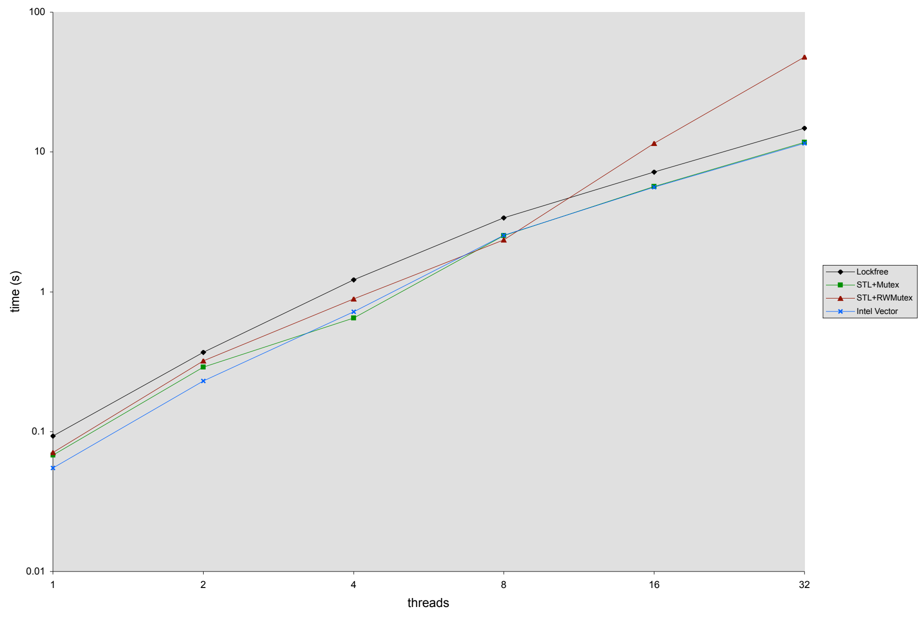


Fig. 9. Shared Vector Performance Results B - AMD 8-way Opteron

CHAPTER IV

SOFTWARE TRANSACTIONAL MEMORY

A variety of recent STM approaches [45], [26] claim safe and easy to use concurrent interfaces. The most advanced STM implementations allow the definition of efficient "large-scale" transactions, such as *dynamic* and *unbounded* transactions. *Dynamic transactions* are able to access memory locations that are not statically known. *Unbounded transactions* pose no limit on the number of locations being accessed. The basic techniques applied are the utilization of public records of concurrent operations and a number of conflict detection and validation algorithms that prevent side-effects and race conditions [45]. To guarantee progress, transactions help those ahead of them by examining the public log record. The availability of nonblocking, unbounded, and dynamic transactions provides an alternative to CAS-based designs for the implementation of nonblocking data structures. The complex designs of such advanced STMs often come with an associated cost:

a. *Two Levels of Indirection:* A large number of the nonblocking STM designs require two or more levels of indirection in accessing data.

b. *Linearizability:* The linearizability requirements are hard to meet for an unbounded and dynamic STM. To achieve efficiency and reduce the design's complexity, all known nonblocking STMs offer the weaker *obstruction-free* nonblocking guarantee [38].

c. *STM-oriented Programming Model:* The use of STM requires the developer to be aware of the STM implementation and apply an STM-oriented programming model. The effectiveness of such programming models is a topic of current discussions in the research community.

d. *Closed Memory Usage:* Both nonblocking and lock-based STMs often require a *closed memory system* [45].

e. *Vulnerability of Large Transactions:* In a nonblocking implementation large transactions are a subject to interference from contending threads and are more likely to encounter conflicts. Large blocking transactions can be subject to time-outs, requests to abort, or introduce a bottleneck for the computation.

f. *Validation:* A validation scheme is an algorithm that ensures that none of the transactional code produces side-effects. Code containing I/O and exceptions needs to be reworked as well as some class methods might require special attention. Consider a class hierarchy with a base class A and two derived classes B and C. Assume B and C inherit a virtual method f and B's implementation is side-effect free while C's is not. A validation scheme needs to disallow a call to C's method *f*.

With respect to our design goals, the main problems associated with the application of STM are meeting the stricter requirements posed by the lock-free progress and safety guarantees and the overhead introduced by the application of an extra level of indirection and the costly conflict detection and validation schemes.

A. RSTM-based Vector

The Rochester Software Transactional Memory (RSTM) [26] is a word- and indirection-based C++ STM library that offers obstruction-free nonblocking transactions. As explained by the authors [26], while helping provide lightweight committing and aborting of transactions, the extra level of indirection can cause a dramatic performance degradation due to the more frequent capacity and coherence misses in the cache. In this section we employ the RSTM library (version 4) to build an STM-based nonblocking shared vector. We chose to use RSTM because of its flexible and efficient object-oriented C++ design, demonstrated high performance when compared to alternative STM techniques, and the availability of nonblocking transactions. In Algorithms 10, 11, 12, and 13, we present the RSTM-based implementation of the read, write, pop_back, and push_back operations, respectively. According to the RSTM API [26], access to shared data is achieved by utilizing four classes of shared pointers: 1) a *shared object* (class sh_ptr $< T >$) representing an object that is untouched by a transaction, 2) a *read only object* (class rd_ptr $< T >$) referring to an object that has been opened for reading, 3) a *writable object* (class wr_ptr $< T >$) pointing to an object opened for writing by a transaction, and 4) a *privatized object* (class un_ptr $< T >$) representing an object that can be accessed by one thread at a time. These smart pointer templates can be instantiated only with data types derived from a core RSTM object class stm::Object. Thus, we need to wrap each element stored in the shared vector in a class STMVectorNode that derives from stm::Object. Similarly, we define a Descriptor class STMVectorDesc (derived from stm::Object) that stores the container-specific data such as the vector's size and capacity. The tail operations need to modify (within a single transaction) the last element and the Descriptor object (of type STMVectorDesc) that is stored in a location L_{desc}. The vector's memory array is named with the string mem. In the pseudo-code in Algorithms 12 and 13 we omit the details related to the management of mem (such as the resizing of the shared vector should the requested size exceed the container's capacity).

B. Analysis and Results

To evaluate the performance of the discussed synchronization techniques, in this section we analyze the performance of three approaches for the implementation of a

Algorithm 10 RSTM vector, operation read location p

```
1: BEGIN_TRANSACTION
2: rd_ptr< STMVectorNode > rp(mem[p])
3: result = rp->value
4: END_TRANSACTION
5: return result
```

Algorithm 11 RSTM vector, operation write v at location p

```
1: BEGIN_TRANSACTION
2: wr_ptr< STMVectorNode > wp(mem[p])
3: wp− > val = v
4: sh_ptr< STMVectorNode > nv =
   new sh_ptr< STMVectorNode >(wp)
5: mem[p] = nv
6: END_TRANSACTION
```

shared vector:

(1) The RSTM-based nonblocking vector implementation.

(2) An RSTM lock-based execution of the vector's transactions. RSTM provides an option for running the transactional code in a lock-based mode using redo locks [26]. Though blocking and not meeting our goals for safe and reliable synchronization, we include the lock-based RSTM vector execution to gain additional insight about the relative performance gains or penalties that the discussed nonblocking approaches offer when compared to the execution of a lock-based, STM-based container.

(3) The hand-crafted CAS-based algorithms design as presented in Section III.

Algorithm 12 RSTM vector, operation pop_back

```
1: BEGIN_TRANSACTION
2: rd_ptr< STMVectorNode > rp(mem[$L_{desc}$ − > size − 1])
3: sh_ptr< STMVectorDesc > desc =
   new sh_ptr< STMVectorDesc >
   (new STMVectorDesc($L_{desc}$ − > size − 1))
4: result = rp->value
5: $L_{desc}$ = desc
6: END_TRANSACTION
7: return result
```

We ran performance tests on an Intel IA-32 SMP machine with two 1.83GHz processor cores with 512 MB shared memory and 2 MB L2 shared cache running the MAC OS X operating system. We designed our experiments by generating a workload of the various operations. We varied the number of threads, starting from 1 and exponentially increased their number to 32. Each thread executed 500,000 lock-free operations on the shared container. We measured the execution time (in seconds) that all threads needed to complete. Each iteration of every thread executed an operation with a certain probability (push_back (+), pop_back (-), random access write (w), random access read (r)). We show the performance graph for a distribution of +:10%, -:10%, w:40%, r:40% on Figure 10. Figure 11 demonstrates the performance results in a read-many-write-rarely scenario, +:10%, -:10%, w:10%, r:70%. Figure 12 illustrates the test results with a distribution +:25%, -:25%, w:12%, r:38%. The number of threads is plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis. To increase the readability of the performance graphs, the y-axis uses a logarithmic scale with a base of 10. Our test results indicate that for

Algorithm 13 RSTM vector, operation push_back v

```
1: BEGIN_TRANSACTION
2: sh_ptr< STMVectorNode > nv =
   new sh_ptr< STMVectorNode >(new STMVectorNode(v))
3: sh_ptr< STMVectorDesc > desc =
   new sh_ptr< STMVectorDesc >
   (new STMVectorDesc($L_{desc}$− > size + 1))
4: mem[size] = nv
5: $L_{desc}$ = desc
6: END_TRANSACTION
```

the large majority of scenarios the hand-crafted CAS-based approach outperforms by a significant factor the transactional memory approaches. Our lock-free vector from Chapter III offers simple application and fast execution. The STM-based design offers a flexible programming interface and easy to comprehend concurrent semantics. The main deterrent associated with the application of STM is the overhead introduced by the extra level of indirection and the application of costly conflict detection and validation schemes. According to our performance evaluation, the nonblocking RSTM vector demonstrates poor scalability and its performance progressively deteriorates with the increased volume of operations and active threads in the system. In addition, RSTM transactions offer obstruction-free semantics. To eliminate the hazards of livelocking, the software designers need to integrate a contention manager with the use of an STM-based container. Because of the limitations present in the state of the art STM libraries [26], [45], we suggest that a shared vector design based on the utilization of nonblocking CAS-based algorithms can better serve the demands for safe and reliable concurrent synchronization in mission critical code. Our per-

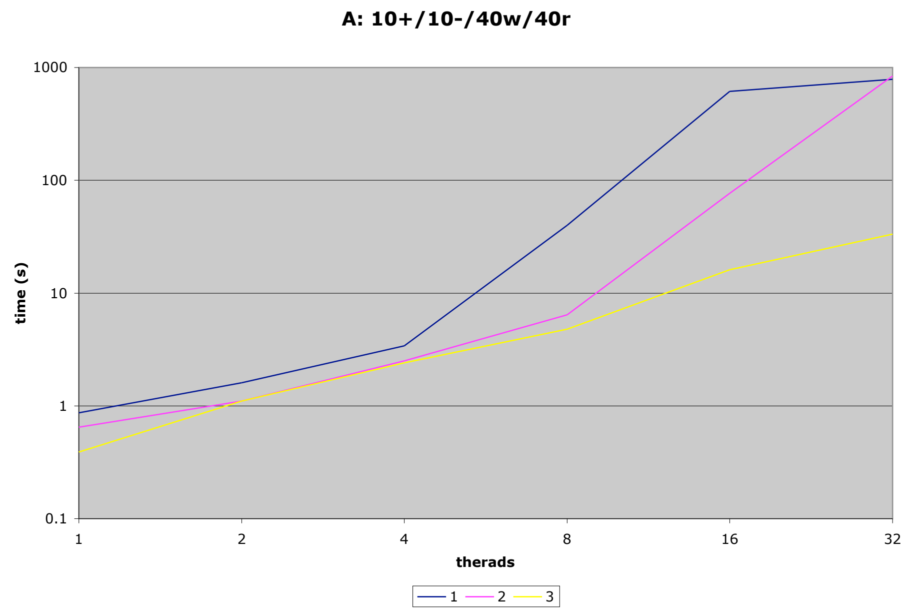


Fig. 10. STM Performance Results A

formance evaluation concluded that while difficult to design, CAS-based algorithms offer fast and scalable performance and in a large majority of scenarios outperform the alternative nonblocking STM-based approaches by a significant factor.

C. Predictive Log Synchronization

The primary advantages of the Software Transactional Memory approach is the reduction of the complexity for programming and verifying concurrent code. It allows concurrent programs to be constructed by following a sequential style of programming without the use of locks. Shalev and Shavit [37] point out that the transactional ap-

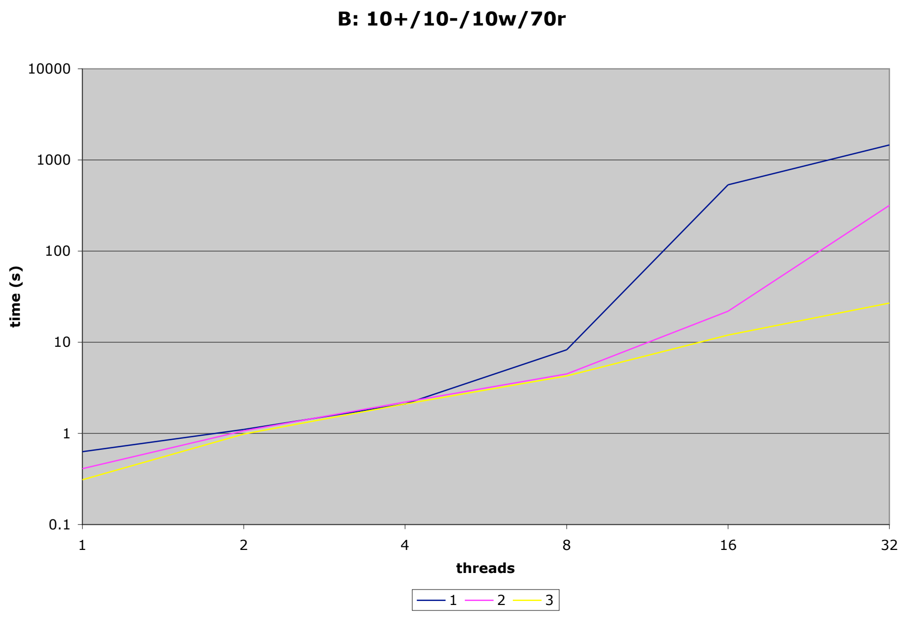


Fig. 11. STM Performance Results B

proach still requires the developer to be aware of many issues related to concurrency and some specifics of the STM-oriented programming model. Thus, the programmer faces design decision that are often similar to the ones they have to make when applying locks. For instance, one has to always balance between performance and the size of the transactions. A proposed alternative to the Software Transactional Memory Model is a recent lock-based speculative scheme called Predictive Log Synchronization [37]. The authors claim that following the Predictive Log Synchronization (PLS) paradigm, a programmer can write simply *specialized sequential* code that is automatically converted into *nonblocking* concurrent code. The paper presents the approach

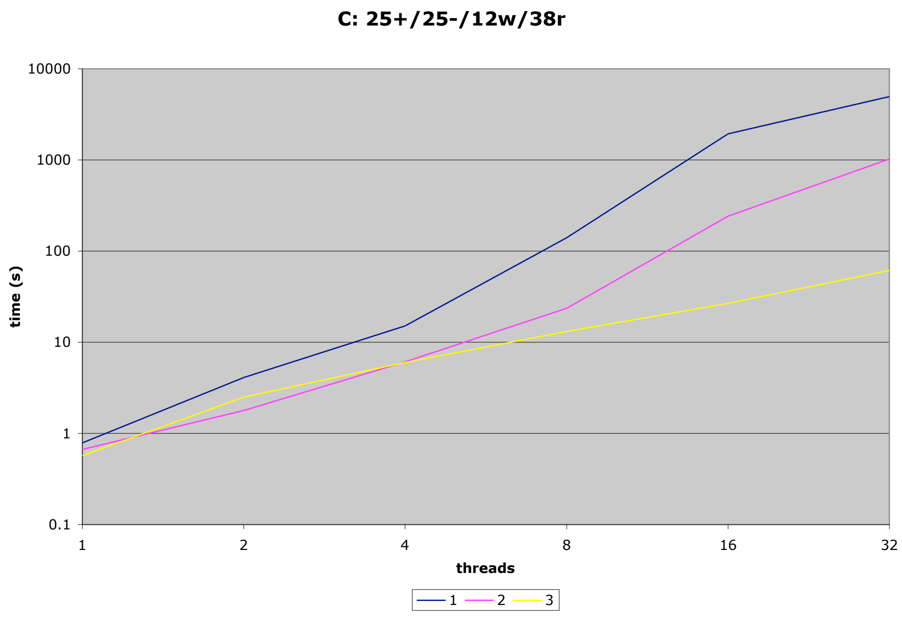


Fig. 12. STM Performance Results C

as a proof-of-concept. To avoid complexity the provided operations follow the weaker Lamport's sequential consistency model. Implementing efficient linearizable operations based on Predictive Log Synchronization is a topic of further research. The method is founded on the utilization of a lock-controlled publicly shared *log* record. All writes are serialized and executed by a single writer thread. The nonblocking property is maintained by applying speculative execution based on the public record of transactions, should a thread find the log's lock unavailable. The presented design is suitable mostly for applications that execute read-many-write-rarely operations. The authors argue that a large number of the widely used data structures have a *nat-*

ural bottleneck that prevents high volume of parallel input. Following this argument, the authors claim that delegating all updates to a single writer does not significantly affect the performance. PLS operates by duplicating the shared data structure and preventing race conditions by applying a high level lock. All concurrent threads access a publicly shared *log* record to store attempted updates and read the status of the data structure. A single thread executes all pending updates recorded in the transaction log on one of the memory copies. This allows the other threads to simultaneously have access and perform parallel reads with no interrupts and waits. The memory copies are swapped right before the lock is released. Threads that are unable to acquire the lock run an optimistic speculation. They examine the publicly shared log, make a prediction of the data structure's state and carry on with their tasks. Preliminary performance tests indicate that in some scenarios the PLS scheme offers performance gains when compare to STM-based approaches. The algorithm's performance deteriorates once the concurrency in the system increases. According to the authors [37], the PLS programming model is simpler than the STM programming model. However, it requires awareness of the way the log operates and it is not a straightforward derivation of the sequential model. In contrast to STM, in PLS the programmer does not need to consider issues related to nesting and side-effects (including I/O and exceptions). PLS requires that operation nesting or I/O can be executed only when a thread holds the global lock. The lack of linearizable lock-free operations, low parallelism, decreased performance in the scenarios with higher contention, the unavailability of the PLS implementation, and the early stage of the library's design and development, diminish the appeal of PLS for its use in the design of lock-free linearizable arrays.

CHAPTER V

SEMANTIC ENHANCEMENT OF THE SYSTEM IMPLEMENTATION

We define product-oriented certification as the process of measuring the system's reliability and efficiency based on the analysis of its design (expressed in models) and implementation (expressed in source code). In this chapter we introduce a framework for model-based product-oriented certification founded on the concept of source code enhancement and analysis. As opposed to process-oriented certification (as suggested by DO-178B [10]), the product-oriented methodology [1] relies on the application of safety concerns directly on implementation source code and its formal models. As suggested in [6], the rationale for source code enhancement is to seek an effective alternative to domain-specific programming languages for high-performance computing systems. A language enhancement can be achieved by extending a programming language by a library defining domain-specific concepts and algorithms and at the same time employing program analysis and validation tools to ensure the correctness of the introduced domain-specific notions. Source code enhancement (such as support for domain-specific policies and concurrency) allows a programmer to reach a high level of expressiveness and while still using the tool-chain of a mainstream programming language. In this chapter we suggest the concept of semantic enhancement of the source code and the application of a number of program analysis and transformation techniques to achieve reliability and efficiency in the system implementation. We describe how we identify and satisfy seven critical certification artifacts in the process of model-driven development and validation of the MDS Goal Network. In the analysis of this process, we establish the relationship among the seven certification artifacts, the applied development and validation techniques and tools, and the levels of abstraction of system design and development.

A. Principles of Model-Based Product-Oriented Certification

In this section we describe a classification of the *certification artifact types*, the *development and validation tools and techniques*, the *application domain-specific factors*, and the *levels of abstraction*. In our framework a *certification artifact type* can be one of the following:

(1) Invariant (η): a critical property or assumption that is constant (does not change throughout the transformations of program/system states) and must hold at all time to ensure the validity and correct operation of a program or a system. *Example a.*: the values stored in a given shared vector must be word-sized pointers. *Example b.*: a graph of temporal constraints [61] must contain no cycles.

(2) Guarantee (γ): a goal or condition that needs to be satisfied. Unlike invariants, goals can be defined differently at different moments of the lifecycle of a system. *Example:* an event E_a must precede an event E_b in the autonomic operation of a robot.

(3) Constraint (κ): a physical and resource constraint that need to be observed. *Example:* the physical memory available to store a graph of autonomic goals is 7168KB.

(4) Performance artifact (ϵ): an artifact describing the quality of operation and degree of optimization. *Example:* complexity and space efficiency of a particular propagation scheme in a network of temporal constraints.

(5) Comprehension artifact (σ): an artifact measuring the human understanding of the interactions, coupling, and behavior of a system. *Example:* a list of the concurrent interleaving of all processes in a goal network leading to a state of inconsistency.

(6) Re-certification and maintenance artifact (μ): an artifact demonstrating the ability to re-establish the validity of the system upon its evolution and reuse. *Example:* an automated program analysis tool that checks for the separation of data and algorithms and thus can demonstrate the validity of the graph implementation upon the replacement of the constraint propagation algorithm.

1. Development

Satisfying the certification artifacts in a software system requires the application of a combination of software development, modeling, formal verification, and analysis techniques and tools. Expressing as well as checking the certification requirements is enabled and directly dependent on the following software development dimensions:

(1) Model of Computation (Δ_{MC}): the computing architecture defined by the hardware and the operating system. It determines the sequential or parallel memory model as well as the available basic machine-level instructions and atomic primitives. *Example:* an embedded multi-core platform with eight cores supporting only single-word atomic primitives, such as the single-word Compare-And-Swap (CAS).

(2) Programming Language (Δ_{PL}): programming constructs, libraries, and techniques available. *Example:* the availability of a nonblocking vector (Chapter III) that can allow safe and lock-free access to shared data (and thus eliminate the hazards of deadlock, livelock, and priority inversion).

(3) Modeling Tools (Δ_{MT}): expressing design notions, automated code generation, and formal verification. *Example:* the application of the SPIN model checker [62] to exhaustively search the interleaving of all concurrent processes.

(4) Analysis Techniques (Δ_{AT}): program static and dynamic analysis. *Example:* the application of static analysis utilizing a high-level program representation to guarantee high performance in parallel systems [6].

(5) Software Architecture (Δ_{SA}): defines the most significant design notions such as system states, system goals, and modes of communication. *Example:* Mission Data System defines a unified model-driven architecture for testing and development of autonomous flight software based on the notions of system goals and states.

2. Application Domains

The application-domain factors have a direct impact on defining the certification requirements and the development process. We identify the following significant application-specific properties for mission critical software:

(1) Real-time (Rt): the system must achieve a goal or provide a response in a time-constrained manner. *Example:* The real-time operation of a robot demands a *system guarantee* that the meteorological process must complete prior to the initiation of communication with mission control.

