Programming and Validation Techniques for Reliable Goal-driven Autonomic Software

Damian Dechev, Nicolas Rouquette, Peter Pirkelbauer and Bjarne Stroustrup

Abstract [1] Future space missions such as the Mars Science Laboratory demand the engineering of some of the most complex man-rated autonomous software systems. According to some recent estimates, the certification cost for mission-critical software exceeds its development cost. The current process-oriented methodologies 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 a first concurrency and time centered framework for verification and semantic parallelization of real-time C++ within the JPL Mission Data System Framework (MDS). The end goal of the industrial project that motivated our work is to provide certification artifacts and accelerated testing of the complex software interactions in autonomous flight systems. As a case study we demonstrate the verification and semantic parallelization of the MDS Goal Networks.

Damian Dechev
Texas A&M University, College Station, TX 77843-3112, e-mail: dechev@tamu.edu

Nicolas Rouquette
Jet Propulsion Laboratory, NASA/California Institute of Technology, e-mail: nicolas.rouquette@jpl.nasa.gov

Peter Pirkelbauer
Texas A&M University, College Station, TX 77843-3112, e-mail: pirkelp@tamu.edu

Bjarne Stroustrup
Texas A&M University, College Station, TX 77843-3112, e-mail: bs@cs.tamu.edu

[1] This is the authors' version of the work. It is posted here by permission of the publisher. Not for redistribution. The definitive version is published in Book Chapter in Autonomic Communication, Vasilakos, A.; Parashar, M.; Karnouskos, S.; Pedrycz, W. (Eds.), ISBN: 978-0-387-09752-7, Springer, May 2009.

1 Introduction

In this work we describe the design, implementation, and application of a first *concurrency* and *time* centered framework for verification and semantic parallelization of real-time C++ within the JPL Mission Data System Framework (MDS). MDS provides an experimental goal- and state- based platform for testing and development of autonomous real-time flight applications[22]. The end goal of the industrial project that motivated our work is to provide certification artifacts and accelerated testing of the complex software interactions in autonomous flight 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[6]. Providing such certification evidence may require the application of a number of software development, analysis, verification, and validation techniques[20]. The dominant paradigms for software development, assurance, and management at NASA rely on the principle "test-what-you-fly and fly-what-you-test". This methodology had 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 the DO-178B[25], the Federal Aviation Administration (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 the Mars Science Laboratory (MSL), Project Constellation, and the development of the Crew Launch Vehicle (CLV) and the Crew Exploration Vehicle (CEV) suggest the engineering of some of the most complex man-rated software systems. As stated in the Columbia Accident Investigation Board Report[3], 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 [26] suggests that the current certification methodologies are prohibitively expensive for systems of such complexity. A detailed analysis by Lowry[20] 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 the Mission Data System (MDS)[22], as well as a number of program synthesis, modeling, analysis, and verification techniques and tools, such as The JavaPathFinder[2], the CLARAty project[29], Project Golden Gate[10], The New Millenium Architecture Prototype (NewMAAP)[9]. The high cost and demands of man-rated certification have motivated the experimental development of several accelerated testing platforms[1]. A great number of the experimental faster-than-real-time flight software simulators require the parallelization of previously sequential real-time algorithms. In this work we present the design and implementation of a first *concurrency* and *time* centered framework for *verification* and *semantic parallelization* of real-time C++ within the JPL Mission Data System Framework. 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' Goal Networks, a critical component of the JPL's Mission Data System.

2 Challenges for Mission Critical Autonomous Software

In [21] Perrow 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. In this work, we argue that 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[18], the major contributing factors to the development and design complexity of such 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.

2.1 Parallelism and Complexity

The most commonly applied 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[15] or context-switching. Often due to the resource limitations of flight-qualified hardware, optimized lock mechanisms are not a desirable alternative[20]. 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.

2.1.1 Parallel Programming without Locks

To achieve higher safety and enhance the performance of our implementation, we consider the application of *lock-free synchronization*. As defined by Herlihy[14], a concurrent object is *non-blocking* (lock-free) if it guarantees that *some* process in the system will make progress in a *finite* amount of steps. Non-blocking 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 [27] is *vector*, offering a combination of dynamic memory management and constant-time random access. In our framework for verification and semantic parallelization of real-time C++ we utilize the design of the first lock-free design and implementation of a dynamically-resizable array in ISO C++ (Section 5). It provides linearizable operations, disjoin-access parallelism for random access reads and writes, lock-free memory allocation and management, and fast execution.

