Semantically Enhanced Containers for Concurrent Real-Time Systems

Damian Dechev[1], Peter Pirkelbauer[1], Nicolas Rouquette[2], and Bjarne Stroustrup[1]
dechev@tamu.edu, peter.pirkelbauer@tamu.edu, nicolas.rouquette@jpl.nasa.gov, bs@cs.tamu.edu

Texas A&M University [1]
College Station, TX 77843-3112

Jet Propulsion Laboratory, California Institute of Technology [2]
4800 Oak Grove Drive, M/S 301-270, Pasadena, CA

Abstract

Future space missions, such as Mars Science Laboratory, are built upon computing platforms providing a high degree of autonomy and diverse functionality. The increased sophistication of robotic spacecraft has skyrocketed the complexity and cost of its software development and validation. The engineering of autonomous spacecraft software relies on the availability and application of advanced methods and tools that deliver safe concurrent synchronization as well as enable the validation of domain-specific semantic invariants. The software design and certification methodologies applied at NASA do not reach the level of detail of providing guidelines for the development of reliable concurrent software. To achieve effective and safe concurrent interactions as well as guarantee critical domain-specific properties in code, we introduce the notion of a Semantically Enhanced Container (SEC). A SEC is a data structure engineered to deliver the flexibility and usability of the popular ISO C++ Standard Template Library containers, while at the same time it is hand-crafted to guarantee domain-specific policies. We demonstrate the SEC proof-of-concept by presenting a shared nonblocking SEC vector. To eliminate the hazards of the ABA problem (a fundamental problem in lock-free programming), we introduce an innovative library for querying C++ semantic information. Our SEC design aims at providing an effective model for shared data access within the JPL's Mission Data System. Our test results show that the SEC vector delivers significant performance gains (a factor of 3 or more) in contrast to the application of nonblocking synchronization amended with the traditional ABA avoidance scheme.

1 Introduction

[1] Future space exploration projects, such as Mars Science Laboratory (MSL) [31] and Project Constellation [27], demand the design of some of the most complex man-rated software systems. The high degree of autonomy and increased complexity of such systems pose significant challenges in assuring their reliability and efficiency. A number of studies led by NASA ([25] and [21]) indicate that the current development and validation methodologies are prohibitively expensive for systems of such complexity. The challenges of developing and testing modern avionics software motivated NASA to initiate a number of advanced experimental software platforms, such as the Jet Propulsion Laboratory's (JPL) Mission Data System (MDS) [23]. MDS provides an experimental goal- and state-based platform for testing and development of autonomous real-time flight applications.

In this work 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 [28], 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. The objective of our work is to introduce the notion, present an initial implementation, and demonstrate the benefits of Semantically Enhanced Containers. The most ubiquitous and versatile data structure in the ISO C++ Standard Template Library is *vector*, offering a combination of dynamic memory management and constant-time random access. 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 ensure *safe and efficient concurrent synchronization* as well as offer the mechanisms to establish the validity of certain *user-defined semantic guarantees* in concurrent real-time systems. Our SEC vector's implementation is based on 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 [14]. In addition, our design is portable: all of the vector's algorithms are based on the word-size compare-and-swap (CAS) instruction [10] available on a large number of hardware platforms

(b) *ensure the validity of user-defined semantic invariants:* we introduce Basic Query (BQ), an innovative 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 [2] (a fundamental problem for all CAS-based designs) in the usage of our concurrent vector.

A number of pivotal concurrent applications in the Mission Data System framework employ a shared STL vector (in all scenarios protected by mutually exclusive locks). Such applications are the Data Management Service containers [32], the Goal Checker - an application for monitoring the status of goals, and Elf - a framework for message passing and transportation. Lock-free programming techniques have been applied in [3] to devise a methodology for automatic parallelization of the MDS Temporal Constraint Network (TCN) library. The TCN library is a critical component of the MDS platform, defining the concepts of goal-driven autonomous behavior. However, the shared containers used in [3] does not employ an ABA prevention scheme and under certain conditions might be vulnerable to the occurrence of ABA.