(2) Safety-critical (Sc): establishes that a failure would lead to a catastrophic or hazardous consequences to the entire system. DO-178B [10] offers a hazards analysis process to assess the risk level upon a module or sub-system failure. *Example:* if the autonomous obstacle avoidance scheme fails, the rover might crash. Thus, the system *invariants* and *guarantees* assuring the correct operation of the obstacle avoidance sub-system are *safety-critical*.

(3) Embedded (Em): since the system is designed and optimized according to a set of pre-defined goals, its software must often control the hardware, consider strict resource constraints, and handle failures and events that may occur in the physical world. *Example:* the embedded nature and limited memory availability of the rover places the *constraint* that a goal network should not exceed a certain concrete limit of memory space.

(4) Autonomous (Au): the system must achieve a set of goals with little or no human interaction, meanwhile possibly responding to the conditions and events in its environment. *Example:* the autonomy of the meteorological and bus management processes requires the *invariant* that the system is free of the hazards of priority inversion.

3. Levels of Abstraction

We classify the system's safety concerns according to their rank in the abstraction hierarchy:

(1) Physical and Hardware (Φ): related to constraints in the hardware resources, organization, and architecture and the conditions in the physical environment. *Example:* the lack of complex atomic primitives on the flight-qualified hardware requires all nonblocking code to rely on the single-word Compare-And-Swap (CAS) atomic primitive. This demands the specification of an *invariant* that the system must eliminate the possibility of occurrence of the ABA problem [7].

(2) Algorithms and Procedures (Θ): invariants of a particular computational routine or algorithm. *Example:* the complexity of Floyd-Warshall's all-pairs-shortest-path algorithm [59] is $O(N^3)$. Due to the frequent execution of the constraint propagation scheme in a goal network, the direct application of the algorithm can

be prohibitively expensive. To meet the *performance* requirements a propagation scheme should execute with complexity of at most $O(N^2)$.

(3) Libraries (Λ): domain-specific concerns on a set of algorithms that are grouped in a standard or custom language extension. *Example:* a library of CAS-based nonblocking algorithms must guarantee its ABA-freedom.

(4) Modules and Sub-Systems (Ψ): guarantees and quality of service provided by the individual components and sub-systems. *Example:* the rover's module performing atmospheric experiments must coordinate its execution with the bus management and the communication systems. Such a coordination might lead to a number of safety-critical invariants and guarantees (such as no priority inversion).

(5) System (Ω): goals critical for the successful completion of the mission. *Example:* the rover's goal is to autonomously navigate the surface of Mars, perform scientific exploration of the planet's atmosphere and geology, and communicate results back to mission control. Meeting these goals impacts the guarantees defined by all of the robot's sub-systems.

(6) Framework (Ξ): conditions related to the principle organization and design of the software development. *Example:* Mission Data System defines the notions of states and goals. Their definition and requirements are described (independently from the implementation of a particular mission) in a number of MDS framework papers such as [9].

As emphasized by Stroustrup in [63], the concept of higher-level systems programming is of significant importance to systems of high complexity and size. Higher-level systems programming implies that while low-level efficiency is important, the emphasis is placed towards the design, maintenance, and validation of the larger system. With respect to the system implementation, it is the programming language facilities for data abstraction and representation of domain-specific concerns that directly address this issue. As defined by Stroustrup [63]:

> A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed and a set of *concepts* for the programmer to use when thinking about what can be done. The first aspect ideally requires a language that is 'close to the machine', so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The C language was primarily designed with this in mind. The second aspect ideally requires a language that is 'close to the problem to be solved', so that the concept of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed with this in mind.

The application of C++ in a framework for complex, autonomous, and embedded flight software, such as Mission Data System, further illustrates and emphasized the significance of the ability of C++ to excel in providing both, instructions 'close to the machine' and facilities that are 'close to the problem to be solved'. Language facilities allowing the definition of high-level design concepts and domain-specific concern are often provided by language libraries. Such libraries enhance the language semantic model by defining notions and guarantees that belong to the problem domain.

Modeling and formal verification tools such as SPIN [62], Alloy Analyzer [64], and Eclipse [65] are used to express and validate high-level domain-specific and design concerns. The challenges associated with the application of modeling and formal verification tools in the development process are:

(1) Bridging the implementation source and the software models.

   (a) From implementation to models: as an abstraction and simplification of the software implementation, a model represents an aspect of the software solution based on a number of assumptions and rules. Defining these assumptions as well as the verification invariants, and establishing whether the model is trustworthy with respect to the source are some of the most challenging tasks.

   (b) From models to implementation: the application of program synthesis techniques such as AutoFilter [35] have been applied successfully in a number or flight applications. However, the certification of the produced software is challenged by the strict FAA requirement of having the program synthesis meet the same certification requirements as the produced flight software.

(2) Limited state space and heavy computational complexity: despite the advanced state space reduction techniques in many modern formal verification tools, the main limitations for their applicability arise from the heavy computational complexity imposed and the state space explosion problem. Program simplification and abstract interpretation techniques are often necessary to reduce the explored state space. Certification standards (such as those from FAA) require the developers to establish the preservation of the program's semantics upon the application of any program transformation and abstract interpretation techniques.

(3) Project Scheduling: the application of formal logic can often be as demanding to the software developers as the engineering of the system implementation itself.

The semantic enhancement of the implementation can allow for the direct validation of some software invariants and guarantees and thus reduce the state space and the computational complexity required in the process of formal verification. In addition, the increased expressiveness and abstraction level of the implementation source can ease the manual or automated transition to and from the software models. Stroustrup and Dos Reis [6] present the notion of *Semantically Enhanced Library Languages (SELLs).* As defined by the authors, a SELL is a domain-specific language derived from a general-purpose programming language by extending it with libraries defining the concepts and functionalities of the problem domain and then applying an analysis tool to guarantee the higher-level semantic invariants. The main advantages of defining and applying a SELL are founded in the availability of the maintenance, training, and tool-chain of the general-purpose language that serves as its base. At the same time, a SELL's main purpose is to deliver a special-purpose language tailored to the ideals and concepts of a specific application domain. The notion of SELL is fundamental for the application of our model-based product-oriented framework for software certification.

The following chapters describe the details of how we extend the semantics of ISO C++ with the libraries defining Temporal Constraint Networks and Semantically Enhanced Containers for safe lock-free concurrent access. Furthermore, in the process of validation and automatic parallelization of MDS Goal Networks, we demonstrate how the applied programming and modeling techniques, formal verification, program transformation, and static analysis relate to our classification framework.

CHAPTER VI

SEMANTICALLY ENHANCED CONTAINERS

In this chapter we present the definition, design, and implementation of the concept of *Semantically Enhanced Containers (SECs).* SECs are data structures designed to provide the flexibility and usability of the popular ISO C++ Standard Template Library (STL) containers [63], while at the same time they are hand-crafted to guarantee domain-specific policies, such as the validity of given user-defined semantic invariants and conformance to a specific concurrency model. In this dissertation we require a SEC to address the following particular design goals: a) built-in safe concurrent synchronization suitable for real-time embedded applications, b) use of static analysis for enhanced safety such as the elimination of the ABA problem, and c) syntactic interface and semantics similar to the widely applied and supported ISO C++ STL containers. The objective of this chapter is to introduce the notion, present an initial implementation, and demonstrate the benefits of Semantically Enhanced Containers. The SEC implementation presented in this chapter targets the effective elimination of the fundamental ABA problem with the application of C++ static analysis. For a detailed discussion and examples on ABA please see Chapter VII. The core tool for implementing our static checks is The Pivot [6], a general high-level framework for ISO C++ program analysis and semantic-based transformations. The application of static analysis allows us to shift the complexity of preventing ABA from the run-time of the system to the compile-time program analysis stage. As our performance analysis confirms, such an approach relying on static analysis delivers significant performance benefits when compared to the traditional run-time and garbage collection-based ABA prevention techniques. We demonstrate the SEC proof-of-concept by providing the design and implementation of a concurrent Semantically Enhanced STL vector. The

SEC vector presented in this work is engineered to meet the following design goals:

(a) *Allow efficient and reliable concurrent interactions:* to achieve high performance and avoid the hazards of deadlock, livelock, and priority inversion, the shared vector's operations are lock-free and linearizable [66]. In addition, our design is portable: all of the vector's algorithms are based on the word-size Compare-And-Swap (CAS) instruction [67] available on a large number of hardware platforms.

(b) *Ensure the validity of user-defined semantic invariants:* we introduce Basic Query (BQ), an expression template-based library for extracting semantic information from C++ source code. BQ defines the programming techniques for specifying and statically checking domain-specific properties in code. We apply BQ to avoid the ABA problem in the usage of our concurrent vector.

The shared vector presented in Chapter III does not employ an ABA prevention scheme beyond the application of nonblocking memory management and if not used according to its usage rules might be vulnerable to the occurrence of ABA. Our test results show that the SEC vector delivers significant performance gains (a factor of three or more) in contrast to the application of nonblocking synchronization amended with the traditional ABA avoidance scheme.

A. Using Static Analysis to Express and Validate Domain-Specific Guarantees

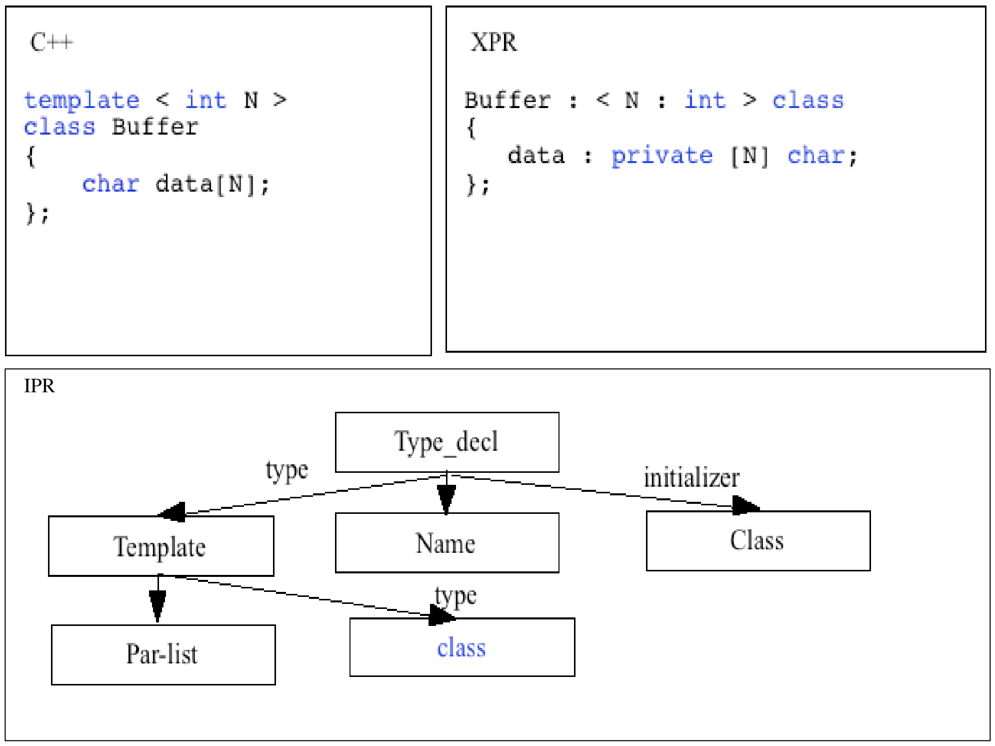
In this section we present Basic Query (BQ), a static analysis library for extracting semantic information from C++ source code. *BQ user-defined actions* are executed by traversing a compact high-level abstract syntax tree (AST) called Internal Program Representation (IPR) [6]. The use of static analysis allows us to reach a far more efficient and reliable implementation of our nonblocking containers than would otherwise have been possible. IPR is at the center of a C++ static analysis framework named The Pivot [6]. We the application of BQ by defining the semantic rule Exclude_push_back that disallows the use of a push_back operation in certain hazardous scenarios and helps us avoid the ABA problem.

The Pivot is a compiler-independent platform for static analysis and semantics-based transformation of the complete ISO C++ programming language and some advanced language features proposed for the next generation C++, C++0x [53]. The Pivot represents C++ programs in two distinct formats (Figure 13):

1. Internal Program Representation (IPR): IPR is a high level, compact, fully typed abstract syntax tree that can represent complete ISO C++ programs as well as incomplete program fragments and individual translation units,

2. eXternal Program Representation (XPR): XPR is a persistent and human readable format for program representation. XPR uses a prefix notation and is quick to parse using only a single token look-ahead and not needing a symbol table.

In addition, The Pivot provides the basic tools for IPR construction from C++, IPR-to-XPR and XPR-to-IPR conversion, some general traversal and transformation interfaces (such as the support of the visitor traversal pattern [68]), and an IPR-to-C++ and IPR-to-XML back-ends. The present state-of-the-art of The Pivot development allows IPR generation from the front ends of the popular EDG and GCC C++ compilers. Our SEC approach is supported by the availability of the high-level and compact Pivot's Internal Program Representation (IPR). IPR aims at delivering a C++ program representation that is general-purpose (effective for a large number of application domains), complete (able to elegantly express all of the language's features), and high-level (express program notions in a compiler-independent fashion that is close to the source code and the programmer's semantic concepts). In IPR a C++ program is represented as a fully-typed AST graph. Each IPR node corresponds



C++

```
template < int N >
class Buffer
{
    char data[N];
};
```

XPR

```
Buffer : < N : int > class
{
   data : private [N] char;
};
```

Fig. 13. An XPR and IPR Representation of a C++ Template Class Definition

to a high-level C++ notion such as a template, expression, declaration, statement, scope, name, etc. The root of the AST graph is an **ipr::Unit** node that contains the program representation of a translation union (post-template-instantation). IPR focuses on the extensive support of C++ types and views types as a main mechanism for ensuring program safety and efficiency. The set of **ipr::Type** nodes includes:

a. the recursive nodes **ipr::Pointer** to **ipr::Type**, **ipr::Reference** to **ipr::Type**, **ipr::Const**, **ipr::Volatile**, and a template of a type (**ipr::Template**),

b. an array representation, **ipr::Array**,

c. a function, **ipr::Function**,

d. a class, **ipr::Class**,

e. a union, **ipr::Union**,

f. an enum, **ipr::Enum**,

g. a namespace, **ipr::Namespace**,

h. a declaration type, **ipr::Decltype**,

i. a type expression, **ipr::Type_expr**,

j. a product, **ipr::Product**,

k. a sum, **ipr::Sum**.

The nodes **ipr::Pointer**, **ipr::Reference**, and **ipr::Array** correspond directly to the conventional C++ language constructs pointer, reference, and array. The pair of nodes **ipr::Decltype** and **ipr::Type_expr** allow for type querying the AST nodes. The purpose of **ipr::Type_expr** is to convert any **ipr::Expr** node into a type. The **ipr:Decltype** class supports interfaces that allow the extraction of the argument used to construct an **ipr::Type_expr**. While not directly supported in the present ISO C++ standard, this type querying functionality is considered to be important for the support of generic programming [53] and is going to be included in the next C++ generation, C++0x [53]. Similarly, the nodes **ipr::Product** and **ipr::Sum** are used to better express higher-level notions and are not directly present in the language's syntax. An **ipr::Product** node represents a list of parameters. An **ipr::Sum** node is a model of a C++0x concept [69], i.e. it represents a sum of types that posses a set of common interfaces and properties. The **ipr::Template** node is a generalization of the C++ template construct. In C++ a function or a class template declaration allows the generic definition of an algorithm or a data type. IPR enhances the notion of a template by assuming that any C++ declaration (e.g. variables, namespaces) could be defined in a generic fashion. An **ipr::Template** node is instantiated with an **ipr::Product** node containing the template's parameter list of types and an **ipr::Expr** node representing the C++ expression (classes and functions are viewed as expression nodes in IPR) to be parameterized. For space efficiency and scalability, IPR features node unification [6], thus nodes that represent semantically equivalent program entities share the same address space.

Fundamental to our BQ library is the design of a fast and flexible methodology for traversing the IPR, The Pivot's AST. We define a depth-first search (DFS) visitor class, called the IPR Xplorer visitor class, that performs the AST search following the order of the ISO C++ grammar definition. The Xplorer allows the programmer to statically define a set of actions to be executed during the DFS traversal, including a terminating condition as well as actions upon the encounter of specific IPR nodes (C++ expressions, declarations, and statements) and AST edges (interfaces of the IPR nodes). In such a design the cost of a user-defined action could be less than a single traversal of the abstract syntax tree. When an action is specified, the programmer instantiates the traversal object with two compile-time arguments, a TRP (trigger point) identifying the exact point in the AST of calling the action's function, and a TN (target nodes) specifying the type of IPR nodes which are the traversal's target. The following examples illustrate the usage of the Xplorer visitor:

(a) xplore_expr_node < discover, ipr::Call >, we specify an action at the point of discovery of each **ipr::Call** node,

(b) xplore_stmt_node < body, ipr::Switch >, a user-defined action is executed prior to exploring the edge body of an IPR node of type **ipr::Switch**.

In some scenarios we prefer to have linear access to the nodes of a program unit and at the same time manipulate the AST through an intuitive and familiar user interface. Our Xplorer visitor defines the classes: IPR_Visitor and IPR_Iterator. Their design closely follows the functionality and philosophy of the visitor design pattern [70] and the C++ STL Iterator [63] classes, providing a convenient way to search, manipulate, or modify a set of IPR objects. The convenience of this method comes at a certain price: the DFS traversal needs to collect and store in advance all of the nodes from a program unit, thus the cost of the user-specified actions is at least a single traversal of the AST.

BQ user-defined actions are constructed at compile time by using the mechanism of expression templates [71] (thus the query implementation avoids the usage of costly pointers to class member functions). Expression templates are not used in the construction of the entire pattern tree because of the heavy syntax that such an approach would impose. Instead, the 'glue' among all statically computed *BQ elements* is encoded in the *BQ operations* (Table 7). The clean and flexible syntax of the BQ user-defined actions is achieved through the exploitation of the C++ compiler's ability to perform complex template argument inference. A *BQ action* (also a BQ pattern) consists of three components: a *Recursive Query Object (RQO)* containing the root of the traversal as well as the result from an applied pattern or a sequence of patterns, a set of *BQ elements*, and a set of *BQ operations*. At each step of the AST traversal, the RQO decides whether the target is reachable from the current point and carry on with the execution of the pattern or terminate the search. A *BQ pattern* is expressed through a combination of a number of BQ elements and BQ operations applied to the recursive query object. There are a number of possible applications of the BQ operations on the BQ elements (Table 8 and Table 9). A BQ element specifies one or several edges in the pattern tree. A BQ element could be one of three possible types:

1. Exe_member $< x, e >$. Execute Member (EM) generates a straightforward edge e from an IPR node (vertex) x. For example, if the vertex x is an IPR node of type ipr::Type_decl and the edge e is ipr::initializer, the result of the operation is the IPR node yielded by the execution of the IPR interface x→initializer (that is the initializer of a C++ type declaration).

2. Exe_condition $< x, e, c >$. Execute Condition (EC) generates an edge e from an IPR node x, only if a specified boolean condition c is met.

3. Exe_iprseq $< x, e_n >$. Execute Sequence (ES) produces a sequence of edges e_n resulting in a set of IPR nodes. An example of such an edge in the pattern tree is the call to retrieve all bases of a class declaration (x→bases()).

Table 7. Basic Query Operations

Operation	Operand	Description
Apply	$<$	applies action specified by a BQ element
Apply & Evaluate	$\wedge$	executes a BQ element and returns
Evaluate	$\rightarrow$	applies a BQ pattern and returns

We use Basic Query to enforce domain-specific semantic rules and avoid certain hazardous concurrent interleaving of the vector's tail operations that might lead to the occurrence of the ABA problem. In a number of MDS concurrent applications, there are multiple reader threads but only a single writer. Such a scenario is ABA-free

Table 8. Application of the Basic Query Operations

Operation	Operation Description
RCO $<$ ES	applies an ES
RCO $<$ EM	executes an EM, stores the result in RQO
RCO $<$ EC	executes an EC, stores the result in RQO
RCO $\wedge$ EC	executes an EC, stores the result in RQO
(Set of IPR Nodes) $\wedge$ EC	searches for a match for EC's condition

Table 9. Result Description of the Basic Query Operations

Operation	Result Description
RCO $<$ ES	sequence of IPR nodes
RCO $<$ EM	a pointer to RQO
RCO $<$ EC	a pointer to RQO
RCO $\wedge$ EC	the evaluation of EC's condition
(Set of IPR Nodes) $\wedge$ EC	true if at least one node satisfies the predicate

since it is not possible to have an interrupting writer thread placing a hazardous old value back to its location. In such a case, it is necessary to implement a BQ routine applied to all reader threads that checks for the exclusion of write operations. In a scenario of multiple writer threads, the ABA-free semantics are achieved by statically enforcing two distinct semantic phases for all writer threads in the system: a growth phase and an operational phase. Table 10 enumerates the possible interleaving of two concurrent operations of the SEC vector and indicates those prone to ABA and those that are ABA-free. The semantic ABA-free phases are:

1. Growth phase: allows only push_back and random access read by all threads.

2. Operational phase: allows all operations (pop_back and the random access read and write) except push_back.

The static enforcement of the semantic phases is achieved by defining BQ rules that exclude the usage of certain operations during a given phase (such as the exclusion of push_back in the operational phase). In Algorithms 14 and 15 we show the pseudo-code and the actual source code of the semantic rule Exclude_push_back defined using the BQ elements and BQ operations. We use the Xplorer visitor to collect all IPR Expression nodes. Afterwards, we apply the IPR_Iterator to search the collection of IPR Expressions for Function_call nodes (expressed by the EM1 element in Algorithm 14) and then test whether a function call's name is "push_back" (expressed by the EC1 element in Algorithm 14).

Algorithm 16 illustrates the source code of a test routine executing 32 threads, each invoking 500,000 operations on the shared vector. To test the application of the static rule shown in Algorithm 15, we have applied BQ and our Exclude_push_back operation in order to raise a warning and eliminate the invocation of push_back at line 19, Algorithm 16, thus eliminate the hazards of ABA occurrence in the vector's operational phase.

Algorithm 14 Exclude_push_back: Find an Illegal push_back

```
1: RCO: ipr::Expr
2: EM1: ipr::Function_call $\rightarrow$ name
3: EC1: ipr::String $\rightarrow$ name_cmp
4: Exclude_push_back: RCO $<$ EM1 $<$ EC1 $\rightarrow$ bool
```

Table 10. ABA-free and ABA-prone Interleaving of Two Concurrent Operations

operation	push	pop	read	write
push	ABA free	ABA	ABA free	ABA
pop	ABA	ABA free	ABA free	ABA free
read	ABA free	ABA free	ABA free	ABA free
write	ABA	ABA free	ABA free	ABA

Algorithm 15 Exclude_push_back: Find an Illegal push_back, source code

```
1: Input: an IPR Expression node $e$
2: Recursive_query RCO($e$);
3: Exe_member<ipr::Function_call, name> Get_name;
4: Exe_condition <ipr::String, name, const ipr::Name&, std::string>
   Is_Name(&name_cmp, parent_name);
5: return RCO < Get_Name < Is_Name;
```

B. SEC Performance Analysis

To gain insight of the possible performance gains of the SEC approach we ran performance tests on an Intel IA-32 SMP machine with two 1.83GHz processor cores with 512 MB shared memory and 2 MB L2 shared cache running the MAC OS X operating system. In our performance analysis we compare:

(a) The SEC vector approach (with the enforcement of semantic phases and integrated lock-free memory management and allocation).