2.2 Motivation and Contributions

As discussed by Lowry[20], 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 study identified the presence of a priority inversion problem caused by the low-priority meteorological process blocking the high-priority bus management process. It has been determined 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 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 certification process[25] 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. In [22] Rasmussen et al. suggest that the challenges for these systems' control is 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, enabling the agents to respond and recover from anomalies, guaranteeing the system's correctness and reliability, and ensuring effective communication across the system's components. In the rest of the paper we describe the definition, design, and implementation of a first *concurrency* and *time* centered framework for verification and semantic parallelization of autonomous flight software within the JPL's MDS Framework. We integrate a nonblocking vector in our parallel implementation of the Mission Data System's Temporal Constraint Network Library (TCN) in order to achieve higher thread safety and boost the performance of the MDS Goal Networks component. We demonstrate how to specify, model, and formally verify the TCN algorithms and their semantic invariants. Based on our formal models and the application's semantics, we derive a technique for automatic and semantic parallelization of the TCN library's constraint propagation algorithm.

3 Temporal Constraint Networks

A Temporal Constraint Network (TCN) defines the goal-oriented operation of a control system in the context of a system under control. The Temporal Constraint Networks (TCN) application is at the core of the Jet Propulsion Laboratory's Mission Data System (MDS)[22] state-based and goal-oriented unified architecture for testing and development of mission software. The framework's state- and model-based methodology and its associated systems engineering processes and development tools have been successfully applied on a number of test applications including the physical rovers Rocky 7 and Rocky 8 and a simulated Entry, Descent, and Landing (EDL) component for the Mars Science Laboratory mission. 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 elaborated plan initially created as a solution for a set of goals. The topology of a temporal constraint network must satisfy a number of invariants.

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

(b) For each time point $TP_i \in S_{tps}$, there is a set of temporal constraints that are immediate successors (S_{succ_i}) of TP_i and a set, S_{pred_i}, consisting of all of TP_i's immediate predecessors
(c) Each temporal constraint $TC_j \in S_{tcs}$ has exactly one successor TP_{succ_j} and one predecessor TP_{pred_j}
(d) For each pair $\{TP_i, TC_j\}$, where $TP_i \equiv TC_{succ_j}$, $TC_j \in S_{pred_i}$ must hold. The reciprocal invariant must also be valid, namely for each pair of $\{TP_i, TC_j\}$ such that $TP_i \equiv TC_{pred_j}$, $TC_j \in S_{succ_i}$
(e) The firing window of a time point $TP_i \in S_{tps}$ is represented by the pair of time instances $\{TP_{min_i}, TP_{max_i}\}$. Assuming that the current moment of time is represented by T_{now}, then $TP_{min_i} \leq T_{now} \leq TP_{max_i}$, for every $TP_i \in S_{tps}$.

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 result in a slow and cumbersome solution. However, any implementation (in C++, Java or another programming language) must operate under the assumptions 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.

4 Verification and Automatic Parallelization Framework

In this section 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 1). The input to the framework is the MDS mission planning and execution module that is based on the definition of temporal constraint networks. 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 in [19]. Constraint propagation poses performance challenges and speed bottlenecks due to the algorithm's frequent execution and the necessary real-time update 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 Section 3) and *automatically* derive an implementation that can be executed concurrently on one of the JPL's experimental testbeds for accelerated testing[1]. 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[16] that

represents the implementation's core semantics and functionality. We refine a formal modeling and analysis methodology, initially suggested by Rouquette[24], that helps us analyze the logical properties of the goal network model and automatically derive a meta-model for our parallel solution.

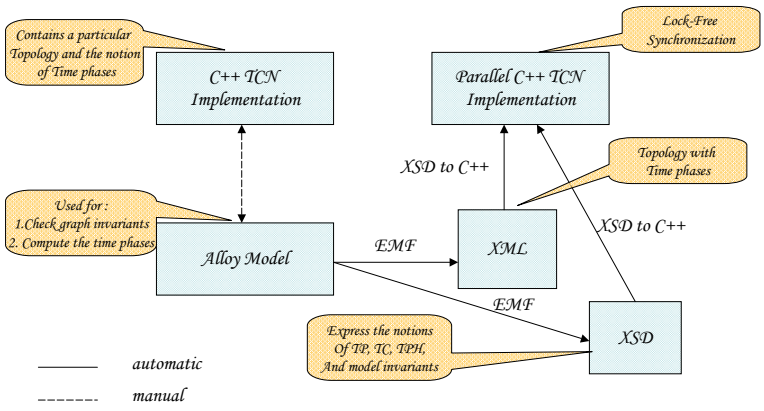

Fig. 1 A Framework for Verification and Semantic Parallelization

4.1 The Problem of 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[4], 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. In [19], Lou describes an innovative and effective 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 consid-