This paper presents the concept, design, and implementation of a SEC shared vector engineered to provide higher safety and faster performance in a number of critical MDS applications. The rest of the paper is organized like this: Section 2: Motivation, Section 3: Nonblocking Synchronization, Section 4: Semantic Enhancement, Section 5: Performance Results, and Section 6: Conclusion.

2 Motivation

A detailed study on the challenges for the development and certification of modern spacecraft software by Lowry [21] reveals that in July 1997 The Mars Pathfinder mission experienced a number of anomalous system resets that caused an operational delay and loss of scientific data. The follow-up analysis identified the presence of a priority inversion problem caused by the low-priority meteorological process blocking the the high-priority bus management process. The software engineers found out that it would have been impossible to detect the problem with the black box testing applied at the time. A more appropriate 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 software are of critical importance to the system's safety and operation. Despite the challenges in debugging and verification of the system's concurrent components, the existing development and certification processes [24] do not provide guidelines at the level of detail reaching the development, application, and testing of concurrent programs.

3 Nonblocking Synchronization

The most common technique for controlling the interactions of concurrent processes is the use of mutual exclusion locks. A mutual exclusion lock guarantees thread-safety of a concurrent object by blocking all contending threads trying to access it except the one holding the lock. In scenarios of high contention on the shared data, such an approach can seriously affect the performance of the system and significantly 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 mutually exclusive locks poses significant safety hazards 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 [18]. Often due to the resource limitations of flight-qualified hardware, optimized lock mechanisms are not a desirable alternative [21]. 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 determine with the traditional testing procedures and a program can be deployed and used for a long period of time before the flaws can become evident and eventually cause anomalous behavior.

To achieve scalability, better performance, reliability, and avoid the dangers of priority inversion, deadlock, and livelock, we rely on the notion of *lock-free synchronization*. As defined by Herlihy [14], a concurrent object is *nonblocking* (lock-free) 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, such as the word-size CAS instruction. Lock-free systems typically utilize CAS in order to implement a an optimistic speculation on the shared data. A contending process attempts to make progress by applying one or more writes on a local copy of the shared data. Afterwards, the process attempts to swap (CAS) the global data with its updated copy. Such an approach guarantees that from within a set of contending processes, there is at least one that succeeds within a finite number of steps.

The system is non-blocking at the expense of some extra work performed by the contending processes. Linearizability [14] is an important correctness condition for concurrent nonblocking objects: a concurrent operation is linearizable if it appears to execute instantaneously in a given point of time between the time t_1 of its invocation and the time t_2 of its completion. The consistency model implied by the linearizability requirements is stronger than the widely applied Lamport's sequential consistency model [20]. 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). The nonblocking synchronization algorithms in our SEC vector are derived from the first design of a lock-free dynamically resizable array presented by Dechev at el. in [2]. The operations of our SEC vector are lock-free and linearizable and in addition they provide disjoin-access parallelism for random access reads and writes and fast execution (outperforming the STL vector protected by a mutex by a factor of 10 or more [2]).

3.1 An Alternative to CAS-based designs: Software Transactional Memory (STM)

A variety of recent STM approaches [6] claim safe and easy to use concurrent interfaces. The most advanced STM implementations allow the definition of efficient "large-scale" transactions, i.e. *dynamic* and *unbounded* transactions. *Dynamic transactions* are able to access memory locations that are not statically known. *Unbounded transactions* pose no limits 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. 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 (including our SEC vector). The complex designs of such advanced STMs often come with an associated cost:

(1) *Two Levels of Indirection:* A large number of the nonblocking designs require two levels of indirection in accessing data

(2) *Linearizability:* The linearizability requirements are hard to meet for an unbounded and dynamic STM. To achieve efficiency and reduce the complexity, many STMs offer the less demanding *obstruction-free* synchronization [17]

(3) *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

(4) *Closed Memory Usage:* Both nonblocking and lock-based STMs require a *closed memory system*