(b) The application of the nonblocking operations of the dynamically resizable array from Chapter III. To prevent ABA we employed the traditional ABA avoidance technique used in CAS-based designs, namely introducing an extra level of indirection (to guarantee the uniqueness of each new element) and protecting the deallocated memory (from being re-allocated and causing ABA) by a lock-free memory management scheme. In our performance tests we used Herlihy et al.'s *Pass The Buck* (PTB) algorithm [55].

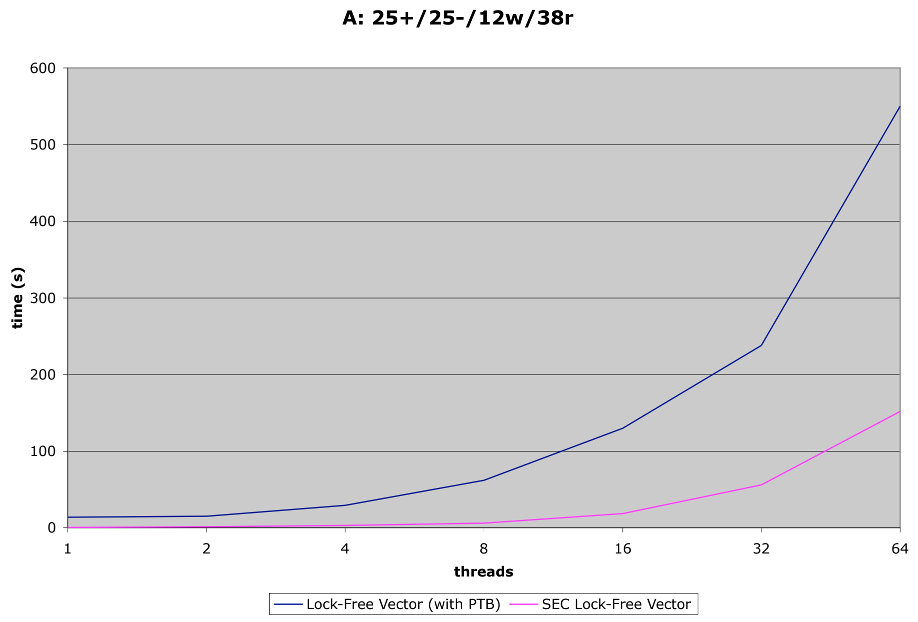
Similarly to the evaluation of other lock-free concurrent containers [43], we have designed our experiments by generating a workload of various operations (`push_back`, `pop_back`, random access `write`, and `read`). We followed the semantic rules of the *operational* and *growth* phase when executing the operations. We varied the number of threads, starting from 1 and exponentially increased their number to 64. Each thread executed 500,000 lock-free operations on the shared container. We measured the execution time (in seconds) that all threads needed to complete. Each iteration of every thread executed an operation with a certain probability; `push_back` (`+`), `pop_back` (`-`), random access `write` (`w`), random access `read` (`r`). We use per-thread linear congruential random number generators where the seeds preserve the exact sequence of operations within a thread across all containers. We executed a number of tests with a variety of distributions and found that the differences in the containers' performances are generally preserved. We illustrate the performance of the concurrent vectors with a distribution of +:16%, −:16%, w:18%, r:50% on Figure 14B. Figure 14A demonstrates the performance results with a distribution containing predominantly writes, +:25%, −:25%, w:12%, r:38%. The number of threads is plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis.

According to the performance results, the SEC approach delivers consistent performance gains in all possible operation mixes by a large factor. The SEC vector has also proved to be scalable as demonstrated by the performance analysis. These observations come as a confirmation to our expectations that introducing an extra level of indirection and the necessity to memory manage each individual element with PTB (or an alternative memory management scheme) to avoid ABA comes with a pricy performance overhead. The SEC approach offers an alternative by introducing the notion of semantic phases in order to reduce the performance overhead of the ABA avoidance mechanism.

This chapter introduced the concept and initial implementation of the notion of *Semantically Enhanced Containers (SECs)*. We demonstrated the SEC proof-of-concept by presenting the design and implementation of a concurrent nonblocking SEC vector. The main design goals are to achieve efficient and reliable concurrent synchronization and allow the specification and validation of user-defined semantic guarantees. In the presented design, the SEC vector's operations are safe (no hazards of deadlock, livelock, priority inversion), lock-free, linearizable, fast, highly parallel, and at the same time providing the semantics of the popular STL C++ vector, with complexity of $O(1)$. To deliver a mechanism for the specification and checking of user-defined semantic invariants, we introduced Basic Query, an expression template-based library for extracting semantic information from C++ source code. We applied Basic Query to help us avoid a fundamental problem in all CAS-based systems, namely the occurrence of the ABA problem. Providing domain-specific guarantees together with a scheme for reliable concurrent synchronization is of critical importance for the design and development of the modern complex and highly autonomous space systems. The integration of the SEC vector's lock-free algorithms can help achieve better performance, scalability, and higher safety in a number of pivotal Mission Data System applications. Our preliminary tests indicate that our SEC approach provides significant performance gains in contrast to the application of nonblocking synchronization amended with the traditional ABA avoidance scheme.

Algorithm 16 Using BQ to eliminate ABA hazards, source code

```
static int no_threads = 32;
static long no_ops = 500000;
static int no_reads = 307;  {25%}
static int no_writes = 512;  {25%}
static int no_push = 819;   {37.5%}
static int no_pop = 1024;   {12.5%}
boost::rand48 nard;
size_t vecsize = 0;
for (int i = 0; i < no_ops; ++i) do
   int op = (nard() % 1024);
   value_type elem = NULL;
   char opdesc;
   if (!vecsize) then
      op = no_writes;
   if (op >= no_writes) then
      if (op < no_push) then
         opdesc = '+';
         elem = new int(i);
         vecsize = vec− >push_back(elem);
      else
         opdesc = '-';
         elem = vec− >pop_back();
   else
      size_t pos = nard() % (vecsize − (vecsize / 16));
      if (op > no_reads) then
         opdesc = 'w';
         elem = new int(i);
         vec− >write_i(pos, elem);
      else
         opdesc = 'r';
         elem = vec− >read_i(pos);
```



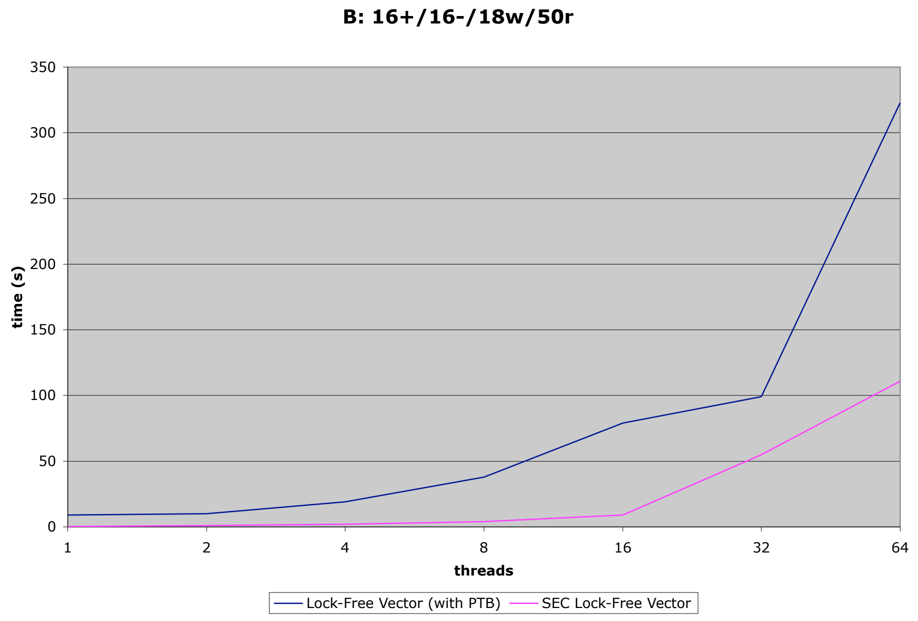


Fig. 14. SEC Performance Analysis

CHAPTER VII

NONBLOCKING ABA-FREE SYNCHRONIZATION

Lock-free and wait-free algorithms exploit a set of portable atomic primitives such as the word-size Compare-and-Swap (CAS) instruction [67]. The design of nonblocking data structures poses significant challenges and their development and optimization is a current topic of research [24], [8]. The Compare-And-Swap (CAS) atomic primitive (commonly known as Compare and Exchange, CMPXCHG, on the Intel *x86* and *Itanium* architectures [27]) is a CPU instruction that allows a processor to atomically test and modify a single-word memory location. CAS requires three arguments: a memory location (L_i), an old value (A_i), and a new value (B_i). The instruction atomically exchanges the value stored at L_i with B_i, provided that L_i's current value equals A_i. The result indicates whether the exchange was performed. For the majority of implementations the return value is the value last read from L_i (that is B_i if the exchange succeeded). Some CAS variants, often called Compare-And-Set, have a return value of type boolean. The application of a CAS-controlled speculative manipulation of a shared location (L_i) is a fundamental programming technique in the engineering of nonblocking algorithms [8], [24] (an example is shown in Algorithm 17).

Algorithm 17 CAS-controlled speculative manipulation of L_i

```
1: repeat
2:     value_type $A_i$=$\hat{}L_i$
3:     value_type $B_i$ = fComputeB
4: until CAS($L_i$, $A_i$, $B_i$) == $Bi$
```

When the value stored at L_i is the target value of a CAS-based speculative manipulation, we call L_i and $\hat{}L_i$ *control location* and *control value*, respectively. We indicate the control value's type with the string value_type. The size of value_type must be equal or less than the maximum number of bits that a hardware CAS instruction can exchange atomically (typically the size of a single memory word). In the most common cases, value_type is either an integer or a pointer value. In the latter case, the implementor might reserve two extra bits per each control value and use them for implementation-specific value marking [24]. This is possible if we assume that the pointer values stored at L_i are aligned and the two low-order bits have been cleared during the initialization. In Algorithm 17, the function fComputeB yields the new value, B_i, to be stored at L_i. We assume that B_i is not dependent on the control value (A_i) and is usually derived from the function's parameter list. A routine where B_i's value is dependent on A_i would be a *read-modify* routine in contrast to the *modify* routine shown in Algorithm 17.

Definition 1: *The ABA problem is a false positive execution of a CAS-based speculation on a shared location L_i.*

As illustrated in Table 11, ABA can occur if a process P_1 is interrupted at any time after it has read the old value (A_i) and before it attempts to execute the CAS instruction from Algorithm 17. An interrupting process (P_k) might change the value at L_i to B_i. Afterwards, either P_k or any other process $P_j \neq P_1$ can eventually store A_i back to L_i. When P_1 resumes, its CAS loop succeeds (false positive execution) despite the fact that L_i's value has been meanwhile manipulated.

Table 11. ABA at L_i

Step	*Action*
Step 1	P_1 reads A_i from L_i
Step 2	P_k interrupts P_1; P_k stores the value B_i into L_i
Step 3	P_j stores the value A_i into L_i
Step 4	P_1 resumes; P_1 executes a false positive CAS

Definition 2: *A nonblocking algorithm is ABA-free if its semantics cannot be corrupted by the occurrence of ABA.*

ABA-freedom is achieved when: *a.* occurrence of ABA is harmless to the algorithm's semantics or *b.* ABA is avoided. The former scenario is uncommon and strictly specific to the algorithm's semantics. The latter scenario is the general case and in this work we focus on providing details of how to eliminate ABA.

A. Known ABA Detection and Avoidance Techniques, Part I

A general strategy for ABA avoidance is based on the fundamental guarantee that no process P_j ($P_j \neq P_1$) can possibly store A_i again at location L_i (Step 3, Table 11). One way to satisfy such a guarantee is to require all values stored in a given control location to be *unique*. To enforce this uniqueness invariant we can place a constraint on the user and request each value stored at L_i to be used only once (*Known Solution 1*). Enforcing this constraint can be facilitated if a programming language's type system supports uniqueness typing [72] that forbids the use of more than a single reference to an object. We are not familiar with any programming language or library that implements uniqueness typing in a concurrent environment. To achieve this goal, it would be necessary to design and apply a complex tool-chain of static and dynamic program analysis. For a large majority of concurrent algorithms, enforcing uniqueness typing would not be a suitable solution since their applications imply the usage of a value or reference more than once.

An alternative approach to satisfying the uniqueness invariant is to apply a *version tag* attached to each value. The usage of version tags is the most commonly cited solution for ABA avoidance [67]. The approach is effective, when it is possible to apply, but suffers from a significant flaw: a portable single-word CAS instruction is insufficient for the atomic update of a word-sized control value and a word-sized version tag. An effective application of a version tag [73] requires the hardware architecture to support a more complex atomic primitive that allows the atomic update of two memory location, such as CAS2 (compare-and-swap two co-located words) or DCAS (compare-and-swap two memory locations). The availability of such atomic primitives might lead to much simpler, elegant, and efficient concurrent designs (in contrast to a CAS-based design). It is not desirable to suggest a CAS2/DCAS-based ABA solution for a CAS-based algorithm, unless the implementor explores the optimization possibilities of the algorithm upon the availability of CAS2/DCAS. A proposed hardware implementation (entirely built into a present cache coherency protocol) of an Alert-On-Update (AOU) instruction [74] has been suggested by Spear et al. to eliminate the CAS deficiency of allowing ABA. The main drawbacks for using version tags is the fact that a large number of the current hardware architectures, such as the majority of real-time embedded systems [11], do not support complex atomic primitives such as CAS2, DCAS, LL/SC, and AOU. A synchronization scheme on such machines can rely only on the portable single-word CAS instruction.

In [75] Reinholtz offers a technique for applying version tags using a 32-bits single-word memory swap (*Known Solution 2*). Similarly to the AtomicStampedReference in the Java Concurrency Library, Reinholtz's Reference Counting Pointers (RCP) split a version counter into two half-words: a half-word used to store the control value (an integer version counter in RCP's case) and a half-word used as a version tag. The limitations of this approach are: *a.* there is a limit of maximum $2^{16} - 1$ writes for each control location, and *b.* the range of values that can be represented in a control value is significantly decreased (by a factor of 2^{16}). To guarantee the uniqueness invariant of a control value of type pointer in a concurrent system with dynamic memory usage, we face an extra challenge: even if we write a pointer value no more than once in a given control location, the memory allocator might reuse the address of an already freed object (A_i) and pose an ABA hazard. To prevent this scenario, all control values of pointer type must be guarded by a concurrent nonblocking garbage collection scheme such as Hazard Pointers [52] (that uses a list of hazard pointers per thread) or Herlihy et al.'s Pass The Buck algorithm [55] (that utilizes a dedicated thread to periodically reclaim unguarded objects). While enhancing the safety of a concurrent algorithm (when needed), the application of a complementary garbage collection mechanism might come at a significant performance cost (see Section H for details).

B. The Descriptor Object

Linearizability [8] is a correctness condition for concurrent objects: a concurrent operation is linearizable if it appears to execute instantaneously in a given point of time τ_{lin} between the time τ_{inv} of its invocation and the time τ_{end} of its completion. The literature often refers to τ_{lin} as a *linearization point.* The implementations of many nonblocking data structures require the update of *two or more memory locations* in a linearizable fashion [7], [24]. The engineering of such operations (e.g. push_back and resize in a dynamically resizable array) is critical and particularly challenging in a CAS-based design. A common programming technique applied to guarantee the linearizability requirements for such operations is the use of a *Descriptor Object (δ object)* [7], [24]. The pseudocode in Algorithm 18 shows the two-step execution of a Descriptor object. In our nonblocking design, a Descriptor object stores three types of information:

(a) Global data describing the state of the shared container ($\upsilon\delta$), e.g. the size of a dynamically resizable array [7].

(b) A record of a pending operation on a given memory location. We call such a record requesting an update at a shared location L_i from an old value, old_val, to a new value, new_val, a *Write Descriptor ($\omega\delta$)*. The shortcut notation we use is $\omega\delta$ @ L_i : old_val $\rightarrow$ new_val. The fields in the Write Descriptor object store the target location as well as the old and the new values.

(c) A boolean value indicating whether $\omega\delta$ contains a pending write operation that needs to be completed.

The use of a descriptor allows an interrupting thread help the interrupted thread complete an operation rather than wait for its completion. As shown in Algorithm 18, the technique is used to implement, using only two CAS instructions, a linearizable update of two memory locations: 1. a reference to a Descriptor object (data type pointer to δ stored in a location L_δ) and 2. an element of type value_type stored in L_i. In Step 1, Algorithm 18, we perform a CAS-based speculation of a shared location L_δ that contains a reference to a Descriptor object. The CAS-based speculation routine's purpose is to replace an existing Descriptor object with a new one. Step 1 executes in the following fashion:

1. We read the value of the current δ reference stored in L_δ (line 3).
2. If the current δ object contains a pending operation, we need to help its completion (lines 4-5).
3. We record the current value, A_i, in location L_i (line 6) and compute the new value, B_i, to be stored in L_i (line 7).
4. A new $\omega\delta$ object is allocated on the heap, initialized (by calling $f_{\omega\delta}$), and its fields Target, OldValue, and NewValue are set (lines 8-11).

5. Any other data stored in a Descriptor object must be computed (by calling $f_{v\delta}$). Such data might be a shared element or a container's size (line 12).

6. A new Descriptor object is initialized containing the new Write Descriptor object and the new descriptor's data. The new descriptor's *pending operation* flag (WDpending) is set to true (lines 13-14).

7. We attempt a swap of the old Descriptor object with the new one (line 15). Should the CAS fail, we know that there is another process that has interrupted us and meanwhile succeeded to modify L_δ and progress. We need to go back at the beginning of the loop and repeat all the steps. Should the CAS succeed, we proceed with Step 2 and perform the update at L_i.

The size of a Descriptor object is larger than a memory word. Thus, we need to store and manipulate a Descriptor object through a reference. Since the control value of Step 1 stores a pointer to a Descriptor object, to prevent ABA, all references to descriptors must be memory managed by a safe nonblocking garbage collection scheme. We use the prefix μ for all variables that require safe memory management. In Step 2 we execute the Write Descriptor, WD, in order to update the value at L_i. Any interrupting thread (after the completion of Step 1) detects the pending flag of $\omega\delta$ and, should the flag's value be still positive, it proceeds to executing the requested update $\omega\delta$ @ $L_i : A_i \rightarrow B_i$. There is no need to execute a CAS-based loop and the call to a single CAS is sufficient for the completion of $\omega\delta$. Should the CAS from Step 2 succeed, we have completed the two-step execution of the Descriptor object. Should it fail, we know that there is an interrupting thread that has completed it already. A false positive execution of the CAS operation from Step 2 can lead to a *spurious write* of B_i into L_i, violate the operation's linearizability guarantee, and corrupt the semantics of a nonblocking algorithm. In the following sections (Sections C, F, G) we discuss a number of possible techniques that help us avoid ABA in this scenario.

Algorithm 18 Two-step execution of a δ object

```
1: Step 1: place a new descriptor in $L_\delta$
2: repeat
3:    $\delta$ $\mu$OldDesc = ^$L_\delta$
4:    if $\mu$OldDesc.WDpending == true then
5:       execute $\mu$OldDesc.WD
6:    value_type $A_i$ = ^$L_i$
7:    value_type $B_i$ = fComputeB
8:    $\omega\delta$ WD = $f_{\omega\delta}()$
9:    WD.Target = $L_i$
10:    WD.OldElement = $A_i$
11:    WD.NewElement = $B_i$
12:    $\upsilon\delta$ DescData = $f_{\upsilon\delta}()$
13:    $\delta$ $\mu$NewDesc = $f_\delta$(DescData, WD)
14:    $\mu$NewDesc.WDpending = true
15: until CAS($L_\delta$, $\mu$OldDesc, $\mu$NewDesc) == $\mu$NewDesc
16:
17: Step 2: execute the write descriptor
18: if $\mu$NewDesc.WDpending then
19:    CAS(WD.Target, WD.OldElement, WD.NewElement) == WD.NewElement
20:    $\mu$NewDesc.WDPending = false
```

C. Known ABA Detection and Avoidance Techniques, Part II

A known approach for avoiding a false positive execution of the Write Descriptor from Algorithm 18 is the application of *value semantics* for all values of type value_type (*Known Solution 3*). As discussed in [50] and [7], an ABA avoidance scheme based on value semantics relies on:

a. *Extra level of indirection*: all values are stored in shared memory indirectly through pointers. Each write of a given value v_i to a shared location L_i needs to allocate on the heap a new reference to v_i (η_{v_i}), store η_{v_i} into L_i, and finally safely delete the pointer value removed from L_i. If the value type of v_i is pointer then η_{v_i}'s type is pointer to pointer .

b. *Nonblocking garbage collection (GC):* all references stored in shared memory (such as η_{v_i}) need to be safely managed by a nonblocking garbage collection scheme (e.g. Hazard Pointers, Pass The Buck).

As reflected in our performance test results (Section H), the usage of both an extra level of indirection as well as the heavy reliance on a nonblocking GC scheme for managing the Descriptor objects *and* the references to value_type objects is very expensive with respect to the space and time complexity of a nonblocking algorithm. However, the use of value semantics is the *only known approach* for ABA avoidance in the execution of a Write Descriptor object. In Section F we present a 3-step execution approach that helps us eliminate ABA, avoid the need for an extra level of indirection, and reduce the usage of the computationally expensive GC scheme.

D. Criteria

To provide a practical and generic solution to the ABA problem without incurring a prohibitive cost to the lock-free application, our search for a solution has been guided by the following design criteria:

(a) *Complexity and Semantics Preservation:* an ABA avoidance scheme should not incur extra algorithmic complexity and should preserve the application's nonblocking guarantees and correctness conditions. For example, a shared vector's tail operations have a complexity of $O(1)$ that must be preserved.

(b) *Dynamic and Open Memory Usage:* ability to support dynamic and open memory [45] usage at a minimal cost.

(c) *Fast Performance:* an ABA prevention scheme should make minimal usage of expensive garbage collection and should not prevent disjoint-access parallelism. Some lock-free container's implementations provide a combination of lock-free (δ-modifying) and wait-free (non-δ-modifying) operations [7]. Wait-free operations are fast and progress regardless the contention on the shared memory. Preserving the wait-free semantics of such operations might be critical to the container's performance. While sometimes necessary to use, the application of GC schemes must be limited to an absolute minimum.