ering only its incoming temporal constraints (Algorithm 1). Similarly, a backward pass recomputes the time windows at each time point by considering only its outgoing temporal constraints (Algorithm 2). 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 [19], prior to the execution of the optimized propagation scheme, it is critical 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.

$min_{tmp} \leftarrow tp.min$;
$max_{tmp} \leftarrow tp.max$;
for j = 0 *to* $tp.preds\_size$ **do**
  $min_{tmp} \leftarrow$ std::max($min_{tmp}, tp.preds[j].pred.min + tp.preds[j].min$);
  $max_{tmp} \leftarrow$ std::min($max_{tmp}, tp.preds[j].pred.max + tp.preds[j].max$);
end
if $tp.min != min_{tmp}$ **then**
  ASSERT($tp.min < min_{tmp}$);
  $tp.min \leftarrow min_{tmp}$;
  vstate.aIncr(vstate.count);
  `/* atomically increment the state vector's counter */`
end
if $tp.max != max_{tmp}$ **then**
  ASSERT($tp.max > max_{tmp}$);
  $tp.max \leftarrow max_{tmp}$;
  vstate.aIncr(vstate.count);
  `/* atomically increment the state vector's counter */`
end
return $!(min_{tmp} > max_{tmp})$;

Algorithm 1: 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)

$min_{tmp} \leftarrow tp.min$;
$max_{tmp} \leftarrow tp.max$;
for j = 0 *to* $tp.succs\_size$ **do**
  $min_{tmp} \leftarrow$ std::max(min_{tmp}, $tp.succs[j].succ.min - tp.succs[j].max$);
  $max_{tmp} \leftarrow$ std::min(max_{tmp}, $tp.succs[j].succ.max - tp.succs[j].min$);
end
if $tp.min != min_{tmp}$ **then**
  ASSERT($tp.min < min_{tmp}$);
  $tp.min \leftarrow min_{tmp}$;
  vstate.aIncr(vstate.count);
  /* atomically increment the state vector's counter */
end
if $tp.max != max_{tmp}$ **then**
  ASSERT($tp.max > max_{tmp}$);
  $tp.max \leftarrow max_{tmp}$;
  vstate.aIncr(vstate.count);
  /* atomically increment the state vector's counter */
end
return $!(min_{tmp} > max_{tmp})$;

Algorithm 2: 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)

4.2 Modeling, Formal Verification, and Automatic Parallelization

Alloy[16] 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[12]. We use the Alloy specification language[16] to formally represent and check the semantics of the temporal constraint networks library (Algorithm 3) and its main invariants (Algorithm 4). In our C++ goal networks implementation we have applied generic programming techniques and concepts[23], 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. In the future, we intend to utilize a static analysis tool such as The Pivot[28] in order to automate this transition (this is the last non-automated component of the presented framework).

In addition, 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 $[TP_{min_i}, TP_{max_i}]$ for each $TP_i \in S_{tps}$, 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 (S_{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 $[TP_{min_a}, TP_{max_a}]$ and $[TP_{min_b}, TP_{max_b}]$ for every pair of $\{TP_a, TP_b\}$, such that $TP_a \in S_{Tph_i}$ and $TP_b \in S_{Tph_i}$, 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 5 and Algorithm 6). Figure 2 provides an example of a goal network containing 15 time points and 6 time phases.

```
/* declaration of the Temporal Constraint signature
   */
sig TC { tc_pred: one TP, tc_succ: one TP} ;
/* declaration of the Time Point signature          */
sig TP { tp_preds: set TC, tp_succs: set TC} ;
```

Algorithm 3: Definition of the notions of Temporal Constraint and Time Point

```
all tc:TC | tc in tc.tc_pred.tp_succs;
all tc:TC | tc in tc.tc_succ.tp_preds;
all tc:TP | some tp.tp_preds ⇒ tp.tp_preds.tc_succ = tp;
all tc:TP | some tp.tp_succs ⇒ tp.tp_succs.tc_pred = tp;
no ∧(tc_pred.tp_preds) & iden;
no ∧(tc_succ.tp_succs) & iden;
/* last two lines check for cycles                 */
```