(5) *Vulnerability of Large Transactions:* In a nonblocking implementation large transactions are a subject to interference from contending threads and thus are more likely to encounter conflicts. Large blocking transactions can be subject to timeouts, requests to abort or simply introduce a bottleneck for the computation

(6) *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 SEC vector implementation, the main problems associated with the application of STM are meeting the stricter requirements posed by the linearizability model and the overhead introduced by the application of the costly conflict detection and validation schemes. Because of these trade-offs present in the state of the art STM libraries, our current SEC vector design is based on the utilization of nonblocking and portable CAS-based algorithms to deliver the targeted safe and reliable concurrent synchronization.

3.2 Practical Lock-Free Programming Techniques

The practical implementation of a hand-crafted lock-free container is notoriously difficult. A nonblocking container's design suggests the update (in a linearizable fashion) of several memory locations. The use of a double-compare-and-swap primitive (DCAS) has been suggest by Detlefs et al. in [5], however such complex atomic operations are rarely supported by the hardware architecture. Harris et al. propose in [13] 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 [9]. 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. The vector's random access, data locality, and dynamic memory management pose serious challenges for its non-blocking implementation. To illustrate the complexity of a CAS-based design of a dynamically resizable array, Table 1 provides an analysis of

the number of memory locations that need to be updated upon the execution of some of the vector's basic operations.

	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*

Table 1. Vector - Operations

3.3 Overview of the Lock-free Operations

In this section we present a brief overview of the most critical lock-free algorithms employed by our SEC vector (see [2] for the full set of the operations of the first lock-free dynamically resizable array). To help tail operations update the size and the tail of the vector (in a linearizable manner), the design presented in [2] suggests the application of of a helper object, named "`Write Descriptor (WD)`" that announces a pending tail modifications and allows interrupting threads help the interrupted thread complete its operations. A pointer to the WD object is stored in the "`Descriptor`" together with the container's size and a reference counter required by the applied memory management scheme. The approach requires that data types bigger than word size are indirectly stored through pointers and 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 remaining part of this section presents the pseudo-code of the tail operations (*push_back* and *pop_back*) and the random access operations (*read* and *write* at a given location within the vector's bounds). We use the symbols ^, &, and . to indicate pointer dereferencing, obtaining an object's address, and integrated pointer dereferencing and field access respectively.

Algorithm 1 pushback $vector, elem$

```
1: repeat
2:     desc_current ← vector.desc
3:     CompleteWrite(vector, desc_current.pending)
4:     if vector.memory[bucket] = NULL then
5:         AllocBucket(vector, bucket)
6:     end if
7:     wop ←
       new WriteDesc(At(desc_current.size)^, elem, desc_current.size)
8:     desc_next ← new Descriptor(desc_current.size + 1, wop)
9: until CAS(&vector.desc, desc_current, desc_next)
10: CompleteWrite(vector, desc_next.pending)
```

Push_back (add one element to 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`". It 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.

Algorithm 2 Read $vector, i$

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

Algorithm 3 Write $vector, i, elem$

```
1: At(vector, i)^ ← elem
```

Algorithm 4 popback $vector$

```
1: repeat
2:     desc_current ← vector.desc
3:     CompleteWrite(vector, desc_current.pending)
4:     elem ← At(vector, desc_current.size − 1)^
5:     desc_next ← new Descriptor(desc_current.size − 1, NULL)
6: until CAS(&vector.desc, desc_current, desc_next)
7: return elem
```

Algorithm 5 CompleteWrite $vector, wop$

```
1: if wop.pending then
2:     CAS(At(vector, wop.pos), wop.value_old, wop.value_new)
3:     wop.pending ← false
4: end if
```

3.4 The ABA Problem

The ABA problem [22] is fundamental to all CAS-based systems. There are two particular instances that create ABA hazards:

(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

The ABA problem can occur in the CAS-based design of a nonblocking dynamic array (Section 3.3) in a number of possible ways. One possible hazardous execution can happen like this: assume a thread T_0 attempts to perform a `push_back`; in the vector's "`Descriptor`", `push_back` stores an announcement declaring that the

value of the object at position i should be changed from A to B. Then a thread T_1 interrupts and reads the *Descriptor Object*. Later, after T_0 resumes and successfully completes the operation, a third thread T_2 can modify the value at position i from B back to A. When T_1 resumes its CAS is going to succeed and erroneously execute the update from A to B.