(d) *Portability:* we assume the availability of single-word atomic read, write, and CAS instructions. We consider solutions based on multi-word CAS, Alert-On-Update [74], or LL/SC to be platform-specific.

(e) *Unlimited Data Usage:* we prefer to avoid placing constraints on the usage of the data values. We assume that the data values stored in a shared container need not be unique, there is no restriction on the range of values (imposed by the ABA prevention algorithm), data elements can be written and read an arbitrary number of times to/from any location, and there is no restriction on the number of writer threads.

(f) *No Extra Levels of Indirection:* a famous quote by David Wheeler states: ”Any problem in computer science can be solved with another layer of indirection, but that usually will create another problem” [4]. As illustrated in Section H, the application of an extra level of indirection suffers performance penalties, leads to heavy usage of the costly GC scheme, increases the complexity of the nonblocking algorithm, and is difficult to integrate in an already existing nonblocking implementation.

E. Nonblocking Concurrent Semantics

The use of a Descriptor object provides the programming technique for the implementation of some of the complex nonblocking operations in a shared container, such as the push_back, pop_back, and reserve operations in a shared vector [7]. The use and execution of a Write Descriptor guarantees the linearizable update of two or more memory locations.

Definition 3: *An operation whose success depends on the creation and execution of a Write Descriptor is called an $\omega\delta$-executing operation.*

The operation push_back of a shared vector (Chapter III, [7]) is an example of an $\omega\delta$-executing operation. Such $\omega\delta$-executing operations have *lock-free* semantics and the progress of an individual operation is subject to the contention on the shared location L_i (under heavy contention, the body of the CAS-based loop from Step 1, Algorithm 18 might need to be re-executed). For a shared vector, operations such as pop_back do not need to execute a Write Descriptor object [7]. Their progress is dependent on the state of the global data stored in the Descriptor object, such as the size of a container.

Definition 4: *An operation whose success depends on the state of the $\upsilon\delta$ data stored in the Descriptor object is a δ-modifying operation.*

A δ-modifying operation, such as pop_back, needs only update the shared global data (the size of type $\upsilon\delta$) in the Descriptor object (thus pop_back seeks an atomic update of only one memory location: L_δ). Since an $\omega\delta$-executing operation by definition always performs an exchange of the entire Descriptor object, every $\omega\delta$-executing operation is also δ-modifying. The semantics of a δ-modifying operation are *lock-free* and the progress of an individual operation is determined by the interrupts by other δ-modifying operations. An $\omega\delta$-executing operation is also δ-modifying but as is the case with pop_back, not all δ-modifying operations are $\omega\delta$-executing. Certain operations, such as the random access read and write in a vector, do not need to access the Descriptor object and progress regardless of the state of the descriptor (Chapter III). Such operations are non-δ-modifying and have *wait-free* semantics (thus no delay if there is contention at L_δ).

Definition 5: *An operation whose success does not depend on the state of the Descriptor object is a non-δ-modifying operation.*

1. Concurrent Operations

When two δ-modifying operations (O_{δ_1} and O_{δ_2}) are concurrent, according to Algorithm 18, O_{δ_1} precedes O_{δ_2} in the linearization history if and only if O_{δ_1} completes Step 1, Algorithm 18 prior to O_{δ_2}.

Definition 6: *We refer to the instant of successful execution of the global descriptor exchange at L_δ (line 15, Algorithm 18) as τ_δ.*

Definition 7: *A point in the execution of a δ object that determines the order of an $\omega\delta$-executing operation acting on location L_i relative to other writer operations acting on the same location L_i, is referred to as the $\lambda\delta$-point ($\tau_{\lambda\delta}$) of a Write Descriptor.*

The order of execution of the $\lambda\delta$-points of two concurrent $\omega\delta$-executing operations determines their order in the linearization history. The $\lambda\delta$-point does not necessarily need to coincide with the operation's linearization point, τ_{lin}. As illustrated in Chapter III, τ_{lin} can vary depending on the operations' concurrent interleaving. The linearization point of a shared vector's $\omega\delta$-modifying operation can be any of the three possible points: *a.* some point after τ_δ at which some operation reads data form the Descriptor object, *b.* τ_δ or *c.* the point of execution of the Write Descriptor, τ_{wd} (the completion of Step 2, Algorithm 18). The core rule for a linearizable operation is that it must appear to execute in a single instant of time with respect to other concurrent operations. The linearization point need not correspond to a single fixed instruction in the body of the operation's implementation and can vary depending on the interrupts the operation experiences. In contrast, the $\lambda\delta$-point of an $\omega\delta$ object corresponds to a single instruction in the objects's implementation. In the pseudo code in Algorithm 18 $\tau_{\lambda\delta} \equiv \tau_\delta$.

Table 12. ABA Occurrence in the Execution of a Descriptor Object

Step	*Action*
Step 1	O_{δ_1}: τ_{read_δ}
Step 2	O_{δ_1}: τ_{access_i}
Step 3	O_{δ_1}: τ_δ
Step 4	O_{δ_2}: τ_{read_δ}
Step 5	O_{δ_1}: τ_{wd}
Step 6	O_i: τ_{write_i}
Step 7	O_{δ_2}: τ_{wd}

Let us designate the point of time when a certain δ-modifying operation reads the state of the Descriptor object by τ_{read_δ}, and the instants when a thread reads a value from and writes a value into a location L_i by τ_{access_i} and τ_{write_i}, respectively. Table 12 demonstrates the occurrence of ABA in the execution of a δ object with two concurrent δ-modifying operations (O_{δ_1} and O_{δ_2}) and a concurrent write, O_i, to L_i. We assume that the δ object's implementation follows Algorithm 18. The execution of O_{δ_1}, O_{δ_2}, and O_i proceeds in the following manner:

(1) O_{δ_1} reads the state of the current δ object as well as the current value at L_i, A_i (Steps 1-2, Table 12). Next, O_{δ_1} proceeds with instantiating a new δ object and replaces the old descriptor with the new one (Step 3, Table 12).

(2) O_{δ_1} is interrupted by O_{δ_2}. O_{δ_2} reads L_δ and finds the WDpending flag's value to be true (Step 4, Table 12).

(3) O_{δ_1} resumes and completes the execution of its δ object by storing B_i into L_i (Step 5, Table 12).

(4) An interrupting operation, O_i, writes the value A_i into L_i (Step 6, Table 12).

(5) O_{δ_2} resumes and executes $\omega\delta$ it has previously read, the $\omega\delta$'s CAS falsely succeeds (Step 6, Step 12).

The placement of the $\lambda\delta$-point plays a critical role for achieving ABA safety in the implementation of an $\omega\delta$-executing operation. The $\lambda\delta$-point in Table 12 guarantees that the $\omega\delta$-executing operation O_{δ_1} completes before O_{δ_2}. However, at time τ_{wd} when O_{δ_2} executes the Write Descriptor, O_{δ_2} has no way of knowing whether O_{δ_1} has completed its update at L_i or not. Since O_{δ_1}'s $\lambda\delta$-point $\equiv \tau_{\delta}$, the only way to know about the status of O_{δ_1} is to read L_{δ}. Using a single-word CAS operation prevents O_{δ_2} from atomically checking the status of L_{δ} and executing the update at L_i.

Definition 8: *A concurrent execution of one or more non-$\omega\delta$-executing δ-modifying operations with one $\omega\delta$-executing operation, O_{δ_1}, performing an update at location L_i is ABA-free if O_{δ_1}'s $\lambda\delta$-point $\equiv \tau_{access_i}$. We refer to an $\omega\delta$-executing operation where its $\lambda\delta$-point $\equiv \tau_{access_i}$ as a $\lambda\delta$-modifying operation.*

Assume that in Table 12 the O_{δ_1}'s $\lambda\delta$-point $\equiv \tau_{access_i}$. As shown in Table 12, the ABA problem in this scenario occurs when there is a hazard of a spurious execution of O_{δ_1}'s Write Descriptor. Having a $\lambda\delta$-modifying implementation of O_{δ_1} allows any non-$\omega\delta$-executing δ-modifying operation such as O_{δ_2} to check O_{δ_1}'s progress while attempting the atomic update at L_i requested by O_{δ_1}'s Write Descriptor. Our *3-step descriptor execution* approach, described in Section F, offers a solution based on Definition 8. In an implementation with two or more concurrent $\omega\delta$-executing operations, each $\omega\delta$-executing operation must be $\lambda\delta$-modifying in order to eliminate the hazard of a spurious execution of an $\omega\delta$ that has been picked up by a collaborating operation. To effectively avoid the ABA hazard at L_i during a descriptor-based linearizable update of L_δ and L_i (see Algorithm 18), we generalize two fundamental strategies:

(a) Guarantee that a Write Descriptor created by O_{δ_1}, or any other $\omega\delta$-executing operation, succeeds at most once. We refer to such a δ object as a *once-execute-descriptor*. *Definition* 8 offers the condition leading to a solution of this type. In our example in Table 12, a *once-execute-descriptor* strategy would cause the attempt to re-execute the Write Descriptor by O_{δ_1} (Step 7, Table 12) or by any other operation to fail. Our 3-step δ execution approach presented in Section F is one possible way of implementing a *once-execute-descriptor*.

(b) Guarantee that no concurrent interleaving of operations can lead to a write of a value posing ABA hazard (such as B_i in Table 12) at L_i. Relying on a methodology that employs unique values, such as *Known Solution* 1, is an approach of this type. Requiring uniqueness typing for ABA prevention is an overkill. The guarantee we need is that no thread can restore an old value A_i in a shared location L_i while there is an alive $\omega\delta$ object in the system requesting $\omega\delta$ @ L_i : A_i $\rightarrow$ $\mathsf{any\_value}_i$. Modern mainstream programming languages do not yet explicitly support concurrency and lack the tools to express and enforce such a concurrent and dynamic correctness condition.

F. Implementing a $\lambda\delta$-modifying Operation

In Algorithm 19 we suggest a design strategy for the implementation of a $\lambda\delta$-modifying operation. Our approach is based on a 3-step execution of the δ object. While similar to Algorithm 18, the approach shown in Algorithm 19 differs by executing a fundamental additional step: in Step 1 we store a pointer to the new descriptor in

L_i prior to the attempt to store it in L_δ in Step 2. Since all δ objects are memory managed, we are guaranteed that no other thread would attempt a write of the value μNewDesc in L_i or any other shared memory location. The operation is $\lambda\delta$-modifying because, after the new descriptor is placed in L_i, any interrupting writer thread accessing L_i is required to complete the remaining two steps in the execution of the Write Descriptor. However, should the CAS execution in Step 2 (line 28) fail, we have to unroll the changes at L_i performed in Step 1 by restoring L_i's old value preserved in WD.OldElement (line 20) and retry the execution of the routine (line 21). To implement Algorithm 19, we have to be able to distinguish between objects of type value_type and δ. A possible solution is to require that all value_type variables are pointers and all pointer values stored in L_i are aligned with the two low-order bits cleared during their initialization. That way, we can use the two low-order bits for designating the type of the pointer values. Subsequently, every read must check the type of the pointer obtained from a shared memory location prior to manipulating it. Once an operation succeeds at completing Step 1, Algorithm 19, location L_i contains a pointer to a δ object that includes both: L_i's previous value of type value_type and a Write Descriptor WD that provides a record for the steps necessary for the operation's completion. Any non-δ-modifying operation, such as a random access read in a shared vector, can obtain the value of L_i (of type value_type) by accessing WD.OldElement (thus going through a *temporary* indirection) and ignore the Descriptor object temporarily stored at L_i. Upon the success of Step 3, Algorithm 19, the temporary level of indirection is eliminated. Such an approach would preserve the wait-free execution of a non-δ-modifying operation. The $\omega\delta$ data type needs to be amended to include a field TempElement (line 9, Algorithm 19) that records the value of the temporary δ pointer stored in L_i. The cost of the $\lambda\delta$ operation is 3 CAS executions to achieve the linearizable update of two shared memory locations (L_i and

L_δ).

Algorithm 19 Implementing a $\lambda\delta$-modifying operation through a three-step execution of a δ object

```
1: *Step 1:* place a new descriptor in $L_i$
2: value_type $B_i$ = fComputeB
3: value_type $A_i$
4: $\omega\delta$ WD = $f_{\omega\delta}()$
5: WD.Target = $L_i$
6: WD.NewElement = $B_i$
7: $\upsilon\delta$ DescData = $f_{\upsilon\delta}()$
8: $\delta$ $\mu$NewDesc = $f_\delta$(DescData, WD)
9: WD.TempElement = &NewDesc
10: $\mu$NewDesc.WDpending = true
11: **repeat**
12:    $A_i$ = ^$L_i$
13:    WD.OldElement = $A_i$
14: **until** CAS($L_i$, $A_i$, $\mu$NewDesc) == $\mu$NewDesc
15:
16: *Step 2:* place the new descriptor in $L_\delta$
17: bool unroll = false
18: **repeat**
19:    **if** unroll **then**
20:       CAS(WD.Target, $\mu$NewDesc, WD.OldElement)
21:       **goto** 3
22:    $\delta$ $\mu$OldDesc = ^$L_\delta$
23:    **if** $\mu$OldDesc.WDpending == true **then**
24:       execute $\mu$OldDesc.WD
25:    unroll = true
26: **until** CAS($L_\delta$, $\mu$OldDesc, $\mu$NewDesc) == $\mu$NewDesc
27:
28: *Step 3:* execute the Write Descriptor
29: **if** $\mu$NewDesc.WDpending **then**
30:    CAS(WD.Target, WD.TempElement, WD.NewElement) == WD.NewElement
31:    $\mu$NewDesc.WDPending = false
```

G. Alternative Solutions

We briefly mention two alternative approaches for ABA avoidance. While failing our solution criteria from Section D by imposing undesirable constraints or inflating the performance overhead in the general case, we believe these alternative techniques might prove helpful in certain application-specific scenarios where the imposed constraints are acceptable.

1. Enforcement of Usage Phases

In Chapter VI we suggested the effective elimination of ABA by restricting the use of a lock-free vector to two usage phases: a *growth phase* and an *operational phase*. A growth phase allows only push_back and random access read by all threads. An operational phase allows all operations (pop_back and the random access read and write) except push_back. This separation of reads and writes avoids the ABA hazardous interleaving of the vector's tail operations. The approach restricts the application of the vector. However, where applicable, it eliminates the ABA problem at a very low cost.

2. Serialize Contending Operations

Similar to the Predictive Log Synchronization approach suggested by Shalev and Shavit in [37], it is possible to serialize all contending writes and delegate them to a single dedicated writer thread. As mentioned in [37], such a design is very costly in the general case and is mostly suitable for applications that execute read-many-write-rarely operations.

H. Performance Evaluation

To evaluate the performance of the ABA-free programming techniques discussed in this work, we incorporated the presented ABA elimination approaches in the implementation of the nonblocking dynamically resizable array as presented in Chapter VI. Our test results indicate that the $\lambda\delta$ approach offers ABA prevention with performance comparable to the use of the platform-specific CAS2 instruction to implement version counting. This finding is of particular value to the engineering of some embedded real-time systems where the hardware does not support complex atomic primitives such as CAS2 [2]. We ran performance tests on an Intel IA-32 SMP machine with two 1.83GHz processor cores with 512 MB shared memory and 2 MB L2 shared cache running the MAC OS X operating system. In our performance analysis we compare:

(1) *$\lambda\delta$ approach*: the implementation of a vector with a $\lambda\delta$-modifying push_back and a δ-modifying pop_back. Table 13 shows that in this scenario the cost of push_back is three single-word CAS operations and pop_back's cost is one single-word CAS instruction.

(2) *All-GC approach*: the application of Known Solution 3 (Section C), namely the use of an extra level of indirection and memory management for each element. Because of its performance and availability, we have chosen to implement and apply Herlihy et al.'s Pass The Buck algorithm [55]. In addition, we use Pass The Buck to protect the Descriptor objects for all of the tested approaches.

(3) *CAS2-based approach:* the application of CAS2 for maintaining a reference counter for each element. A CAS2-based version counting implementation is easy to apply to almost any pre-existent CAS-based algorithm. While a CAS2-based solution is not portable and thus not meeting our goals as described in Section D, we believe that the approach is applicable for a large number of modern architectures. For this reason, it is included in our performance evaluation. In the performance tests, we apply CAS2 (and version counting) for updates at the shared memory locations at L_i and a single-word CAS to update the Descriptor object at L_δ.

Table 13 offers an overview of the shared vector's operations' relative cost in terms of number and type of atomic instructions invoked per operation.

Table 13. A Shared Vector's Operations Cost (Best Case Scenario)

ABA prevention approach / operation	push_back	pop_back	read_i	write_i
1. $\lambda\delta$ approach	3 CAS	1 CAS	atomic read	atomic write
2. All-GC approach	2 CAS + GC	1 CAS + GC	atomic read	atomic write + GC
3. CAS2-based approach	1 CAS2 + 1 CAS	1 CAS	atomic read	1 CAS2

Similarly to the evaluation of other lock-free algorithms [43], we designed our experiments by generating a workload of the various operations. We varied the number of threads, starting from 1 and exponentially increased their number to 64. Each thread executed 500,000 lock-free operations on the shared container. We measured the execution time (in seconds) that all threads needed to complete. Each iteration of every thread executed an operation with a certain probability (`push_back` (`+`), `pop_back` (`-`), random access `write` (`w`), random access `read` (`r`)). We show the performance graph for a distribution of `+`:40%, `-`:40%, `w`:10%, `r`:10% on Figure 15. Figure 16 demonstrates the performance results with less contention at the vector's tail, `+`:25%, `-`:25%, `w`:10%, `r`:40%. Figure 17 illustrates the test results with a distribution containing predominantly random access read and write operations, `+`:10%, `-`:10%, `w`:40%, `r`:40%. Figure 18 reflects our performance evaluation on a vector's use with mostly random access read operations: `+`:20%, `-`:0%, `w`:20%, `r`:60%, a scenario often referred to as the most common real-world use of a shared container [43]. The number of threads is plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis. According to the performance results, compared to the *All-GC* approach, the $\lambda\delta$ approach delivers consistent performance gains in all possible operation mixes by a large factor, a factor of at least 3.5 in the

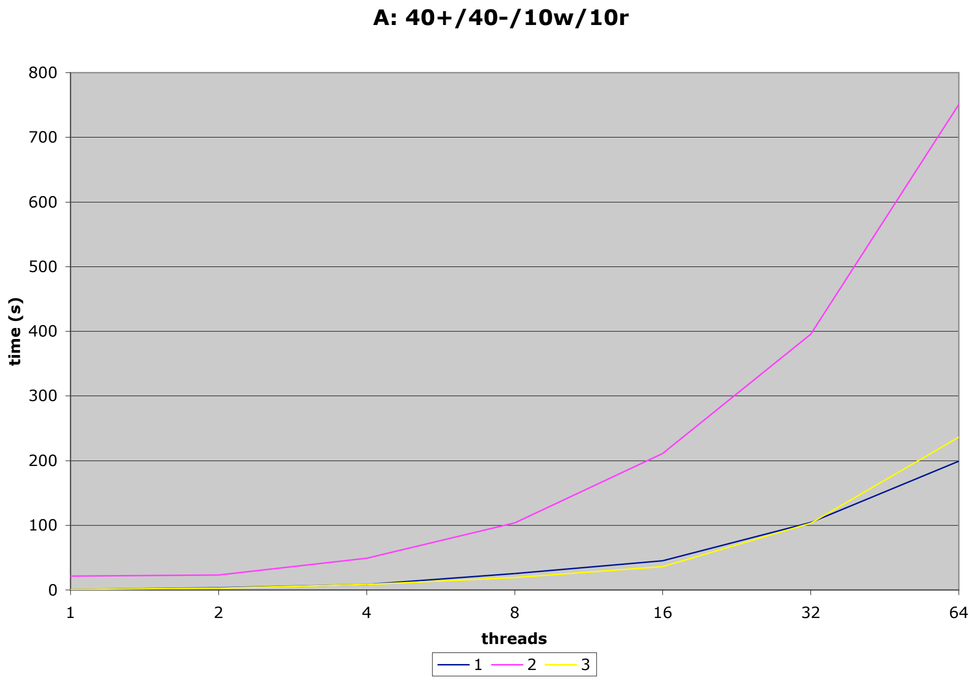


Fig. 15. ABA-free Synchronization Performance Results A

cases with less contention at the tail and a factor of 10 or more when there is a high concentration of tail operations. Once again we have observed that introducing an extra level of indirection and the necessity to memory manage each individual element with PTB (or an alternative memory management scheme) to avoid ABA comes with a pricy performance overhead. The $\lambda\delta$ approach offers an alternative by introducing the notion of a $\lambda\delta$-point and enforces it though a 3-step execution of the δ object. The application of version counting based on the architecture-specific CAS2 operation is the most commonly cited approach for ABA prevention in the

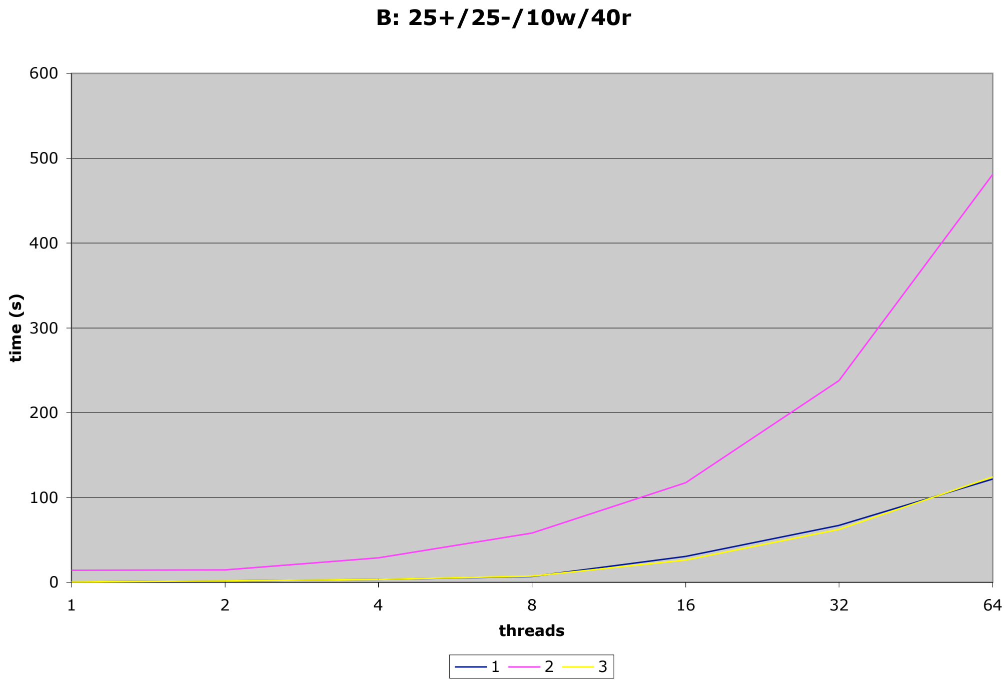


Fig. 16. ABA-free Synchronization Performance Results B

literature [50], [55]. Our performance evaluation shows that the $\lambda\delta$ approach delivers performance comparable to the use of CAS2-based version counting. CAS2 is a complex atomic primitive and its application comes with a higher cost when compared to the application of atomic write or a single-word CAS. In the performance tests we executed, we notice that in the scenarios where random access write is invoked more frequently (Figures 17 and 18), the performance of the CAS2 version counting approach suffers a performance penalty and runs slower than the $\lambda\delta$ approach by about 12% to 20%. The implementation of our $\lambda\delta$-modifying operation as shown in