Algorithm 4: Main TCN invariants expressed in the Alloy Specification Language

```
/* declaration of the Time Phase signature         */
sig Tph{events: set TP, next: one Tph, tcn: one TCN};
/* declaration of the TCN signature                */
sig TCN{epoch : TP, tps: set TP, tcs: set TC, init: one Tph};
```

Algorithm 5: Definition of the notions of Time Phase and Temporal Constraint Network (with time phases)

forall *p:Tph* **do**
  p.events.tp_succs.tc_succ *in* p.∧next.events;
  p.events.tp_preds.tc_pred *in* p.∧∼next.events;
  p *in* p.tcn.init.*next;
  p.events *in* p.tcn.tps;
  no p.events & p.∧(next).events;
end

Algorithm 6: Main Time Phase invariants expressed in the Alloy Specification Language

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[24] for model transformation 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.

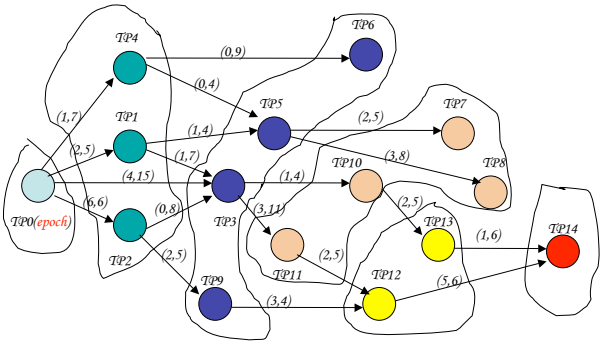

Fig. 2 A Parallel TCN Topology with 15 Time Points and 6 Time Phases

To achieve higher safety and better performance, our parallel propagation scheme employs a number of innovative 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 wide-spread 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[15]. 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 requires 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. Moreover, the interdependency of processes implied by the use of locks diminishes the parallelism of a concurrent system. A lock-free object guarantees that within a set of contending processes, there is at least one process that will make progress within a finite number of steps. We have employed the implementation of the lock-free vector described in Section 5 in order to meet our goals for thread-safe and effective non-blocking 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, impact 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.

5 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 [15]. Often due to the resource limitations of flight-qualified hardware, optimized lock mechanisms are not a desirable alternative [20]. 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 reliability, avoid the dangers of priority inversion, deadlock, and livelock, and at the same time gain performance, we rely on the notion of *lock-free synchronization*. 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 nonblocking at the expense of some extra work performed by the contending processes. Linearizability 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 [17]. 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). Our vector's nonblocking algorithms are directly derived from the lock-free operations of the first implementation of a lock-free dynamically resizable array presented by Dechev at el. in [5]. The operations of our 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 [5]).

5.1 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 [7], 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 [11]. 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 update upon the execution of some of the vector's basic operations.

Table 1 Vector - 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*

5.2 Overview of the Lock-free Operations

In this section we present a brief overview of the most critical vector's lock-free algorithms (see [5] for the full set of the nonblocking algorithms). To help tail operations update the size and the tail of the vector (in a linearizable manner), the design presented in [5] 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 [5]. 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.

repeat
  $desc_{current} \leftarrow vector.desc$;
  $CompleteWrite(vector, desc_{current}.pending)$;
  if $vector.memory[bucket] == NULL$ **then**
    $AllocBucket(vector, bucket)$;
  end
  $writeop \leftarrow new\ WriteDesc(At(desc_{current}.size), elem, desc_{current}.size)$;
  $desc_{next} \leftarrow new\ Descriptor(desc_{current}.size + 1, writeop)$;
until $CAS(\&vector.desc, desc_{current}, desc_{next})$;
$CompleteWrite(vector, desc_{next}.pending)$;

Algorithm 7: push_back $vector, elem$

return $At(vector, i)$;

Algorithm 8: Read $vector, i$

$At(vector, i)\hat{} \leftarrow elem$;

Algorithm 9: Write $vector, i, elem$

repeat
  $desc_{current} \leftarrow vector.desc$;
  $CompleteWrite(vector, desc_{current}.pending)$;
  $elem \leftarrow At(vector, desc_{current}.size - 1)$;
  $desc_{next} \leftarrow new\ Descriptor(desc_{current}.size - 1, NULL)$;
until $CAS(\&vector.desc, desc_{current}, desc_{next})$;
return $elem$;

Algorithm 10: pop_back $vector$

if $writeop.pending$ **then**
  $CAS(At(vector, writeop.pos), writeop.value_{old}, writeop.value_{new})$;
  $writeop.pending \leftarrow false$;
end

Algorithm 11: CompleteWrite $vector, writeop$

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.