As a common technique for overcoming the ABA problem it has been suggested to use a version tag attached to each value. Such an approach demands the application of an atomic instruction such as a CAS2 (compare-and-swap two co-located words), a hardware primitive that is available on some common Intel architectures. Alternative hardware primitives that help us eliminate ABA are the DCAS (compare and swap two non-colacated words, implemented on some Motorola 68000 processors) or the load-link/store-conditional (LL/SC) semantics, provided by Alpha, PowerPC, MIPS, and ARM architectures. For our nonblocking implementation we cannot assume the availability of such atomic primitives since they are specific to a limited number of hardware platforms. A proposed hardware implementation (entirely built into a present cache coherency protocol) of an innovative Alert-on-Update (AOU) instruction [26] has been suggested by Spear et al. to eliminate the CAS deficiency of allowing ABA. It is unlikely that CAS2, DCAS, LL/SC or AOU would be supported by an embedded real-time hardware, such as the hardware platform on the Mars Science Laboratory.

ABA avoidance on CAS-based architectures has been typically limited to two possible approaches:

(a) split a 32-bit memory word into a value and a counter portions (thus significantly limiting the usable address space or the range of values that can be stored) [8]

(b) apply value semantics (by utilizing an extra level of indirection, i.e. create a unique pointer to each value to be stored) in combination with a memory management approach that disallows the reuse of potentially hazardous memory locations [16], [22] (thus impose a significant performance overhead)

To eliminate the ABA problem of (2), the authors in [2] suggest the application of a memory management scheme such as Herlihy et al.'s *Pass The Buck* algorithm [15] 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. This approach is undesirable because it introduces an extra level of indirection and its overhead might reduce the vector's performance gains. In this work we suggest the avoidance of the ABA problem by enforcing (through the application of static analysis) of a number of semantic usage patterns of the SEC vector (Section 4.1).

4 Semantic Enhancement

As emphasized by Stroustrup in [28], 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. Stroustrup explains [28]:

> *"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 autonomous embedded flight software, such as the Mission Data System, further illustrates 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 [11] and Alloy Analyzer [19] aim at expressing and validating 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 [4] 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. According to the FAA certification standards, it is required to establish the preservation of the program 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

4.1 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). IPR is at the center of a C++ static analysis framework named The Pivot [29]. We demonstrate BQ's efficiency by defining the semantic rule *Exclude_push_back* that disallows the use of a $push\_back$ operation in certain scenarios and helps us avoid the ABA problem (Section 3.4).

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 [1]. The Pivot represents C++ programs in two distinct formats (Figure 1):

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 (a single token look ahead and no symbol table needed)

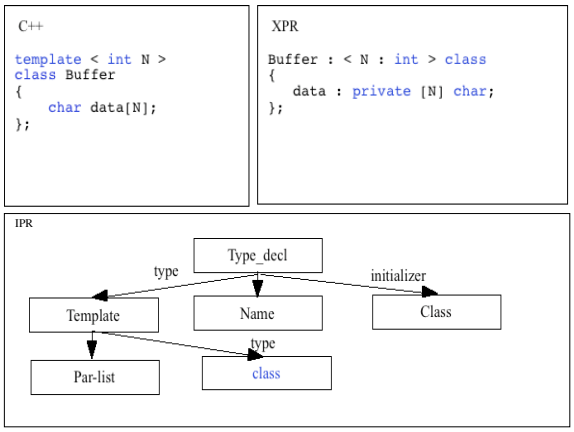

Figure 1. An XPR and IPR representations of a C++ template class definition

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 [7], 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 way, 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 each of these classes with two compile-time arguments, a TP (trigger point), identifying the exact point in the AST of triggering the action, 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 it is preferred 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 [12] and the C++ STL Iterator [28] 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 [30] (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 2). 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 3 and Table 4). A BQ element specifies one or several edges in the pattern tree. A BQ element could be one of three possible types:

1. $Exe\_member < x, e >$. (EM) generates a straightforward edge e from an IPR node 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. $Exe\_condition < x, e, c >$. (EC) generates an edge e from an IPR node x, only if a specified boolean condition c is met

3. $Exe\_iprseq < x, e_n >$. (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 ($x- > bases()$).

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

Table 2. BQ 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 3. Application of the BQ operations

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 (Section 3.4). In a number of MDS concurrent applications, there are multiple reader threads but only a single writer. Such a scenario is $ABA - free$ since it is not possible to have an interrupting writer thread placing the 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 5 enumerates all possible interleavings of two concurrent operations of the SEC vector and indicates those prone to ABA and those that are ABA-free. Thus our 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 6 and 7 we show the pseudo-code and the actual source code of the semantic rule $Exclude\_push\_back$ defined using the

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

Table 4. Result description of the BQ operations

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 6) and then test whether a function call's name is "push_back" (expressed by the $EC1$ element in Algorithm 6).

operation	$push$	pop	$read$	$write$
$push$	$ABA\ free$	ABA	$ABA\ free$	ABA
pop	$ABA\ free$	ABA	$ABA\ free$	$ABA\ free$
$read$	$ABA\ free$	$ABA\ free$	$ABA\ free$	$ABA\ free$
$write$	ABA	$ABA\ free$	$ABA\ free$	ABA

Table 5. ABA-free and ABA-prone interleaving of two concurrent operations

Algorithm 6 $Exclude\_push\_back$: find an illegal push_back

1: $RCO:\ ipr :: Expr$
2: $EM1:\ ipr :: Function\_call- > name$
3: $EC1:\ ipr :: String- > name\_cmp$
4: $Exclude\_push\_back:\ RCO < EM1 < EC1- > bool$

Algorithm 7 $Exclude\_push\_back$: 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: $Exe\_condition < ipr :: String,\ name,\ constipr :: Name\&,\ std :: string > Is\_Name(\&name\_cmp,\ parent\_name)$;
5: $return\ RCO < Get\_Name < Is\_Name$;

5 Performance Results

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 10.5.1 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 [2]. To prevent ABA we employed the traditional ABA avoidance technique used in CAS-based designs (see Section 3.4), 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 [15].

Similarly to the evaluation of other lock-free concurrent containers [9], 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.

As discussed by Fraser [9], it has been observed that in real-world concurrent application, the read operations dominate (and account to more than 50% of all operations). For this reason we illustrate the performance of the concurrent vectors with a distribution of +:16%, −:16%, w:18%, r:50% on Figure 2B. Figure 2A demonstrates the performance results with a distribution containing predominantly writes, +:25%, −:25%, w:12%, r:38%. In these diagrams, 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.

6 Conclusion

This paper introduced the concept and initial implementation of the notion of *Semantically Enhanced Containers (SECs)*. SECs are data structures that are engineered to provide the flexibility of the popular ISO C++ Standard Template Library (STL) containers, while at the same time they are hand-crafted to guarantee domain-specific policies (such as conformance to a given concurrency model).