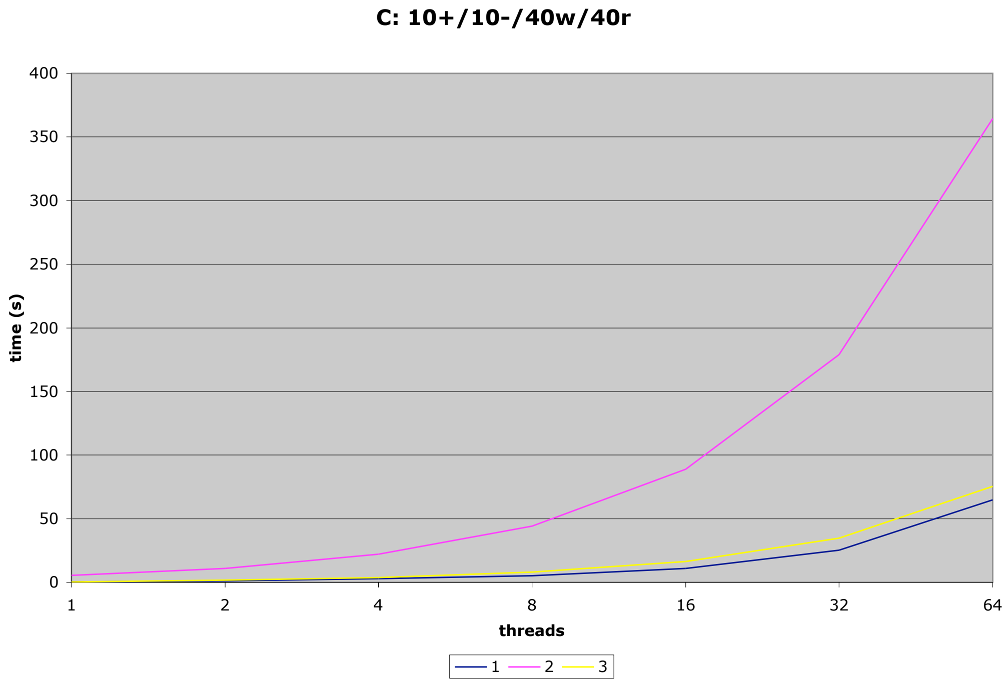


Fig. 17. ABA-free Synchronization Performance Results C

Algorithm 19 is similar to the execution of Harris et al.'s MCAS algorithm [42]. Just like our $\lambda\delta$-modifying approach, for an MCAS update of L_δ and L_i the cost of Harris et al.'s MCAS is at least 3 successful executions of the single-word CAS instruction. Harris et al.'s work on MCAS [42] brings forward a significant contribution in the design of lock-free algorithms, however, it lacks an analysis of the hazards of ABA and the way the authors manage to avoid it. According to our performance evaluation, the $\lambda\delta$ approach is a systematic, effective, portable, and generic solution for ABA avoidance. The $\lambda\delta$ scheme does not induce a performance penalty when compared

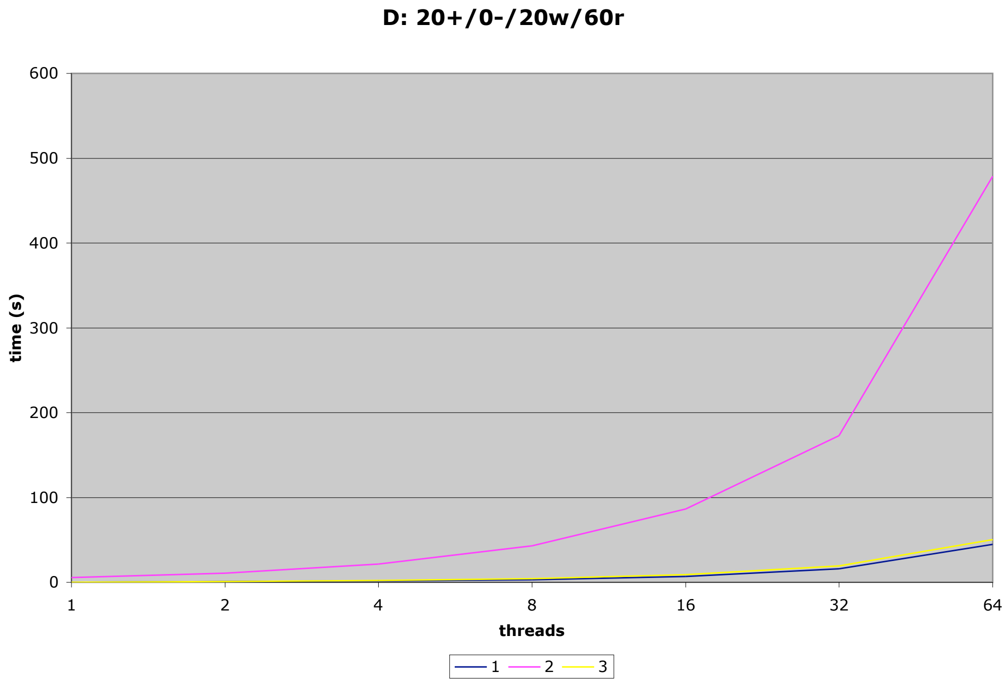


Fig. 18. ABA-free Synchronization Performance Results D

to the architecture-specific application of CAS2-based version counting and offers a considerable performance gain when compared to the use of *All-GC*.

In this chapter we studied the ABA problem and the conditions leading to its occurrence in a Descriptor-based lock-free linearzibale design. We offered a systematic and generic solution, called the $\lambda\delta$ approach, that outperforms by a significant factor the use of garbage collection for the safe management of each shared location and offers speed of execution comparable to the application of the architecture-specific CAS2 instruction used for version counting. Having a practical alternative to the application of the architecture-specific CAS2 is of particular importance to the design of some modern embedded systems such as Mars Science Laboratory. We defined a condition for ABA-free synchronization that allows us to reason about the ABA safety of a lock-free algorithm. We presented a practical, generic, and portable implementation of the $\lambda\delta$ approach and evaluated it by integrating the $\lambda\delta$ technique into a nonblocking shared vector. The literature does not offer a detailed analysis of the ABA problem and the general techniques for its avoidance in a lock-free linearizable design. At the present moment of time, the challenges of eliminating ABA are left to the ingenuity of the software designer. The goal of this chapter is to deliver a guide for ABA comprehension and prevention in Descriptor-based lock-free linearizable algorithms. In our future work, we plan to utilize a model-checker [76] to express the $\lambda\delta$ condition and be able to formally verify the ABA-freedom of nonblocking data structures and algorithms.

CHAPTER VIII

VERIFICATION AND SEMANTIC PARALLELIZATION OF GOAL-DRIVEN AUTONOMOUS SOFTWARE

In this chapter we describe the design, implementation, and practical application of our framework for verification and semantic parallelization of real-time C++ within JPL's MDS Framework (Figure 19). The input to the framework is the MDS mission planning and execution module that is based on the definition of Temporal Constraint Networks (Chapter II). At the core of the most recent implementations at JPL of this critical module is an optimized iterative algorithm for the real-time propagation of temporal constraints, developed and described by Lou [77]. Constraint propagation poses performance challenges and speed bottlenecks due to the algorithm's frequent execution and the necessary real-time updates of the goal network's topology. The end goal of our work is, given the implementation of the optimized iterative propagation scheme and the topology of a particular goal network, to establish the correctness of the core TCN semantic invariants (see Chapter II) and *automatically* derive an implementation that can be executed concurrently on one of the JPL's experimental testbeds for accelerated testing [20]. Our approach for achieving concurrent execution is based on the idea of identifying Time Phases within a goal network, which allow the semantic parallelization of the constraint propagation algorithm. In this work, we define *semantic parallelization* as the thread-safe concurrent execution of an algorithm (whose operation is dependent on shared data), derived from the application's semantics and invariants. In the following sections we describe how we reach our goal of verification and semantic parallelization of the mission planning and control module by constructing and executing a formal verification model in Alloy [64] that represents the implementation's core semantics and functionality. We refine a formal modeling and analysis methodology, initially suggested by Rouquette [78], that helps us analyze the logical properties of the goal network model and automatically derive a meta-model for our parallel solution.

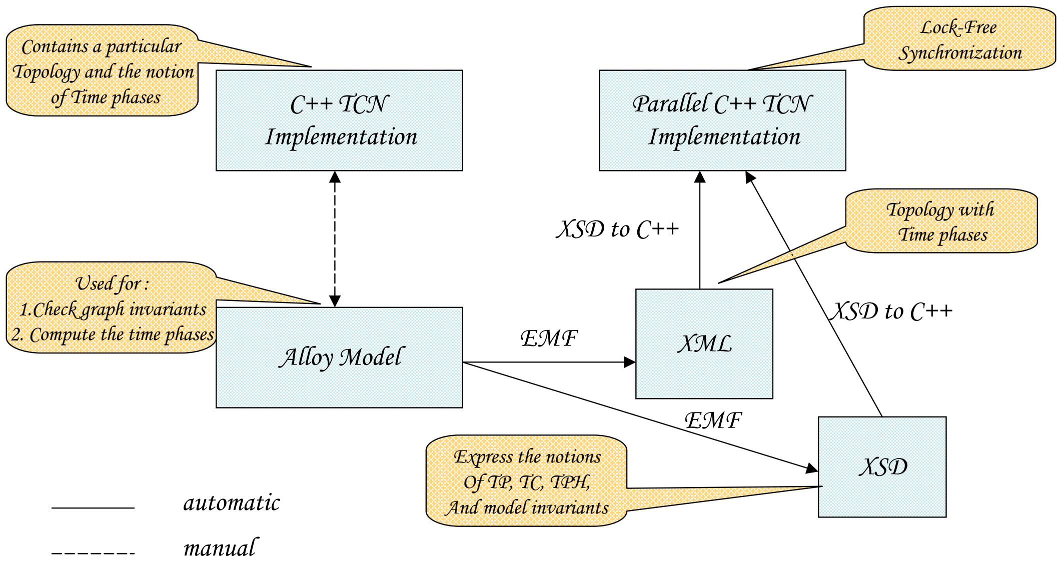

Contains a particular Topology and the notion of Time phases
C++ TCN Implementation
Parallel C++ TCN Implementation
Lock-Free Synchronization
XSD to C++
Topology with Time phases
Used for : 1.Check graph invariants 2. Compute the time phases
Alloy Model
EMF
XML
XSD to C++
EMF
Express the notions Of TP, TC, TPH, And model invariants
XSD
automatic
manual

Fig. 19. A Framework for Verification and Semantic Parallelization

A. The MDS Architecture

Mission Data System (MDS) [9] is the Jet Propulsion Laboratory's framework for designing and implementing complete end-to-end data and control autonomous flight systems. The framework focuses on the representation of three main software architecture principles (defining the highest Δ_{SA} level of development in the certification framework):

(1) System control: a state-based control architecture with explicit representation of controllable state [79].

(2) Goal-oriented operation: control intent is expressed by defining a set of goals and a goal network [80].

(3) Layered data management: an integrated data management and transport protocols [5].

In MDS a state variable provides access to the data abstractions representing the physical entities under control over a continuous period of time, spanning from the distant past to the distant future. In other words, a state variable is a programming (Δ_{PL}) and a modeling (Δ_{MT}) representation of a set of constraint (κ) certification variables ($S_{\kappa_{sv}}$) expressed in the library level of abstraction (Λ). As explained by Wagner [5], the implementation's intent is to define a goal timeline overlapping or coinciding with the state variables' timeline. This means that the implementation must rely on an algorithm (abstraction level Θ) that transforms the engineers intent together with $S_{\kappa_{sv}}$ into a set of invariants $S_{\eta_{sv}}$ and a set of guarantees $S_{\gamma_{sv}}$ and establish the validity and consistency of all $\eta_i \in S_{\eta_{sv}}$ and all $\gamma_i \in S_{\gamma_{sv}}$ so that the system's operations corresponds with its Rt, Sc, Em, and Au behavior.

Computing the invariants (a set of $S_{\eta_{gi}}$) necessary for achieving a goal (any $\gamma_i \in S_{\gamma_{sv}}$) might require the lookup of past states as well as the computation of projected future states. MDS employs the concept of goals to represent control intent. Goals are expressed as a set of temporal constraints. Each state variable is associated with exactly one state estimator whose function is to collect all available data and compute a projection of the state value and its expected transitions. Control goals are considered to be those that are meant to control external physical states. Knowledge goals are those goals that represent the constraints on the software system regarding a property of a state variable. Not all states are known at all time. The most trivial knowledge goal is the request for a state to be known, thus enabling its estimator. A data state is defined as the information regarding the available state and goal data and its storage format and location. The MDS framework considers data states an integral part of the control system rather than a part of the system under control. There are dedicated state variables representing the data states. In addition, data states can be controlled through the definition of data goals. A data state might store information such as location, formatting, compression, and transport intent and status of the data. A data state might not be necessary for every state variable. In a simple control system where no telemetry is necessary, the state variable implementation might as well store the information regarding the variable's value history and its extrapolated states.

The representation of the data states and the data management in MDS is implemented in the Data Management Service module [5]. The problem of data management in an embedded control system (often requiring the satisfaction of Em, Rt, Au, and Sc -driven certification requirements at the same time) is one of translating the intent of remote operations into actions and then safely returning the observed information. The system should be robust to the extent of overcoming possible communication loss, hardware failures or a system reboot. The resource constraint of an embedded control systems (its collection of κ variables) dictate that command-oriented control systems typically do not retain information specific to the intent of the observations. In addition, telemetry systems process and transport data in unlabeled packages where the scientific data is often mixed with other data. In this context, Wagner argues that it is of significant importance to address the challenges of providing uniform models for managing the flow of observation and control data. The MDS Data Management Service Library implements a Catalog for organizing the storage of physical observations in terms of storage products. Its functionality is responsible for the remote transport of data products with respect to the behavior of other spacecraft components. According to the current lock-based Catalog design, locks are applied in a complex manner within the inheritance hierarchy that leads to an exponential increase of the verification state space.

To achieve higher reliability (expressed as a set of safety invariants $S_{\eta_{safety}}$) and enhance the performance (measured in terms of speed of execution in the ϵ_{exe} variable), we consider the application of *lock-free synchronization* (see Chapter III). Lock-free algorithms rely on a set of atomic primitives supported by the hardware architecture. This means that a nonblocking technique represents an algorithmic (Θ) or library (Λ) solution to an important Sc problem, namely avoidance of the hazards of deadlock, livelock, and priority inversion (expressed as three separate Sc invariants: η_{lvlock}, η_{delock}, and η_{pinv}) while at the same time offering a significant performance boost (measured in ϵ_{exe}). The application of a library of nonblocking algorithms shifts the complexity of engineering shared data access from the user's source into the lock-free library's implementation. Thus lock-free programming techniques can often help increase the comprehensibility of the concurrent interactions in the user's implementation. In the process of creating a parallel network of temporal constraints (by utilizing nonblocking synchronization), we measure the increased simplicity of the code (in contrast to the application of mutual exclusion) with the certification variable σ_{ptcn}.

Practical nonblocking programming techniques stand at a development level Δ_{PL} and when used properly help in assuring safe and efficient access to shared data (in a concurrent system defined by the system's Δ_{MC}) and their semantics and implementation is directly related to the atomic primitives available by the system's Φ level (such as the availability of atomic primitives like Compare-And-Swap or Double-Compare-And-Swap). In hardware platforms that do not provide complex atomic primitives (involving the atomic update of more than a single-word location), the implementation of lock-free algorithms is CAS-based. Such systems impose yet another important invariant η_{aba} where the programmer must eliminate the possibility of occurrence of the ABA problem (Chapter VII).

One approach to eliminating ABA is to strictly define the semantic usage pattern of a nonblocking algorithm (meaning that not all operations might be total at all time). Such usage rules are another example of a transformation of an invariant (η_{aba}) into a set of guarantees ($S_{\gamma_{aba}}$) variables that need to be satisfied. One possibility is the application of static analysis (Chapter VI) that can check for a hazardous interleaving of the concurrent processes. In such a scenario the ABA problem (η_{aba}) is resolved by the application of ΔAT development tools.

B. TCN Constraint Propagation

A classic solution to the problem of constraint propagation in TCN is the direct application of Floyd-Warshall's all-pairs-shortest-path algorithm [59], offering a complexity of $O(N^3)$, where N is the number of time points in the TCN topology. Since, by definition, the goal of the TCN propagation algorithm is to compute the real-time values of the network's temporal constraints, the algorithm is frequently executed and, given the massive scale of a real world goal network, can cause significant bottleneck for the overall system's performance. Lou [77] derives a TCN propagation scheme with a complexity close to linear. Lou's TCN propagation is based on the concept of alternating forward and backward propagation passes. A forward pass updates the time interval at each time point by considering only its incoming temporal constraints (Algorithm 20). Similarly, a backward pass recomputes the time windows at each time point by considering only its outgoing temporal constraints (Algorithm 21). The scheme utilizes a shared container, named a *propagation queue*, to keep track of all time points whose successor time points' windows are about to be updated next (during a forward pass) and all time points whose predecessor time points' windows are about to be updated next (during a backward pass). A forward pass begins by selecting all time points with no predecessors and inserts them into the propagation queue. A backward pass begins by selecting all time points with no successors and inserts them into the propagation queue. Each iteration is carried out until:

(a) An iteration completes without updating any temporal constraints (thus indicating that there are no more updates to be performed during the pass). In this case, the TCN topology is considered to be *temporally consistent*.

(b) The iteration has stumbled upon a time window of negative value and the algorithm terminates with the outcome of having a temporally inconsistent network.

As stated by Lou [77], prior to the execution of the optimized propagation scheme, we need to guarantee the validity of the core TCN invariants for the topology of the particular goal network. For example, the propagation scheme operates under the assumption that the goal network graph is cycle free. Should there be cycles, the propagation would enter into an endless loop.

Algorithm 20 Forward Pass. Arguments: a reference to the time point about to be updated (tp) and a reference to the global data structure recording the state updates (vstate).

```
1: $\mathsf{min}_{\mathsf{tmp}} \leftarrow$ tp.min
2: $\mathsf{max}_{\mathsf{tmp}} \leftarrow$ tp.max
3: for $j = 0$ to tp.preds_size do
4:    $\mathsf{min}_{\mathsf{tmp}} \leftarrow$ std::max($\mathsf{min}_{\mathsf{tmp}}$, tp.preds[j].pred.min + tp.preds[j].min)
5:    $\mathsf{max}_{\mathsf{tmp}} \leftarrow$ std::min($\mathsf{max}_{\mathsf{tmp}}$, tp.preds[j].pred.max + tp.preds[j].max)
6: if tp.min! = $\mathsf{min}_{\mathsf{tmp}}$ then
7:    ASSERT ( tp.min < $\mathsf{min}_{\mathsf{tmp}}$ )
8:    tp.min $\leftarrow \mathsf{min}_{\mathsf{tmp}}$
9:    vstate.aIncr(vstate.count) {atomically increment the state vector's counter}
10: if tp.max! = $\mathsf{max}_{\mathsf{tmp}}$ then
11:    ASSERT ( tp.max > $\mathsf{max}_{\mathsf{tmp}}$ )
12:    tp.max $\leftarrow \mathsf{max}_{\mathsf{tmp}}$
13:    vstate.aIncr(vstate.count) {atomically increment the state vector's counter}

14: return !($\mathsf{min}_{\mathsf{tmp}} > \mathsf{max}_{\mathsf{tmp}}$)
```

C. Modeling, Formal Verification, and Automatic Parallelization

Alloy [64] is a lightweight formal specification and verification tool for the automated analysis of user-specified invariants on complete or partial models. The Alloy Analyzer is implemented as a front-end, performing the role of a model-finder, to a boolean SAT-solver. Formal verification and modeling of JPL's flight software has been previously demonstrated to be effective and successful by Holzmann [76]. We use the Alloy specification language [64] to formally represent and check the semantics of the temporal constraint networks library (Algorithm 22) and its main invariants (Algorithm 23). In our C++ goal networks implementation we have applied generic programming techniques and concepts [69], so that we can maintain a higher level of expressiveness. As a result we have achieved a significant similarity in the way the main TCN notions and invariants are expressed in our actual implementation and the Alloy verification models.

Algorithm 21 Backward Pass. Arguments: a reference to the time point about to be updated (tp) and a reference to the global data structure recording the state updates (vstate).

```
1: $\mathsf{min_{tmp}} \leftarrow$ tp.min
2: $\mathsf{max_{tmp}} \leftarrow$ tp.max
3: for $j = 0$ to tp.succs_size do
4:     $\mathsf{min_{tmp}} \leftarrow$ std::max($\mathsf{min_{tmp}}$, tp.succs[j].succ.min − tp.succs[j].max)
5:     $\mathsf{max_{tmp}} \leftarrow$ std::min($\mathsf{max_{tmp}}$, tp.succs[j].succ.max − tp.succs[j].min)
6: if tp.min! = $\mathsf{min_{tmp}}$ then
7:     ASSERT ( tp.min < $\mathsf{min_{tmp}}$ )
8:     tp.min $\leftarrow \mathsf{min_{tmp}}$
9:     vstate.aIncr(vstate.count) {atomically increment the state vector's counter}
10: if tp.max! = $\mathsf{max_{tmp}}$ then
11:     ASSERT ( tp.max > $\mathsf{max_{tmp}}$ )
12:     tp.max $\leftarrow \mathsf{max_{tmp}}$
13:     vstate.aIncr(vstate.count) {atomically increment the state vector's counter}

14: return !($\mathsf{min_{tmp}} > \mathsf{max_{tmp}}$)
```

We utilize the Alloy Analyzer to implement our semantic parallelization approach. Our method for semantic parallelization of the goal network is based on the observation that in a topology we can identify groups of time points that would allow the concurrent execution of the propagation passes. A possible criterion for identifying such groups would be to identify the time points in a topology that allow disjoin-access to the shared data. Given the method used to compute the time window $[\mathrm{TP}_{\mathrm{min}_i}, \mathrm{TP}_{\mathrm{max}_i}]$ for each $\mathrm{TP}_i \in \mathrm{S}_{\mathrm{tps}}$, we have observed that the functionally-independent time points are the time points that are equidistant (with respect to the longest path) from the root of the graph. Thus, in our methodology, we define a *Time Phase* TPH_i as the set of the time points ($\mathrm{S}_{\mathrm{TPH}_i}$) in a topology that are equidistant, with respect to the longest path, from the root of the graph. In such a way, by definition, the computations of $[\mathrm{TP}_{\mathrm{min}_a}, \mathrm{TP}_{\mathrm{max}_a}]$ and $[\mathrm{TP}_{\mathrm{min}_b}, \mathrm{TP}_{\mathrm{max}_b}]$ for every pair of $\{\mathrm{TP}_a, \mathrm{TP}_b\}$, such that $\mathrm{TP}_a \in \mathrm{S}_{\mathrm{TPH}_i}$ and $TP_b \in \mathrm{S}_{\mathrm{TPH}_i}$, are mutually independent and allow disjoin-access to the shared data. With the support of Alloy Analyzer we define and identify the time phases in a goal network graph (Algorithm 24 and Algorithm 25). Figure 20 provides an example of a goal network containing 15 time points and 6 time phases.