6 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[19] 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. 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[1] 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[24] 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 10.5.1 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[19]. In the experiments (Figure 3), we have generated a number of TCN graph topologies (each consisting of 4 to 8 Time Phases), in a manner similar to the pseudo-random graph generation methodology

described in [8]. In the presented results on Figure 3 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 of-

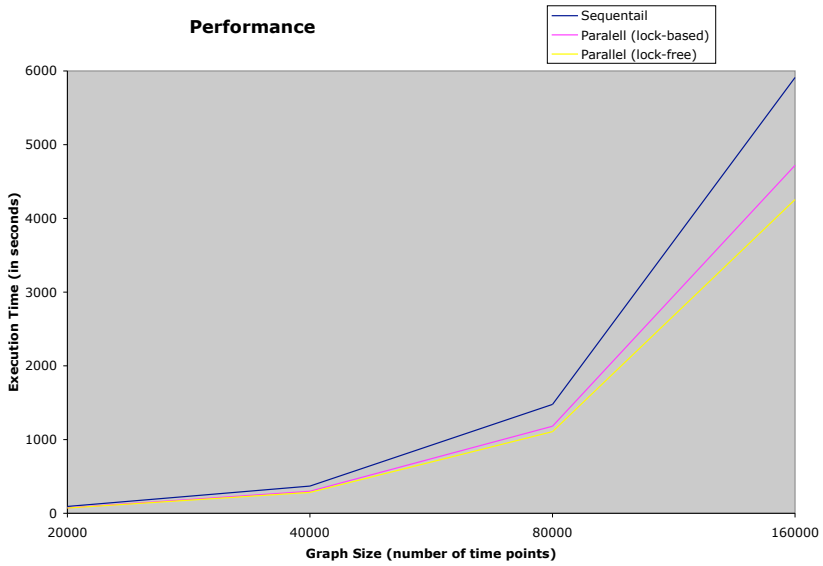

Fig. 3 Performance Analysis. x-axis represents the number of TPs in each experimental TCN topology, y-axis represents the execution time in seconds of each of the three propagation algorithms

fers 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[5]. In a scenario like our parallel TCN propagation scheme with medium or low contention 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. Based on the experimental results, we expect that the integration of our parallel propagation algorithm in the accelerated testing framework (consisting of several dozen processing units) will deliver significant benefits in reaching cost-effective and reliable flight software certification of control modules based on massive real-world goal networks.

7 Conclusion

The notions of time and concurrency are of critical importance for the design and development of autonomous space systems. The current certification methodologies do not reach the level of detail of providing guidelines for the development and validation of concurrent and real-time software. The increasing number of complex interactions and tight coupling of the future autonomous space systems pose significant challenges for their development and man-rated certification. A number of platforms for accelerated testing suggest a paradigm shift by applying a combination of modeling and verification methods, code generation tools, and software parallelization for establishing a cost-effective and reliable certification process. In the light of the challenges posed by the design and development of these highly experimental approaches, we presented in this work 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 innovative lock-free synchronization techniques to achieve better performance and higher safety of our parallel implementation. Our preliminary tests indicate that our parallel propagation approach, upon integration in the accelerated testing framework, can support cost-effective and reliable flight software certification of control modules based on massive real-world goal networks. In our future work we plan to focus on developing a component for automatic derivation of the model specification directly from implementation source code. This can be accomplished by utilizing the high-level internal program representation and the analysis tools provided by The Pivot[28], a framework for static analysis and transformations in C++.