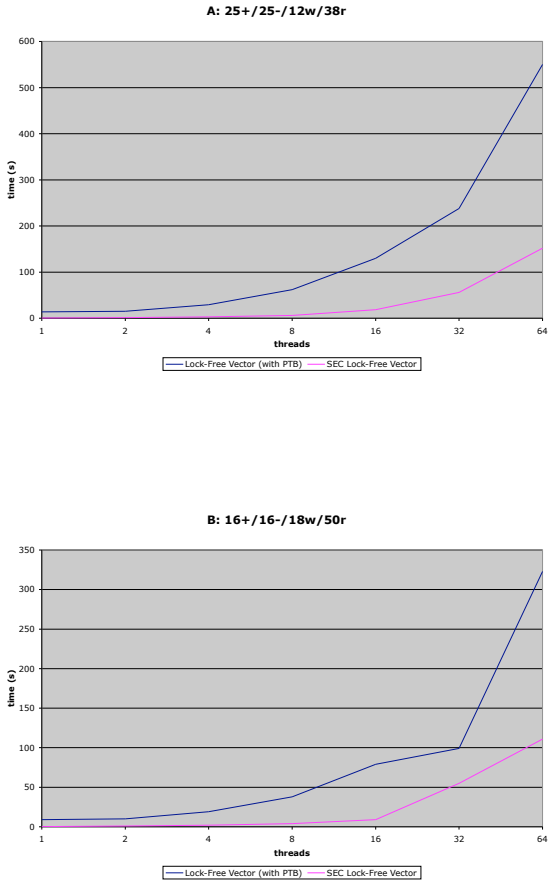

Figure 2. Performance Results

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 functionality 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 innovative 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.

References

[1] P. Becker. Working Draft, Standard for Programming Language C++, ISO WG21 N2009, April 2006.

[2] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-Free Dynamically Resizable Arrays. In A. A. Shvartsman, editor, *OPODIS*, volume 4305 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2006.

[3] 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.

[4] E. Denney, B. Fischer, J. Schumann, and J. Richardson. Automatic Certification of Kalman Filters for Reliable Code Generation. In *In Proceedings of the 2005 IEEE Aerospace Conference*, 2005.

[5] 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*, pages 59–73, 2000.

[6] 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.

[7] A. Duret-Lutz, T. Graud, and A. Demaille. Design Patterns for Generic Programming in C++. USENIX COOTS, 2001.

[8] D. Dvorak and W. Reinholtz. Hard real-time: C++ versus RTSJ. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 268–274, New York, NY, USA, 2004. ACM.

[9] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[10] D. Gifford and A. Spector. Case study: IBM's system/360-370 architecture. *Commun. ACM*, 30(4):291–307, 1987.

[11] R. Gluck and G. Holzmann. Using SPIN Model Checker for Flight Software Verification. In *In Proceedings of the 2002 IEEE Aerospace Conference*, 2002.

[12] A. Gurtovoy and D. Abrahams. The Boost C++ Metaprogramming Library. Technical report, 2001, 2003.

[13] 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.

[14] M. Herlihy. The art of multiprocessor programming. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 1–2, New York, NY, USA, 2006. ACM.

[15] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.

[16] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 339–353, London, UK, 2002. Springer-Verlag.

[17] 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*, page 522, Washington, DC, USA, 2003. IEEE Computer Society.

[18] Intel. Reference for Intel Threading Building Blocks, version 1.0, April 2006.

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

[20] L. Lamport. How to make a multiprocessor computer that correctly executes programs, September 1979.

[21] M. R. Lowry. Software Construction and Analysis Tools for Future Space Missions. In J.-P. Katoen and P. Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2002.

[22] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[23] 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.

[24] RTCA. Software Considerations in Airborne Systems and Equipment Certification (DO-178B), 1992.

[25] J. Schumann and W. Visser. Autonomy Software: V&V Challenges and Characteristics. In *In Proceedings of the 2006 IEEE Aerospace Conference*, 2006.

[26] 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*, pages 132–133, New York, NY, USA, 2007. ACM.

[27] 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 *In Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics*, October 2005.

[28] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[29] B. Stroustrup and G. D. Reis. Supporting SELL for High-Performance Computing. In *In Proceedings of the International Workshop on Languages and Compilers for Parallel Computing, LCPC 2005*, 2005.

[30] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[31] 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.

[32] D. Wagner. Data Management in the Mission Data System. In *In Proceedings of the IEEE System, Man, and Cybernetics Conference*, 2005.