Algorithm 22 Definition of the notions of Temporal Constraint and Time Point

```
1: sig TC   {declaration of the Temporal Constraint signature}
2:   tc_pred: one TP,
3:   tc_succ: one TP
4: sig TP   {declaration of the Time Point signature}
5:   tp_preds: set TC,
6:   tp_succs: set TC
```

Algorithm 23 Main TCN invariants expressed in the Alloy Specification Language

```
1: all tc:TC | tc in tc.tc_pred.tp_succs
2: all tc:TC | tc in tc.tc_succ.tp_preds
3: all tc:TP | some tp.tp_preds ⇒ tp.tp_preds.tc_succ = tp
4: all tc:TP | some tp.tp_succs ⇒ tp.tp_succs.tc_pred = tp
5: no ∧(tc_pred.tp_preds) & iden   {check for cycles}
6: no ∧(tc_succ.tp_succs) & iden   {check for cycles}
```

Algorithm 24 Definition of the notions of Time Phase and Temporal Constraint Network (with time phases)

```
1: sig Tph   {declaration of the Time Phase signature}
2:   events: set TP,
3:   next: lone Tph,
4:   tcn: one TCN
5: sig TCN   {declaration of the TCN signature}
6:   epoch: TP,
7:   tps: set TP,
8:   tcs: set TC,
9:   init: one Tph
```

Having identified the time phases in our temporal constraint network specification in Alloy, the aim of the rest of our tool-chain is to *automatically* derive the C++ implementation of the parallel solution through a number of code transformation techniques. Following Rouquette's methodology [78] for model transformation

Algorithm 25 Main Time Phase invariants expressed in the Alloy Specification Language

```
1: all p:Tph
2:   p.events.tp_succs.tc_succ in p.^next.events
3:   p.events.tp_preds.tc_pred in p.^~next.events
4:   p in p.tcn.init.*next
5:   p.events in p.tcn.tps

6:   no p.events & p.^(next).events
```

through the application of the Object Constraint Language (OCL) and the Eclipse Modeling Framework (EMF), we are able to automatically derive an intermediary XML and XSD representations of the graph's topology and the TCN semantic notions, respectively. We apply an XML parser (XercesC) and a CodeSynthesis XSD transformation tool to deliver the C++ implementation of the goal network and our parallel propagation method.

To achieve higher safety and better performance, our parallel propagation scheme employs a number of recent multi-processor synchronization techniques. In our implementation we have encountered and addressed the following challenges:

(1) Achieving low-overhead parallelization. Our experiments indicated that the widespread pthreads are computationally expensive when applied to the parallel propagation algorithm. Given the frequent real-time changes in the graph topology, employing a thread per iteration for the computations of each time phase comes at a prohibitive cost. To avoid this problem, we have incorporated in our design the application of the Intel tasks from the Threading Building Blocks Library [25]. Our experiments indicate that the Intel tasks provide low-cost overhead when applied in the concurrent execution of the forward and backward passes of the propagation scheme.

(2) Allowing fast and safe access to the shared data. The parallel algorithm re-

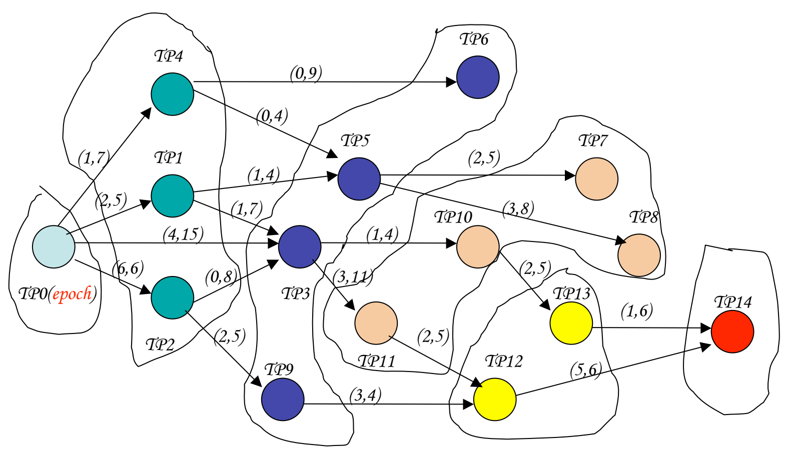


Fig. 20. A Parallel TCN Topology with 15 Time Points and 6 Time Phases

quires the safe and efficient concurrent synchronization of its shared data: the propagation queue and the vector containing control data (reflecting the updates during an iteration). By the definition of our algorithm, the propagation queue is synchronized by allowing only disjoint-access writes. While the access to the shared vector is less frequent, its concurrent synchronization is more challenging since we do not have a guarantee that the concurrent writes would be disjoint. The application of mutual exclusion locks is a possible but likely an ineffective solution due to the risks of deadlock, livelock, and priority inversion. We have employed the implementation of the lock-free vector described in Chapter III in order to meet our goals for thread-safe and effective nonblocking synchronization. The lock-free vector provides the functionality of the popular STL C++ vector as well as linearizable and safe operations with complexity of $O(1)$ and fast execution (outperforming the STL vector protected by a mutex by a factor of 10 or more).

A number of graph properties, in a particular TCN topology, have a significant impact on the application and performance of the parallel propagation scheme. We expect better performance (with respect to the sequential propagation scheme) when:

(1) The computational load per time point is high. This is the case of a real-world massive-scale goal network. For instance, instructing the Mars Science Laboratory to autonomously find its way in a Martian crater, probe the soil, capture images, and communicate to Mission Control will result in a goal network containing tens or hundreds of thousands of time points. In a small experimental graph topology with a low computational cost per time point (such as a few arithmetic operations), a single processor computation will perform best (when we take into account the parallelization overhead).

(2) Time phases with large number of time points: a topology implying a sequential ordering of the planned events will not benefit from a parallel propagation scheme. The parallel propagation algorithm is beneficial to goal networks representing a large number of highly interactive concurrent system processes.

D. Framework Application for Accelerated Testing

The presented design and implementation of our parallel propagation technique enable the incorporation of the optimized propagation approach described by Lou [77] in an experimental framework for accelerated testing currently still under development at NASA. Accelerated testing platforms suggest a paradigm shift in the certification process employed by NASA from system testing with the actual flight hardware and software to accelerated cost-effective certification using hardware simulators and distributed software implementations (Figure 21). Such frameworks aim faster-than-real-time testing and analysis of the complex software interactions in JPL's autonomous flight systems. A number of these platforms require automated refactoring of previously sequential code into modular parallel implementations. Preliminary results reported in academic work [20] as well as experience reports from a number of commercial tools (such as Simics by Virtutech and ADvantage BEACON by Applied Dynamics International) suggest the possible speedup of the flight system testing by a significant factor. We have followed Rouquette's methodology [78] that suggests the application of formal modeling and validation techniques that provide certification evidence for a number of functional dependencies in order to compensate for the added hazards in establishing the fidelity of the simulators. Due to the incomplete status of the accelerated testing framework as well as the lack of the actual flight hardware, it is difficult to measure a priori the effect of our parallel propagation scheme in achieving acceleration (with respect to the execution on the actual flight hardware) in the process of flight software testing. To gain insight of the possible performance gains and the algorithm's behavior we ran performance tests on a conventional Intel IA-32 SMP machine with two 2.0GHz processor cores with 1GB shared memory and 4 MB L2 shared cache running the MAC OS X operating system. In our performance analysis we have measured the execution time in seconds of two versions of our parallel propagation algorithm (one applying mutually exclusive locks and the other relying on nonblocking synchronization) and the original sequential scheme presented by Lou [77]. In the experiments (Figure 22), we have generated a number of TCN graph topologies (each consisting of 4 to 8 Time Phases), in a manner similar to the

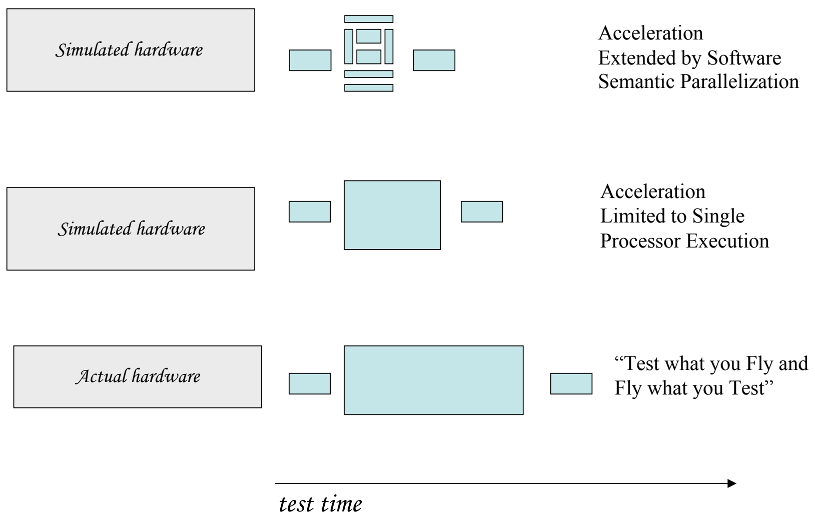

Simulated hardware
Acceleration Extended by Software Semantic Parallelization
Simulated hardware
Acceleration Limited to Single Processor Execution
Actual hardware
"Test what you Fly and Fly what you Test"
test time

Fig. 21. Testing Scenarios of Mission Software

pseudo-random graph generation methodology described in [81]. In the presented results on Figure 22 the $x - axis$ represents the average measured execution time (in seconds) of each propagation scheme and the $y - axis$ represents the number of time points in the exponentially increasing graph size (starting with a graph of 20000 TPs and reaching a TCN having 160000 TPs). In the experimental setup we observed that the parallel propagation algorithm offers effective execution and a considerable speedup in all scenarios on our dual-core platform. We measured performance acceleration reaching 28% in the case of the nonblocking implementation and 20% for our algorithm relying on mutually exclusive locks. Lock-free algorithms deliver significant speedup in applications utilizing shared data under high contention (Chapter III). In a scenario like our parallel TCN propagation scheme with medium or low contention

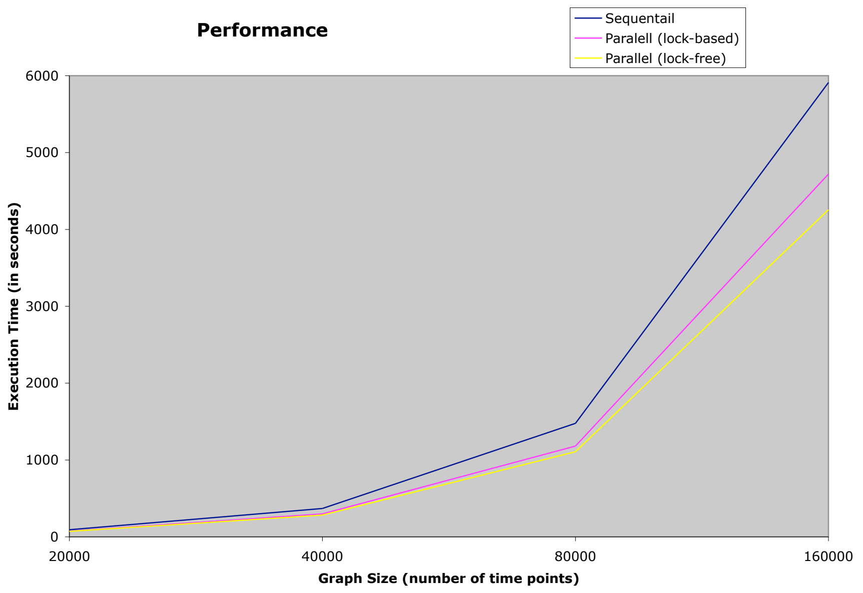


Fig. 22. TCN Constraint Propagation Performance Analysis

on the shared data, besides safety and prevention of priority inversion and deadlock, a lock-free implementation can guarantee better scalability. As our experimental results suggest, the gains from the lock-fee implementation gradually progress and we observe better scalability with respect to the blocking propagation scheme.

Table 14 provides a summary of the applied development tools that help us satisfy the seven critical certification variables in the process of TCN verification and parallelization. Each non-empty cell indicates the level of abstraction of the applied development tool. Empty cells are designated by the ∅ symbol. Below we briefly explain each entry in the table:

Table 14. Linking Certification Artifacts, Development Tools, and Levels of Abstraction

Cert. Artifact	Δ_{MC}	Δ_{PL}	Δ_{MT}	Δ_{AT}	Δ_{SA}
$S_{\eta_{safety}}$	$\emptyset$	Θ, Λ	$\emptyset$	$\emptyset$	$\emptyset$
ϵ_{exe}	$\emptyset$	Θ, Λ	$\emptyset$	$\emptyset$	$\emptyset$
η_{nb}	Φ, Θ	Θ	$\emptyset$	$\emptyset$	$\emptyset$
η_{lin}	Φ, Θ	Θ	$\emptyset$	$\emptyset$	$\emptyset$
η_{aba}	Φ, Θ	Θ	$\emptyset$	Λ	$\emptyset$
$S_{\eta_{tcn}}$	$\emptyset$	$\emptyset$	Λ	Λ	Ξ
σ_{ptcn}	$\emptyset$	Λ	Ψ	Θ, Λ	$\emptyset$

(1) $S_{\eta_{safety}}$: to eliminate the dangers of deadlock (η_{delock}), livelock (η_{lvlock}), and priority inversion (η_{pinv}) we have relied on the use of a library of nonblocking algorithms that allow the fast and safe implementation of shared data access of the C++ STL vector. Thus our approach to deliver safe concurrent interactions is based on the application of advanced algorithms (Θ) and language library extensions (Λ).

(2) ϵ_{exe}: as described in detail in Chapter III, when used under contention a nonblocking shared vector can deliver a significant performance boost (by a factor of 10 or more) when compared with the application of the most recent and optimized mutual exclusion schemes. In the scenarios when the shared data structure access patterns show less contention, the nonblocking techniques provide a scalable and efficient solution with performance better or equal to the most optimal mutual exclusive schemes [61]. Achieving better performance and scalability of our parallel goal network is also based on the application of programming techniques at

the algorithms/library level.

(3) η_{nb}: the careful application of CAS-based speculation on single-word memory locations allows us to guarantee that among a set of contending processes trying to manipulate the shared vector, there is at least one that is guaranteed to progress. To construct our library of nonblocking algorithms we have relied on the atomic primitives provided by the hardware architecture (Φ) and a set of practical lock-free programming techniques (Θ).

(4) η_{lin}: some operations in a shared vector require the update (in a linearizable fashion) of two or more memory locations. Such operations are push_back (need to update the tail and the size of the vector) and resize (need to update the size and copy all elements). Implementing such operations in a linearizable fashion with the support of only single-word atomic primitives is notoriously difficult. We have employed a set of practical lock-free programming techniques to guarantee that the vector's operations are linearizable (such a technique is the use of a Descriptor Object, Chapter VII).

(5) η_{aba}: the ABA problem is fundamental to all CAS-based systems and can affect the semantics of the nonblocking algorithms. In systems allowing complex atomic primitives such as CAS2 or DCAS, ABA can be easily avoided by attaching a version counter to each value. In such a case we would have had an algorithmic solution with a strong support from the hardware architecture. We cannot assume the availability of such complex atomic primitives in the hardware architecture of the flight-qualified embedded hardware. One possible solution we suggest to the ABA problem (Chapter VI) is the application of a library for program analysis [6] that can help us eliminate the hazardous interleaving of concurrent operations.

(6) $S_{\eta_{tcn}}$: to guarantee the correct operation of our autonomous goal-driven application, we have build a framework (Figure 19) that relies on modeling, program analysis, and program transformation programming techniques.

(7) σ_{ptcn}: we have increased the comprehensibility of our parallel goal network implementation by: a. shifting the complexity of allowing safe and efficient concurrent operations into a library of nonblocking containers (Λ), b. used the Alloy modeling notation to express the software architectural and design notions (Ψ), and c. applied program analysis and transformation techniques to automatically derive the implementation source. Any further evolution of the system would rely on high-level models expressed in simpler design-level domain-specific terms.

In Chapter V we introduced a framework for model-based product-oriented certification founded on the concept of source code enhancement and analysis. We offered a classification of the certification artifact types, the development and validation tools and techniques, the application's domain-specific factors, and the levels of abstraction used in our certification platform. In this chapter, we used our certification platform to analyze the model-driven development of a parallel propagation scheme of the MDS temporal constraint network module. In our analysis we identified seven critical certification artifact:

1. providing the safety of the concurrent interactions (by eliminating the hazards of deadlock, livelock, and priority inversion),

2. achieving better scalability and overall system performance,

3. allowing nonblocking synchronization,

4. having linearizable operations on the shared data,

5. eliminating the possibility of ABA corrupting the concurrent operations' semantics,
6. establishing the correctness of the core TCN graph invariants,
7. having simpler to analyze and maintain parallel processes.

In our discussion we explained the relationships among these seven certification artifacts and the underlying hardware architecture, the applied programming techniques, and program analysis, modeling, and transformation techniques. Our certification framework helped us formulate, express, and analyze the process of product-oriented certification for a complex computer-based system, such as the model-driven development tool-chain of parallel autonomous goal networks.

We presented in this chapter a first time and concurrency centered framework for validation and semantic parallelization of real-time C++ within JPL's MDS Framework. We demonstrated the application of our framework in the validation of the semantic invariants of the Temporal Constraint Network Library. Temporal constraint networks are at the core of the mission planning and control architecture of the Mission Data System framework. In addition, we presented an approach for automatic semantic parallelization of the propagation scheme establishing the consistency of the temporal constraints in a goal network. Our parallel propagation scheme is based on the identification of time phases within a goal network and is implemented through the application of model transformation and formal analysis techniques to the model specifications of the TCN semantics. We have relied on lock-free synchronization techniques to achieve better performance and higher safety of our parallel implementation.

CHAPTER IX

C++ DYNAMIC CAST IN AUTONOMOUS SPACE SYSTEMS

The dynamic cast operation allows flexibility in the design and use of data management facilities in object-oriented programs. Shared data in a real-time cyber-physical system can often be polymorphic (as is the case with a number of components part of the MDS Data Management Services (DMS) [5]). The use of dynamic cast is important in the design of autonomous real-time systems since the operation allows for a direct representation of the management and behavior of polymorphic data. Workaround techniques often lead to restricted error-prone solutions with poor maintainability and high complexity. To allow for the application of dynamic cast in mission critical code, we analyze and validate a methodology for constant-time dynamic cast that transfers the complexity of the operation to the compiler's static checker. Dynamic cast has an important role in the implementation of the Data Management Services (DMS) of the Mission Data System Project (MDS). DMS is responsible for the storage and transport of control and scientific data in a remote autonomous spacecraft. Like similar operators in other languages, the C++ dynamic cast operator does not provide the timing guarantees needed for hard real-time embedded systems. In a recent study, Gibbs and Stroustrup (G&S) devised a dynamic cast implementation strategy that guarantees fast constant-time performance [29]. This chapter presents the definition and application of a co-simulation framework to formally verify and evaluate the G&S fast dynamic casting scheme and its applicability in the Mission Data System DMS application. We describe the systematic process of model-based simulation and analysis that has led to performance improvement of the G&S algorithm's heuristics by about a factor of two.

ISO Standard C++ [23] has become a common choice for hard real-time embedded systems such as the Jet Propulsion Laboratory's Mission Data System [82]. This is so because ISO C++ offers efficient abstraction model, good hardware use, and predictability. C++'s model of computation has helped engineers deliver more correct, maintainable, and comprehensible software compared to code relying on lower-level programming concepts [83]. However, several C++ features are usually considered unsuitable for programming real-time systems because they do not guarantee predicable constant-time performance [4]. ISO C++ does not provide the necessary timing guarantees for free store (heap) allocation, exception handling, and dynamic casting. In particular, the most common compiler implementations of the dynamic cast operator traverse the representation of the inheritance tree (at run time) searching for a match. Such implementations of dynamic cast are not predictable and are unsuitable for real-time programming. Gibbs and Stroustrup (G&S) [29] describe a technique for implementing dynamic cast that delivers significantly improved and constant-time performance. The key idea is to replace the runtime search through the class hierarchy with a simple (constant-time) calculation, much as the common implementations of the C++ virtual function calls search the class hierarchy at compile time to reduce the runtime action to a simple array subscripting operation. In the G&S scheme, a heuristic algorithm assigns an integer type ID at link time to each class. The type ID assignment rules guarantee that at run time a simple modulo operation can reveal the type information and check the validity of the cast. The requirements for the heuristics assigning the type IDs are that:

(1) They must keep the size of the type ID to a small number of bits. A 64-bit type ID should be sufficient for very large class hierarchies.

(2) Avoid conflicts and type ID assignments that create ambiguous or erroneous type resolution at run time.

(3) Handle virtual inheritance.

There are four heuristic schemes and a few possible optimizations suggested in [29]. However, none of those heuristics guarantee the best solution for every possible class hierarchy. The quality of the type ID assignment heuristics has a critical importance for the performance of the G&S scheme. With better heuristics, a smaller type ID size would be sufficient to facilitate complex and large class hierarchies that would need a significantly bigger type ID size with a poor assignment scheme. The main contribution of this work is to present how the algorithm optimization problem encountered has been successfully automated and moreover that its automation has led us to quick but significant improvements of the initial scheme.

In this chapter we present a co-simulation framework based on the SPIN model checker [62] to simulate, evaluate, and formally verify the G&S fast dynamic casting algorithm and its application in mission critical code such as the Data Management Services [5] of the Mission Data System. We use the feedback from the model checker to perform systematic analysis of the G&S scheme and look for improvements to the heuristics for type ID assignment. SPIN is an on-the-fly, linear-time logic model-checking tool that is designed for the formal verification of dynamic systems with asynchronously executed processes. The most recent advances in the state space reduction techniques have made it possible to validate large software applications. Model-checking tools have been widely applied for the verification of a large variety of systems, including flight software [76], network protocols [84], and scheduling algorithms [85]. We are unaware of work suggesting its use for the analysis and optimization of compiler heuristics. Compiler verification usually focuses on seeking a proof on the preservation of the program semantics during the various compiler passes [86]. Our work presents the application of a model-checking tool for the analysis and refinement of the combinatorial optimization problem posed by the G&S type ID assignment scheme. Our co-simulation framework consists of the following components:

(1) An abstract model of the G&S type ID assignment heuristics.