References

1. Boehm, B. and Bhuta, J. and Garlan, D. and Gradman, E. and Huang, L. and Lam, A. and Madachy, R. and Medvidovic, N. and Meyer, K. and Meyers, S. and Perez, G. and Reinholtz, K. and Roshandel, R. and Rouquette, N.: Using Empirical Testbeds to Accelerate Technology Maturity and Transition: The SCRover Experience. ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering (2004)
2. Brat, G. and Drusinsky, D. and Giannakopoulou, D. and Goldberg, A. and Havelund, K. and Lowry, M. and Pasareanu, C. and Venet, A. and Washington, R. and Visser, W.: Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. Formal Methods in Systems Design Journal, September (2005)
3. Columbia Accident Investigation Board: Columbia Accident Investigation Board Report Volume 1, http://caib.nasa.gov/
4. Cormen, T. and Leiserson, C. and Rivest, R. and Stein, C.: Introduction to algorithms. ISBN 0-262-03293-7, MIT Press (2001)
5. Dechev, D. and Pirkelbauer, P. and Stroustrup, B.: Lock-Free Dynamically Resizable Arrays. OPODIS 2006, Lecture Notes in Computer Science, Volume 4305 (2006)
6. Denney, E. and Fischer, B.: Software Certification and Software Certification Management Systems. SoftCement05: In Proceedings of the 2005 ASE Workshop on Software Certificate Management (2005)
7. Detlefs, D. and Flood, C. and Garthwaite, A. and Martin, P. and Shavit, N. and Steele, G.: Even Better DCAS-Based Concurrent Deques. International Symposium on Distributed Computing (2000)
8. Dick, R. and Rhodes, D. and Wolf, W.: TGFF: task graphs for free. CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign (1998)
9. Dvorak, D.: Challenging encapsulation in the design of high-risk control systems. Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA (2002)
10. Dvorak, D. and Bollella, G. and Canham, T. and Carson, V. and Champlin, V. and Giovannoni, B. and Indictor, M. and Meyer, K. and Murray, A. and Reiinholtz, K.: Project Golden Gate: Towards Real-Time Java in Space Missions. In the Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC (2004)
11. Fraser, K.: Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory (2004)
12. Gluck, R. and Holzmann, G.: Using SPIN Model Checker for Flight Software Verification. In Proceedings of the 2002 IEEE Aerospace Conference (2002)
13. Harris, T. and Fraser, K. and Pratt, I.: A practical multi-word compare-and-swap operation. Proceedings of the 16th International Symposium on Distributed Computing (2002)
14. Herlihy, M.: A methodology for implementing highly concurrent data structures. PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming (1990)
15. Intel: Reference for Intel Threading Building Blocks, Version 1.0 (2006)
16. Jackson, D.: Software Abstractions: Logic, Language and Analysis. The MIT Press (2006)
17. Lamport, L.: How to make a multiprocessor computer that correctly executes programs. IEEE Trans. Comput. (1979)
18. Lee, E. and Neuendorffer, S.: Concurrent Models of computation for Embedded Software. IEEE Proceedings on Computers and Digital Techniques (2005)
19. Lou, J.: An Efficient Algorithm for Propagation of Temporal Constraint Networks. NASA Tech Brief Vol. 26 No. 4 from JPL New Technology Report NPO-21098 (2002)
20. Lowry, M.: Software Construction and Analysis Tools for Future Space Missions. TACAS 2002: Lecture Notes in Computer Science, Volume 2280 (2002)
21. Perrow, C.: Normal Accidents. Princeton University Press (1999)
22. Rasmussen, R. and Ingham, M. and Dvorak, D.: Achieving Control and Interoperability Through Unified Model-Based Engineering and Software Engineering. AIAA Infotech at Aerospace Conference (2005)

23. Dos Reis, G. and Stroustrup, B.: Specifying C++ Concepts. ISO WG21 N1886 (2005)
24. Rouquette, N.: Analyzing and verifying UML models with OCL and Alloy. EclipseCon (2008)
25. RTCA: Software Considerations in Airborne Systems and Equipment Certification DO-178B (1992)
26. Schumann, J. and Visser, W.: Autonomy Software: V&V Challenges and Characteristics. In Proceedings of the 2006 IEEE Aerospace Conference (2006)
27. Stroustrup, B.: The C++ Programming Language. Addison-Wesley Longman Publishing (2000)
28. Stroustrup, B. and Dos Reis, G.: Supporting SELL for High-Performance Computing. In Proceedings of the International Workshop on Languages and Compilers for Parallel Computing LCPC (2005).
29. Volpe, R. and Nesnas, I. and Estlin, T. and Mutz, D. and Petras, R. and Das, H.: The CLARATy Architecture for Robotic Autonomy. IEEE Aerospace Conference (2001)