(2) A procedure for exhaustive search of the state space discovering the best type ID assignment.

The analysis of the heuristics simulation performed in SPIN provides us with ideas of possible improvements to the G&S type ID assignment. We include and evaluate the proposed improvements in the abstract model in order to seek refinement of the G&S type ID assignment scheme. The experiments we have executed show that the G&S priority assignment is not optimal with respect to the best possible type ID assignment where non-virtual multiple inheritance is used. While potentially dangerous if not constructed carefully, such hierarchies happen to be of significant practical importance [4]. Based on our experiments, we suggest optimizations that lead to significant improvement of the G&S heuristics performance. This chapter makes the following contributions:

(1) Introduces the use of a co-simulation framework based on model-checking for the analysis and improvement of a compiler-heuristics optimization problem.

(2) Verifies and analyzes the G&S C++ fast dynamic casting scheme and its application in mission critical code such as the MDS Data Management Services.

(3) Implements optimizations to the G&S heuristics leading to the discovery of optimal type ID assignment in 85% of the class hierarchies, in contrast to 48% for the original G&S algorithm.

A. Fast Dynamic Casting Algorithm

The G&S fast constant-time implementation of the dynamic cast operator works as follows: at link time, a static integer type ID number, preferably 32 or 64-bit long, is assigned to each class. The ID numbers are selected so that the operation id_a *modulus* id_b yields zero if and only if the class with id_a is derived from the class with id_b. This is done by exploiting the uniqueness of factorization of integers into prime factors. Each class is assigned a *key* prime number. The *type ID* of a class is calculated by multiplying its *key number* with the key numbers of each of its base classes. In the cases where a class contains more than a single copy of a base class, the type ID is computed by taking the square of the corresponding base class ID. The only constraint of the approach is the requirement to limit the ID size to fit the machine's built-in integer types. The key primes are not required to be unique and the same prime key can be used for classes that belong to different groups (i.e. do not share common descendants). Gibbs and Stroustrup suggest four approaches for assigning the type IDs in a space-efficient manner. Each method is based on a preliminary computation of the priority factor of each class. The priority reflects the class impact on the growth of the type ID numbers in the hierarchy. Thus, classes with greater number of descendants should receive higher priority and smaller key prime number values respectively. The four possible schemes suggest that:

1 The priority of a class is the *maximum* number of ancestors that any of its descendants has. This scheme was chosen for the initial implementation and testing of the G&S algorithm and also closely followed in the implementation of the abstract model used for our simulation.

2 ,3, 4. If a range of primes is assigned to every level with wider levels receiving larger initial values, then each node could be assigned an additional value that is

proportional to the logarithm of the (*2. minimum, 3. mean, 4. maximum*) prime in its level. Priorities of hierarchy leaves are computed by taking the sum of these additional values for the leaf itself and all of its ancestor classes.

After the priority of each class has been computed, the classes with the highest priority get the smallest prime numbers. According to this scheme, prime numbers can be reused only if there are two classes on the same level of the class hierarchy and only if they do not share common descendants, they are not siblings, and also that none of their parents share a common descendant. Given the class hierarchy on Figure 23,

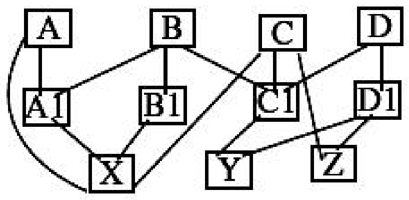


Fig. 23. Fast Dynamic Cast, a Class Hierarchy with 11 Classes

we follow the ID assignment rules and establish that:

(1) $id_x = k_x \times (k_a)^2 \times k_{a_1} \times (k_b)^2 \times k_{b_1} \times k_c$,

(2) $id_y = k_y \times k_c \times k_{c_1} \times (k_d)^2 \times k_{d_1} \times k_b$,

(3) $id_z = k_z \times k_d \times k_{d_1} \times k_c$.

Given a set C with 11 classes in the hierarchy and the set of the first 11 prime numbers $P = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31\}$, we must assign each class V a key $k_v \in P$ such that, the maximum of the set $id_{leaf} = \{id_x, id_y, id_z\}$, the set consisting of the ID numbers of all leaf nodes in C, is minimal. As we already know, prime numbers need not be unique for each class and can be reused in same circumstances.

B. A Co-simulation Framework

The goals of the co-simulation framework are to validate the main invariants of the G&S heuristics, improve its performance, and establish its applicability in mission critical systems. The co-simulation process in the framework (Figure 24) consists of three consecutive stages: verification, evaluation, and analysis. The verification phase is a straightforward application of model checking where an abstract description of the system's behavior is checked against a set of invariants. In the evaluation stage the simulation results from the probabilistic approach are contrasted to the outcome of the deterministic approach. The aim of the analysis stage is to closely examine the instances where the solutions yielded by the two implementations differ. We identify patterns among the inconsistent results that reveal the weaknesses of the probabilistic solution. The framework works by executing two independent models,

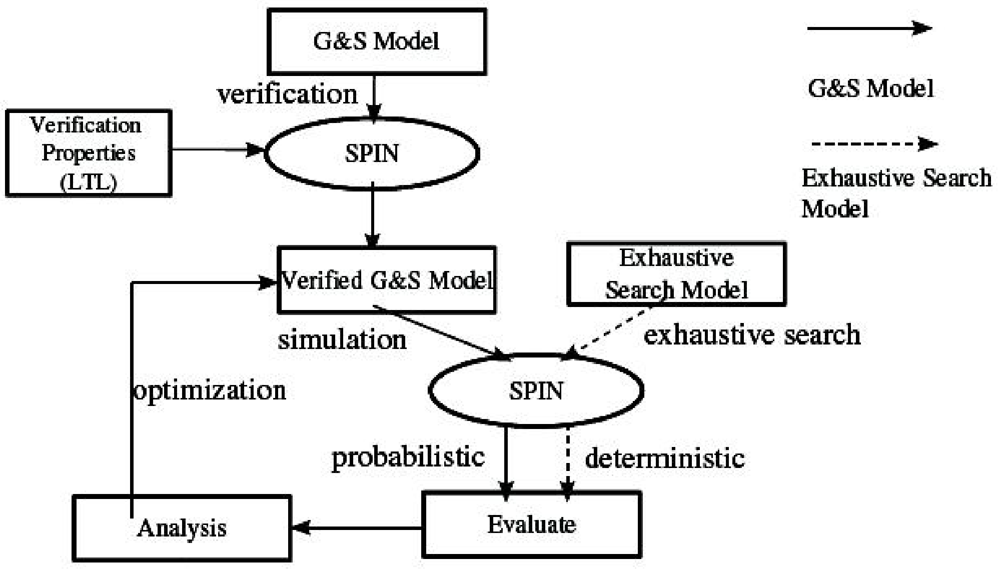


Fig. 24. A Co-Simulation Framework for G&S Improvement and Verification

the G&S model and the exhaustive search model. The first input component to the co-simulation framework (Figure 24) is an abstract model of the G&S fast dynamic casting heuristics, implemented in Promela (SPIN's input language) and the embedded C primitives it allows. The G&S abstract model is subsequently used to verify the main invariants of the G&S heuristics and at the same time provide us with a simulation testbed to examine the heuristics performance. The second component of the framework is the exhaustive search model that simply looks into all possible type ID assignments to discover the optimal solution for a given class hierarchy. We employ SPIN's search engine to perform the exhaustive search. In Algorithm 26 we present the pseudocode of our co-simulation approach. The following sections elaborate in more details on each of the stages of the framework.

1. Formal Verification

Every G&S implementation operates under the assumption that when a prime number is reused, it is assigned to non-conflicting classes. In addition, the maximum type ID must fit within the boundaries of a memory word. We check these invariants during the program verification phase. Establishing the validity of the G&S invariants is done by straightforward application of model-checking with SPIN. In SPIN the critical system properties are expressed in the syntax of linear time logic. Based on the G&S abstract specification, the model-checker performs a systematic exploration of all possible states. In case of failure, SPIN provides a counterexample that demonstrates a behavior that has led to an illegal state. In our model, the invariants are expressed as a *never claim* [62], and are checked just before and after the execution of every statement.

2. Evaluation

SPIN has been previously employed to implement solutions of scheduling [87] and discrete optimization [85] problems. The problem we face in the G&S heuristics is a combinatorial optimization problem [88]. Given a finite set I, a collection F of subsets of I, and a real-valued function w defined on I, a discrete optimization problem could be defined as the task of finding a member S of F, such that: $\sum_{e \in S} w(e)$ is as small (or as large) as possible.

Except for the simplest cases, a discrete optimization problem is difficult because its design space is typically disjoint and nonconvex. Therefore, the optimization methods applied to continuous optimization problems cannot be utilized in this case. In a small discrete problem, it would be possible to exhaustively list all possible combinations. As the number of parameters increase, the state explosion makes optimizations difficult. The two general strategies for approaching a discrete optimization problem can be classified as *deterministic* and *probabilistic*. What we do for the G&S exploration in SPIN could be described as applying a deterministic approach for the evaluation of a set of proposed probabilistic methods. The Branch and Bound method [88] guarantees the discovery of the global optimum in the cases when the problem is linear or convex and is the most frequently used discrete optimization method. It is based on the sequential analysis of the discrete tree of each parameter. The branches that can be estimated to reach invalid or unfeasible solutions are consequently eliminated. This simple optimization could also be applied in some limited cases in the SPIN's Fast Dynamic Casting exhaustive search. Let us explore a class hierarchy with three classes A, B, and C, where B is derived from A, and C is derived from both A and B. In this case, we have $C = \{A, B, C\}$, $P = \{2, 3, 5\}$, and $id_{leaf} = \{id_c\}$. The enumeration is given in Table 15. We assume that the computation starts at a state

Table 15. Fast Dynamic Cast, Enumeration of All Solutions

$id_c = k_c \times k_b \times (k_a)^2$	k_a	k_b	k_c
60	2	3	5
60	2	5	3
90	3	2	5
90	3	5	2
150	5	2	3
150	5	3	2

S_0 where all three keys k_a, k_b, and k_c are uninitialized. Then we assign possible values from the set P to the key variables of the classes A, B, and C. The enumeration shown above can be expressed as the computation shown on Figure 25. The graph

$$\begin{tikzcd}
 & S_1 \arrow[r, "K_B = 3"] \arrow[dr, "K_B = 5"] & S_4 \\
 & & S_5 \\
S_0 \arrow[uur, "K_A = 2"] \arrow[r, "K_A = 3"] \arrow[ddr, "K_A = 5"'] & S_2 \arrow[r, "K_B = 2"] \arrow[dr, "K_B = 5"] & S_6 \\
 & & S_7 \\
 & S_3 \arrow[r, "K_B = 2"] \arrow[dr, "K_B = 3"'] & S_8 \\
 & & S_9
\end{tikzcd}$$

Fig. 25. Fast Dynamic Cast, Exhaustive Search Computation

shows only the valid states of the computation. There are a number of invalid states that are not shown on the graph. For example, according to the rules defined in G&S,

it is possible to reuse some of the prime numbers in P. Thus, we can try and add an edge $k_b = 2$ in state S_1, however the reuse of 2 in this case is invalid since A and B are conflicting classes.

The illustrated automation in Figure 25 provides a foundation for the construction of a Promela model for the deterministic solution. Each possible prime number assignment to a given class key is represented by a separate state transition in the exhaustive search model. SPIN initiates the optimum search at state S_0 and visits all possible states. At each end state the value of the minimum of the set of leaves, in this case represented only by id_c, is computed and compared to the current minimum. This approach is similar to the algorithm described by Ruys [85] and shown in Algorithm 27. For such an application, we use the model checker in a somewhat unusual fashion. In this scenario, the validation property checks whether the value of id_c is greater than the current minimum. Each time this condition is violated, the current minimum is updated and the process is automatically repeated until SPIN confirms that there are no routes violating the specification. Since the solution is deterministic, it is guaranteed to discover the global optimum for type ID assignment. The performance of the G&S heuristics is measured by running a simulation of the G&S model that has been used earlier for verification. Now we are left with only one important task (not automated at this stage), the comparison of the results from the probabilistic and deterministic solutions. Once we identify a set of inconsistent results, we try to find a pattern and refine the G&S heuristics. Then the refined scheme is implemented in the probabilistic model and the evaluation process is reiterated.

3. Analysis

The simulation and enumeration models are continuously executed until, if possible, a set of instances with inconsistent solutions can be identified. Thus, each instance in the *Set of Inconsistent Solutions* (SIS), represents a given class hierarchy for which the deterministic and probabilistic approach have discovered different solutions. The class hierarchies for each test could be guided or created in a random fashion. For the generation of the test data in our experiments we implemented a pseudo random class hierarchy generation algorithm, in a manner similar to the TGFF (Task-Graphs-For-Free) method as described in [81]. We look for patterns among the collected hierarchies in SIS and seek clues that can lead us to improvements of the G&S scheme. Potential improvements are tested by adding them to the G&S model and evaluating their effect with our co-simulation approach. Such scheme modifications are carefully selected since it is possible that they might enhance a given G&S feature and at the same time weaken another. Ideally, the improvements lead to a heuristic scheme that provides the best solutions for a larger number of the test hierarchies and at the same time has a time complexity equal to or less than the earlier heuristic scheme.

Despite the numerous advanced state space reduction techniques utilized by the SPIN model checker, little can be done to further optimize the exhaustive search. The main goal of our experiments is to reach quick and effective optimization of the G&S scheme, thus the class hierarchies considered were not the largest and most complex that our models can handle. The models developed for our experiments are capable of handling class hierarchies of double or triple the size of the ones presented in the paper, and even larger number of classes can be facilitated with increased computational power. In the framework, the exhaustive search is used to identify flaws in the G&S type ID assignment scheme, thus, there is no need to create and simulate much larger hierarchies. In this work our goal is to demonstrate that the current size of the class hierarchies is sufficient to discover significant flaws of the original heuristics.

C. Application in Mission-Critical Software

Modern space mission systems have evolved from simple embedded devices into complex computing platforms with high autonomy and an exceptionally large demand for human-computer interaction. Consequently, such systems require reliable and flexible data systems managing the collection, storage, and transportation of data. MDS provides the building blocks for the implementation of embedded platforms based on the concepts of state estimation and control. The Data Management Services (DMS) is the MDS component responsible for the production, storage, processing, and transfer of control and scientific data. Wagner [5] defines the challenges of data management in MDS as the problems of producing and storing data and converting the data to various formats as needed by its consumers. In addition, DMS needs to ensure the secure and lossless transport of the data with limited resources and through unreliable physical medium. To design and relate the data system entities, DMS employs concepts from high-level ISO C++ including templates, object-oriented class encapsulation, and dynamic casting necessary for the conversion of the data formats.

The actual telemetry data objects in MDS communicate with each other via byte streams produced by the transport protocol (e.g. spacecraft to ground communication). The receiver of the telemetry data needs to recreate the data object from the byte stream and thus invoke type casting in numerous occasions. Constant-time dynamic cast is also needed by the MDS Goal Network in the case when a controller or estimator [5] passes a goal via the Coordinating Goal Network (CGN), typically a large dynamic data structure. In CGN the goal is propagated using only its abstract attributes (start and end time, and the associated state variable). The achiever object who eventually picks up the goal needs to reconstruct the data object via dynamic downcasting to the specific type that conveys the state-specific achievement criteria.

The application of the common compiler implementation of dynamic cast has proved to be unacceptable due to poor performance and the lack of the timing guarantees.

The G&S scheme was devised as a solution to a real industrial problem related to C++ use for hard real-time code. Inquiries in the C++ community revealed that the problem was fundamental and common, rather than isolated: developers simulate dynamic casting with other language features, leading to type-unsafe special-purpose code or the avoidance of best object-oriented practices. Naturally, such workaround code slows down development, complicates maintenance, and increases the need for testing.

D. Results

We applied the co-simulation process described in the previous section to a large number of class hierarchies. The tested hierarchies are not built into our models. Instead, we have followed a pseudo random generation methodology similar to TGFF task graph generation as described in [81] to automatically generate hundreds of possible test cases. For illustration, we show the results from a set of seven pseudo random class hierarchies. The results of the G&S heuristics model and the exhaustive search are shown in Table 16. A brief comparison of the results indicates that the G&S heuristics do not give optimal performance for class hierarchies with non-virtual multiple inheritance. A closer look at the algorithm reveals that the priority calculation routine takes into account only the number of descendants that each class has. Let us consider the class hierarchy from test case 7. We notice that according to the current scheme, the base classes 0, 1, and 2 all get the same priority rank since they all share the descendant 6. Class 6 is at the lowest level of the hierarchy and has the largest number of ancestors. If we would like to optimize the heuristics, we must find a way to increase the priority of base class 2. Our reasoning is derived from the fact that Class 2 is ambiguous and the leaf Class 6 contains two copies of it. Similarly, let us have a closer look at test case 1. In the optimal solution, Class 5 takes the lower prime number (11) compared to Class 4, despite the fact that its only descendant has less ancestors compared to Class 4. The reason for this result is the fact that the derived Class 3 contains two ambiguous bases while Class 4 contains only one ambiguous base. As a result of our analysis we conclude that higher priority should be given to derived classes and their ancestors who contain more ambiguous base classes. To fix these weaknesses, we extend the G&S heuristics by adding two simple rules:

(1) We count every ambiguous ancestor twice when we determine the number of ancestors to each class.

(2) For each base class, we count the number of derived classes that include more than one copy of it, and add that number directly to its priority.

We call this enhanced G&S heuristics Fast Dynamic Casting Plus (FDC+). As Table 16 shows, for the initial set of test cases, FDC+ performance is 100% equivalent to the performance of the deterministic approach. In the performed tests, we have generated 127 pseudo random class hierarchies and applied G&S, FDC+, and the exhaustive search to each one of them. Figures 27, 28, 29 and 30 show seven examples of our test scenarios. The experimental results showed that FDC+ was able to yield the best type ID assignment in 85% of the class hierarchies compared to 48% for the G&S heuristics. The time performance of the three schemes is shown in Figure 26. While the time performances of the G&S and FDC+ algorithms are equal and both run in a very low constant-time (the function at 00:01 min on Figure 26), logically the time performance of the exhaustive search increases exponentially with the increase

Table 16. Fast Dynamic Cast, Co-simulation of the Seven Cases

Case No	G&S	Exhaustive search	FDC+
Case 1 (keys)	(2, 3, 5, 7, 11, 13, 17)	(3, 2, 5, 7, 13, 11, 17)	(3, 2, 5, 7, 13, 11, 17)
Case 1(*ids* of all leaves)	(16380, 16830)	(13860, 13260)	(13860, 13260)
Case 2 (keys)	(2, 13, 3, 5, 17, 7, 11)	(2, 13, 3, 5, 17, 7, 11)	(2, 13, 3, 5, 17, 7, 11)
Case 2 (*ids* of all leaves)	(1326, 2310)	(1326, 2310)	(1326, 2310)
Case 3 (keys)	(2, 3, 13, 5, 7, 17, 11)	(2, 3, 13, 5, 7, 17, 11)	(2, 3, 13, 5, 7, 17, 11)
Case 3 (*ids* of all leaves)	(26, 51, 2310)	(26, 51, 2310)	(26, 51, 2310)
Case 4 (keys)	(2, 3, 5, 7, 11, 13, 17)	(2, 3, 5, 7, 11, 13, 17)	(2, 3, 5, 7, 11, 13, 17)
Case 4 (*ids* of all leaves)	(2310, 1547)	(2310, 1547)	(2310, 1547)
Case 5 (keys)	(2, 3, 5, 7, 11, 7, 11)	(2, 3, 5, 7, 11, 7, 11)	(2, 3, 5, 7, 11, 7, 11)
Case 5 (*ids* of all leaves)	(42, 66, 70, 110)	(42, 66, 70, 110)	(42, 66, 70, 110)
Case 6 (keys)	(2, 3, 5, 11, 13, 7, 17)	(2, 3, 5, 11, 13, 7, 17)	(2, 3, 5, 11, 13, 7, 17)
Case 6 (*ids* of all leaves)	(66, 78, 420, 170)	(66, 78, 420, 170)	(66, 78, 420, 170)
Case 7 (keys)	(2, 3, 5, 7, 11, 13, 17)	(3, 5, 2, 7, 11, 13, 17)	(3, 5, 2, 7, 11, 13, 17)
Case 7 (*ids* of all leaves)	(2552550)	(1021020)	(1021020)

of the number of classes nodes in a given class hierarchy. The analysis of the test results indicates that FDC+ finds a better type ID compared to the G&S approach in 39% of the test scenarios. For the greater part of the test cases, FDC+ matched the optimal type ID assignment computed by the exhaustive search. This efficiency boost is due to the optimized performance of FDC+ in the cases where multiple non-virtual inheritance is present in the class hierarchy. We have also observed that the implementation of these optimizations does not lead to efficiency loss in other scenarios and the performance of FDC+ is always at least as good as the performance of G&S. Our experimental results have indicated that the introduced optimizations in FDC+ have fixed a weakness of the original G&S approach and have improved the success rate in finding the best type ID assignment. The G&S scheme requires a key of a memory size that is a function of the size and shape of a class hierarchy. Thus, the improved heuristics almost double the size of class hierarchies that can be handled by a given key size. Since the scheme gets significantly slower when a key gets too large for a machine word, the improvements to the heuristics address the main limitation of the G&S scheme.

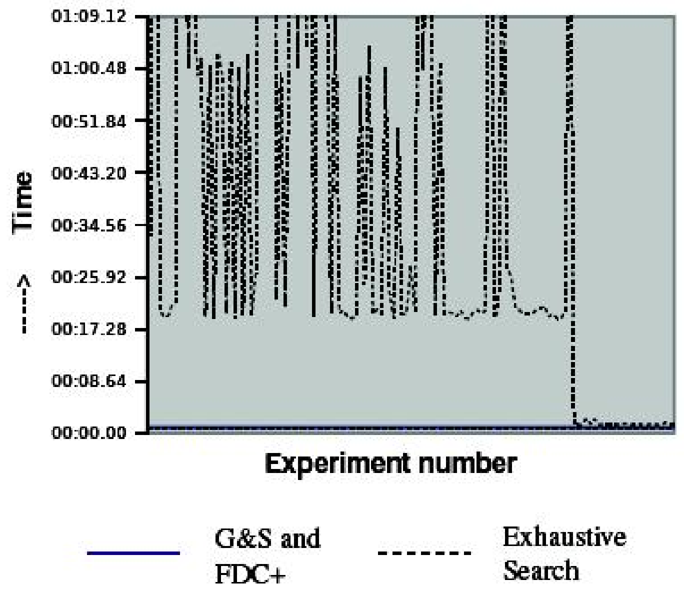


Fig. 26. Fast Dyanmic Cast, Search Time for Type ID Assignment

In this chapter we applied co-simulation of the deterministic and probabilistic solutions to the combinatorial optimization problem posed by the G&S type ID assignment scheme. Our framework proved successful in verifying and refining the existing G&S heuristics. We demonstrated how we use the simulation results to devise improvements to the G&S algorithm and evaluate them. The results from our experiments indicate that the improved G&S heuristics (FDC+) provide the optimal type ID assignment in 85% of the class hierarchies, compared to 48% for the regular G&S algorithm. The efficiency of the type ID assignment scheme has significant importance for the performance of the fast dynamic casting by Gibbs and Stroustrup [29]. This paper presented a practical approach of how to discover improvements to the type ID assignment scheme in a simple and effective manner. The main advantage of the presented approach is the ease and simplicity of the discovery and testing for potential improvements. The improved heuristics that we have described in this work almost doubles the size of class hierarchies that can be handled by a given key size. A more extensive simulation might suggest further improvements to the type ID assignment scheme. Our main goal in this work has been to demonstrate how an algorithm optimization problem has been successfully automated and moreover that its automation has led us to quick but significant improvements of the initial scheme. In the future we intend to utilize a static analysis tool for automatic class hierarchy analysis and extraction.

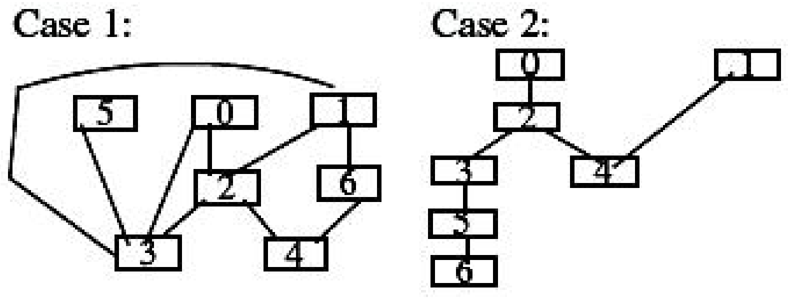


Fig. 27. Fast Dynamic Cast, Test Cases 1 and 2

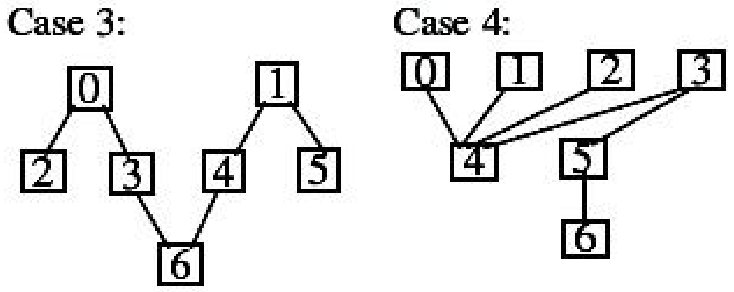


Fig. 28. Fast Dynamic Cast, Test Cases 3 and 4

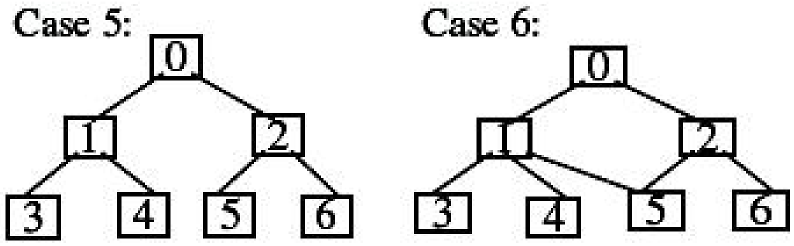


Fig. 29. Fast Dynamic Cast, Test Cases 5 and 6

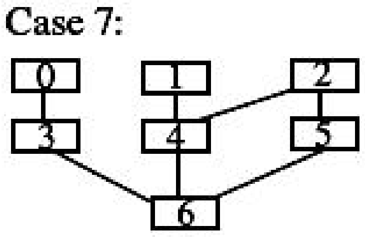


Fig. 30. Fast Dynamic Cast, Test Case 7

Algorithm 26 Pseudocode of the co-simulation approach.

```
1: const int MAX_NUMBER_TESTS
2: VERIFY:
3: repeat
4:   Formal Verification (G&S Model) $\rightarrow$ error report
5:   if (no errors) then
6:     goto EVALUATE
7:   else
8:     study counter example
9:     correct G&S
10: until TRUE
11: EVALUATE:
12: count=0
13: for (count < MAX_NUMBER_TESTS) do
14:   Simulation(G&S Model) $\rightarrow$ G&S solution
15:   Enumeration(Exhaustive Search Model) $\rightarrow$ best solution
16:   if (G&S solution $\neq$ best solution) then
17:     add instance to SIS
18:   count++
19: ANALYZE:
20: for all $i \in$ SIS do
21:   look for a pattern
22:   modify G&S
23:   goto EVALUATE
```

Algorithm 27 Finding the global minimum in the state space

```
1: intput: Promela model M
2: output: the optimal minimum for the problem M
3: min=(worst case) maximum value for id
4: repeat
5:    use SPIN to check M with condition ($id_c$ >min)
6:    if (error found) then
7:       min = $id_c$
8: until (error found)
```

CHAPTER X

CONCLUSION

In this dissertation we explored the problem of how to design, implement, and validate safe and efficient software for the highly complex, embedded, and autonomous computing platforms of the future robotic spacecraft. Our approach is centered on the idea of semantic enhancement of the highly efficient C++ computing model. The notions of time and concurrency are critical to the design of autonomous space software. We have reached our goals of delivering *reliable* and *efficient* concurrent synchronization by introducing and implementing the concept of Semantically Enhanced Containers. We have presented and utilized innovative nonblocking programming techniques to avoid the hazards of deadlock, livelock, convoying, and priority inversion. To eliminate the fundamental ABA problem, we defined and applied a generic condition for ABA-free nonblocking designs, called the $\lambda\delta$ approach. The experimental data from a number of tests demonstrate that our SEC approach is fast, scalable, and efficient. Our lock-free design outperforms under contention the alternative lock-based approaches by a factor of 10 or more. In a scenario with less contention and a lack of shared cache memory, our lock-free SEC containers offer performance comparable to that of the most optimal lock-based techniques. Compared to the application of the popular Software Transactional Memory approach, our SEC design showed high scalability and superior performance, outperforming the STM-based container by a significant factor. Our generic strategy for ABA avoidance, the $\lambda\delta$ approach, offers a solution to the ABA problem without the need to rely on the application of a complex atomic primitive, and offers performance comparable to the use of the architecture-specific CAS2 instruction. To enable the application of dynamic cast in the MDS object-oriented designs, we applied our model-based semantic enhancement framework in order to validate and improve a methodology for implementing a constant-time dynamic cast operation that is safe for hard real-time software. Our SEC approach allowed us to design a framework for validation and automatic semantic parallelization of the MDS Goal Networks, a critical component of the MDS goal and state based architecture.

At the present moment of time the notions of time and concurrency lack first class representation in the popular general purpose programming languages. Our SEC approach offers an alternative that allows the use of the advanced C++ tool-chain while at the same time providing explicit support for safe concurrent interactions. The modern hardware architectures and programming languages' memory models do not explicitly support the concept of nonblocking synchronization. As a result, nonblocking algorithms are notoriously difficult to design and the software designers need to exploit a set of low level programming "hacks" for their effective implementation. In the future, we are interested to explore the problem of offering hardware support for lock-free synchronization. In addition, we intend to look into the possibilities of extending the ISO C++ memory model for the explicit support of effective and safe concurrent multi-core programming techniques that are also suitable for real-time embedded applications.

REFERENCES

[1] E. Denney and B. Fischer, “Software certification and software certificate management systems,” in *Proceedings of the 2005 ASE Workshop on Software Certificate Management (SoftCement 2005)*, 2005, pp. 17–28.

[2] M. R. Lowry, “Software construction and analysis tools for future space missions,” in *TACAS*, ser. *Lecture Notes in Computer Science*, J.-P. Katoen and P. Stevens, Eds., vol. 2280. Grenoble, France: Springer, 2002, pp. 1–19.

[3] M. Herlihy, “A methodology for implementing highly concurrent data structures,” in *PPOPP ’90: Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming.* New York: ACM Press, 1990, pp. 197–206.

[4] B. Stroustrup, *The C++ Programming Language.* Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[5] D. Wagner, “Data management in the Mission Data System,” in *Proceedings of the IEEE System, Man, and Cybernetics Conference*, 2005, pp. 3788–3792.

[6] B. Stroustrup and G. D. Reis, “Supporting SELL for high-performance computing,” in *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing, LCPC*, 2005, pp. 458–465.

[7] D. Dechev, P. Pirkelbauer, and B. Stroustrup, “Lock-free dynamically resizable arrays.” in *OPODIS*, ser. *Lecture Notes in Computer Science*, A. A. Shvartsman, Ed., vol. 4305. Bordeaux, France: Springer, 2006, pp. 142–156.

[8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming.* Cambridge, MA: Morgan Kaufmann, 2008.

[9] R. Rasmussen, M. Ingham, and D. Dvorak, "Achieving control and interoperability through unified model-based engineering and software engineering," in *AIAA Infotech at Aerospace Conference*, 2005, pp. 177–189.

[10] RTCA, "Software considerations in airborne systems and equipment certification (DO-178B)," 1992. [Online]. Available: www.assconline.co.uk/assc_do-178b.asp

[11] R. Volpe and S. Peters, "Rover technology development and mission infusion for the 2009 Mars Science Laboratory Mission,," in *7th International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, May 2003, pp. 67–78.

[12] A. Stoica, D. Keymeulen, A. Csaszar, Q. Gan, T. Hidalgo, J. Moore, J. Newton, S. Sandoval, and J. Xu, "Humanoids for lunar and planetary surface operations," in *Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics*, October 2005, pp. 2649–2654.

[13] P. Wooster, W. Hofstetter, and E. Crawley, "Crew exploration vehicle for human lunar exploration: the lunar surface," in *Proceedings of the AIAA 2005*, August 2005, pp. 871–892.

[14] National Aeronautics and Space Administration, Columbia Accident Investigation Board, "Columbia Accident Investigation Board Report Volume 1." [Online]. Available: http://caib.nasa.gov/

[15] J. Schumann and W. Visser, "Autonomy software: V&V challenges and characteristics," in *Proceedings of the 2006 IEEE Aerospace Conference*, 2006, pp. 1233–1249.

[16] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, R. Washington, and W. Visser, "Experimen-

tal evaluation of verification and validation tools on Martian Rover software," in *Formal Methods in Systems Design Journal*, Sept. 2005, pp. 167–198.

[17] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The CLARAty architecture for robotic autonomy," in *IEEE Aerospace Conference*, March 2001, pp. 121–132.

[18] D. Dvorak, G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reiinholtz, "Project Golden Gate: Towards real-time Java in space missions," in *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, 2004, pp. 15–22.

[19] D. Dvorak, "Challenging encapsulation in the design of high-risk control systems," in *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, 2002, pp. 87–99.

[20] B. Boehm, J. Bhuta, D. Garlan, E. Gradman, L. Huang, A. Lam, R. Madachy, N. Medvidovic, K. Meyer, S. Meyers, G. Perez, K. Reinholtz, R. Roshandel, and N. Rouquette, "Using empirical testbeds to accelerate technology maturity and transition: The SCRover experience," in *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering.* Washington, DC: IEEE Computer Society, 2004, pp. 117–126.

[21] C. Perrow, *Normal Accidents.* Princeton, NJ, Princeton University Press, 1999.

[22] E. Lee and S. Neuendorffer, "Concurrent models of computation for embedded software," in *IEEE Proceedings on Computers and Digital Techniques*, March 2005, pp. 239–250.

[23] International Organization for Standardization, ISO/IEC 14882 International Standard, *Programming languages C++.* Geneva, Switzerland, American National Standards Institute:, Sept. 1998.

[24] K. Fraser and T. Harris, "Concurrent programming without locks," *ACM Trans. Comput. Syst. (TOCS)*, vol. 25, no. 2, p. 5, 2007.

[25] Intel, "Reference for Intel threading building blocks, version 1.0, Santa Clara, CA," April 2006.

[26] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott, "Nonblocking transactions without indirection using alert-on-update," in *SPAA '07: Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures.* New York: ACM, 2007, pp. 210–220. [Online]. Available: http://www.cs.rochester.edu/research/synchronization/rstm/v4api.shtml

[27] Intel, "*IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*," Santa Clara, CA, 2004.

[28] D. Dechev and B. Stroustrup, "Model-based product-oriented certification," in *16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2009), IEEE Computer Society*, 2009, pp. 244–253.

[29] M. Gibbs and B. Stroustrup, "Fast dynamic casting," *Software Practice and Experience*, vol. 36, no. 2, pp. 139–156, 2006.

[30] G. Brat, E. Denney, D. Giannakopoulou, J. Frank, and A. Jonsson, "Verification of autonomous systems for space applications," in *IEEE Aerospace Conference*, March 2006, pp. 1389–1397.

[31] J. U. Gärtner, "Certified software factory: Open software toolsuites, safe methodologies and system architectures," in *SCS '06: Proceedings of the Eleventh Australian Workshop on Safety Critical Systems and Software.* Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006, pp. 19–22.

[32] S.-W. Lee, R. A. Gandhi, and S. Wagle, "Towards a requirements-driven workbench for supporting software certification and accreditation," in *SESS '07: Proceedings of the Third International Workshop on Software Engineering for Secure Systems.* Washington, DC: IEEE Computer Society, 2007, pp. 8–9.

[33] E. Denney and B. Fischer, "Certifiable program generation," in *Proceedings of GPCE'05: 4th International Conference on Generative Programming and Component Engineering*, 2005, pp. 17–28.

[34] M. R. Lowry, T. Pressburger, and G. Rosu, "Certifying domain-specific policies." in *Automated Software Engineering (ASE).* IEEE Computer Society, 2001, pp. 81–90.

[35] E. Denney, B. Fischer, J. Schumann, and J. Richardson, "Automatic certification of Kalman filters for reliable code generation," in *Proceedings of the 2005 IEEE Aerospace Conference*, 2005, pp. 1–10.

[36] E. Denney and B. Fischer, "Extending source code generators for evidence-based software certification," in *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006)*, 2006, pp. 138–145.

[37] O. Shalev and N. Shavit, "Predictive log synchronization," in *Proceedings of the EuroSys 2006 Conference*, 2006, pp. 305–315.

[38] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems.* Washington, DC: IEEE Computer Society, 2003, p. 522.

[39] L. Lamport, "How to make a multiprocessor computer that correctly executes programs, *IEEE Trans. Comput.*" pp. 779–782, September 1979.

[40] A. Alexandrescu and M. Michael, "Lock-free data structures with hazard pointers," in *C++ User Journal*, November 2004, pp. 17–20.

[41] D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. S. Jr., "Even better DCAS-based concurrent deques," in *International Symposium on Distributed Computing*, 2000, pp. 59–73.

[42] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Proceedings of the 16th International Symposium on Distributed Computing*, 2002, pp. 155–167.

[43] K. Fraser, "Practical lock-freedom," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-579, Feb. 2004. [Online]. Available: http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-579.pdf

[44] G. Barnes, "A method for implementing lock-free shared-data structures," in *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures.* New York: ACM Press, 1993, pp. 261–270.

[45] D. Dice and N. Shavit, "Understanding tradeoffs in software transactional memory," in *Proc. of the 2007 International Symposium on Code Generation and Optimization (CGO)*, 2007, pp. 21–33.

[46] T. L. Harris, “A pragmatic implementation of non-blocking linked-lists,” in *DISC ’01: Proceedings of the 15th International Conference on Distributed Computing.* London, UK: Springer-Verlag, 2001, pp. 300–314.

[47] M. M. Michael, “High performance dynamic lock-free hash tables and list-based sets,” in *SPAA ’02: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures.* New York: ACM Press, 2002, pp. 73–82.

[48] M. Michael, “CAS-based lock-free algorithm for shared deques,” in *Euro-Par 2003: The Ninth Euro-Par Conference on Parallel Processing, LNCS volume 2790*, 2003, pp. 651–660.

[49] H. Sundell and P. Tsigas, “Lock-free and practical doubly linked list-based deques using single-word compare-and-swap.” in *OPODIS*, 2004, pp. 240–255.

[50] D. Hendler, N. Shavit, and L. Yerushalmi, “A scalable lock-free stack algorithm,” in *SPAA ’04: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures.* New York: ACM Press, 2004, pp. 206–215.

[51] O. Shalev and N. Shavit, “Split-ordered lists: Lock-free extensible hash tables,” in *PODC ’03: Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing.* New York: ACM Press, 2003, pp. 102–111.

[52] M. M. Michael, “Hazard pointers: Safe memory reclamation for lock-free objects,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, 2004.

[53] P. Becker, “ Working draft, standard for programming language C++, ISO WG21 N2009,” April 2006. [Online]. Available: http://www.open-std.org/JTC1/SC22/WG21/

[54] H. Boehm and S. Adve, "Foundations of the C++ concurrency memory model," in *PLDI '08: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation.* ACM Press, 2008, pp. 68–78.

[55] M. Herlihy, V. Luchangco, P. Martin, and M. Moir, "Nonblocking memory management support for dynamic-sized data structures," *ACM Trans. Comput. Syst.*, vol. 23, no. 2, pp. 146–196, 2005.

[56] M. M. Michael and M. L. Scott, "Correction of a memory management method for lock-free data structures," University of Rochester, Rochester, NY, Tech. Rep., December 1995.

[57] M. M. Michael, "Scalable lock-free dynamic memory allocation," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation.* New York: ACM Press, 2004, pp. 35–46.

[58] A. Gidenstam, M. Papatriantafilou, and P. Tsigas, "Allocating memory in a lock-free manner." in *ESA*, 2005, pp. 329–342.

[59] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* Cambridge, MA: MIT Press, 2001.

[60] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[61] D. Dechev, N. Rouquette, P. Pirkelbauer, and B. Stroustrup, "Verification and semantic parallelization of goal-driven autonomous software," in *Proceedings of ACM Autonomics 2008: 2nd International Conference on Autonomic Computing and Communication Systems*, 2008, pp. 1–8.

[62] G. Holzmann, *The Spin Model Checker, Primer and Reference Manual.* Reading, MA: Addison-Wesley, 2003.

[63] B. Stroustrup, *The Design and Evolution of C++.* New York: ACM Press/Addison-Wesley Publishing Co., 1994.

[64] D. Jackson, *Software Abstractions: Logic, Language and Analysis.* Cambridge, MA: The MIT Press, 2006.

[65] M. Sherriff and L. Williams, "DevCOP: A software certificate management system for Eclipse," in *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering.* Washington, DC: IEEE Computer Society, 2006, pp. 375–384.

[66] M. Herlihy, "The art of multiprocessor programming," in *PODC '06: Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing.* New York: ACM, 2006, pp. 1–2.

[67] D. Gifford and A. Spector, "Case study: IBM's system/360-370 architecture," *Commun. ACM*, vol. 30, no. 4, pp. 291–307, 1987.

[68] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[69] G. D. Reis and B. Stroustrup, "Specifying C++ concepts, ISO WG21 N1886," Geneva, Switzerland, Tech. Rep., Sept. 2005.

[70] D. Gregor and J. Järvi, "Variadic templates for C++," in *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing.* New York, NY: ACM, 2007, pp. 1101–1108.

[71] T. L. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, no. 5, pp. 26–31, Jun. 1995.

[72] E. Vries, R. Plasmeijer, and D. M. Abrahamson, "Uniqueness typing simplified," in *Implementation and Application of Functional Languages: 19th International Workshop*, Sept. 2007, pp. 19–27.

[73] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. S. Jr., "Lock-free reference counting," *Distributed Computing*, vol. 15, no. 4, pp. 255–271, 2002.

[74] M. F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M. L. Scott, "Alert-on-update: A communication aid for shared memory multiprocessors," in *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* New York: ACM, 2007, pp. 132–133.

[75] K. Reinholtz, "Atomic reference counting pointers," *C++ User Journal*, pp. 18–22, December 2008.

[76] R. Gluck and G. Holzmann, "Using SPIN model checker for flight software verification," in *Proceedings of the 2002 IEEE Aerospace Conference*, 2002, pp. 177–192.

[77] J. Lou, "An efficient algorithm for propagation of temporal constraint networks," *NASA Tech Brief Vol. 26 No. 4 from JPL New Technology Report NPO-21098*, April 2002. [Online]. Available: http://www.techbriefs.com/

[78] N. Rouquette, "Analyzing and verifying UML models with OCL and Alloy," *EclipseCon 2008*, 2008.

[79] D. Dvorak, R. Rasmussen, and T. Starbird, "State knowledge representation in the Mission Data System," in *Proceedings of IEEE Aerospace Conference*, 2002,

pp. 232–244.

[80] A. Barett, R. Knight, J. Morris, and R. Rasmussen, "Mission planning and execution within the Mission Data System," in *Proceedings of the International Workshop on Planning and Scheduling for Space*, 2004, pp. 97–105.

[81] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free," in *CODES/CASHE '98: Proceedings of the 6th International Workshop on Hardware/Software Codesign.* Washington, DC: IEEE Computer Society, 1998, pp. 97–101.

[82] M. Ingham, R. Rasmussen, M. Bennett, and A. Moncada, "Engineering complex embedded systems with wtate analysis and the Mission Data System," in *Proceedings of First AIAA Intelligent Systems Technical Conference*, 2004.

[83] B. Stroustrup, "Abstraction and the C++ machine model." in *ICESS*, ser. Lecture Notes in Computer Science, Z. Wu, C. Chen, M. Guo, and J. Bu, Eds., vol. 3605. Beijing, China: Springer, 2004, pp. 1–13.

[84] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *NSDI'04: Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation.* Berkeley, CA: USENIX Association, 2004, pp. 12–12.

[85] T. C. Ruys, "Optimal scheduling using branch and bound with spin 4.0," in *Model Checking Software, Proceedings of the 10th International SPIN Workshop*, ser. Lecture notes in Computer Science, T. Ball and S. K. Rajamani, Eds., vol. 2648. Berlin: Springer Verlag, 2003, pp. 1–17.

[86] S. Lerner, T. Millstein, and C. Chambers, "Automatically proving the correctness

of compiler optimizations," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation.* New York: ACM Press, 2003, pp. 220–231.

[87] E. Brinksma and A. Mader, "Verification and optimization of a PLC control schedule," in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification.* London, UK: Springer-Verlag, 2000, pp. 73–92.

[88] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization.* New York: Wiley-Interscience, 1988.

VITA

Damian Dechev

Contact Information

Parasol Laboratory	*phone:* (979) 739-4909
Department of Computer Science	*e-mail:* dechev@tamu.edu
Texas A&M University	*www:* damiandechev.com
College Station, TX 77843-3112 USA	

Education

Texas A&M University, College Station, Texas

- Ph.D., Computer Science, December, 2009
  - Research Advisor: Dr. Bjarne Stroustrup
  - GPA 4.0 / 4.0

University of Delaware, Newark, Delaware

- M.S., Computer Science, May, 2003
  - GPA 3.7 / 4.0

University of Indianapolis, Indianapolis, Indiana

- B.S., Computer Science, May, 2001
  - GPA 3.9 / 4.0

The typist for this thesis was Damian Dechev.