Model-based Product-Oriented Certification

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

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

Abstract

Future space missions such as the Mars Science Laboratory and Project Constellation suggest the engineering of some of the most complex man-rated software systems. The present process-oriented certification methodologies employed by NASA are becoming prohibitively expensive when applied to systems of such complexity. The process of software certification establishes the level of confidence in a software system in the context of its functional and safety requirements. Providing such certification evidence may require the application of a number of software development, analysis, and validation techniques. We define product-oriented certification as the process of measuring the system's reliability and efficiency based on the analysis of its design (expressed in models) and implementation (expressed in source code). In this work we introduce a framework for model-based product-oriented certification founded on the concept of source code enhancement and analysis. We describe a classification of the certification artifact types, the development and validation tools and techniques, the application domain-specific factors, and the levels of abstraction. We demonstrate the application of our certification platform by analyzing the process of model-based development of the parallel autonomic goals network, a critical component of the Jet Propulsion Laboratory's Mission Data System (MDS). We describe how we identify and satisfy seven critical certification artifacts in the process of model-driven development and validation of the MDS goal network. In the analysis of this process, we establish the relationship among the seven certification artifacts, the applied development and validation techniques and tools, and the level of abstraction of system design and development.

1 Introduction

[1] In this work we introduce a framework for model-based product-oriented certification founded on the concept of source code enhancement and analysis by the utilization of advanced programming techniques and tools. As opposed to process-oriented certification (as suggested by DO-178B [24]), the product-oriented methodology[7] relies on the application of safety concerns directly on implementation source code and its formal models. The product-oriented approach is inherently more flexible by offering the developers the freedom to follow a variety of software development life-cycle paradigms. In addition, the certification authority itself has the ability to collect all required artifacts for the system's safety and quality assurance. As suggested in [29], the rationale for source code enhancement is to seek an effective alternative to domain-specific programming languages for high-performance computing systems. A language enhancement can be achieved by supersetting a programming language by a library defining domain-specific concepts and algorithms and at the same time employing program analysis and validation tools to ensure the correctness of the introduced domain-specific notions. Source code enhancement allows a programmer to reach a high level of expressiveness and at the same time rely on the tool-chain of a mainstream programming language. We demonstrate the use of our certification platform by describing its application to the process of model-based development and verification of the autonomic goals network [6], a critical component of the Jet Propulsion Laboratory's Mission Data System (MDS). MDS provides an experimental goal- and state- based platform for model-driven testing and development of autonomous real-time flight applications [21]. 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 [7]. Providing such certification evidence may require the application of a number of software development, analysis, verification, and validation techniques [18]. Our approach offers a classification of and establishes the relationship among the certification artifact types, the development and validation tools and techniques, the application domain-specific factors, and the levels of abstraction. We further suggest the concept of semantic enhancement of the source code and the application of a number of program analysis and transformation techniques to achieve reliabil-

[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 Proceedings of 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (IEEE ECBS 2009), San Francisco, California, April 2009.

ity and efficiency in the system implementation. Our case study in Section 4 illustrates the practical application of our approach. We describe how we identify and satisfy seven critical certification artifacts in the process of model-driven development and validation of the MDS goal network. In the analysis of this process, we establish the relationship among the seven certification artifacts, the applied development and validation techniques and tools, and the level of abstraction of system design and development.

2 Background and Previous Work

Future space exploration projects such as the Mars Science Laboratory (MSL) [31] and Project Constellation [27] suggest the engineering of some of the most complex manrated 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 [25] suggests that the current process-oriented certification methodologies are prohibitively expensive for systems of such complexity. A detailed analysis by Lowry [18] 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) [21], as well as a number of program synthesis, modeling, analysis, and verification platforms, such as The JavaPathFinder [2], the CLARAty project [30], Project Golden Gate [10], The New Millenium Architecture Prototype (NewMAAP) [9].

2.1 Complex Systems Analysis

In [20] 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 [20] 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.

2.2 Parallelism and Complexity

In a parallel application, there are a number of challenges that are not known in sequential programming: most importantly to correctly manipulate data where multiple threads access it. The most commonly applied technique for controlling the interactions of the concurrent processes is the application of mutual exclusion locks. A mutual exclusion lock guarantees thread-safety of a concurrent object by blocking all contending threads except the one holding the lock. This can seriously affect the performance of the system and diminish its parallelism. For the majority of applications, the problem with locks is one of difficulty of providing correctness more than one of performance. The application of 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 [14] or context-switching. Often due to the resource limitations of flight-qualified hardware, optimized lock mechanisms are not a desirable alternative [18]. 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 cause anomalous behavior.

As discussed by Lowry [18], 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. The investigation furthermore revealed that it would have been impossible to detect the problem with the black box testing applied at the time to derive the certification artifacts. A 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 efficient and reliable control of the subtle interactions in the concurrent applications of the modern aerospace autonomous systems is of critical importance to the overall system's safety and correct operation. Despite the challenges in debugging and verification of the system's concurrent components, the existing certification process [24] 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. According to [21] some of the most significant challenges for such systems are managing a large number of tightly-coupled components, performing operations in uncertain remote environments, ability to respond and recover from anomalies, guaranteeing the system's correctness and reliability, and the effective communication across the system's components.

3 Principles of Model-based Product-Oriented Certification

We define *product-oriented certification* as the process of establishing the system's reliability and efficiency based on the analysis of its design (expressed in models) and implementation (expressed in source code). In this section we describe a classification of the *certification artifact types*, the *development and validation tools and techniques*, the *application domain-specific factors*, and the *levels of abstraction*. In our framework a *certification artifact type* can be one of the following:

(1) Invariant (η): a critical property or assumption that is constant (does not change throughout the transformation of program/system states) and must hold at all time to ensure the validity and correct operation of a program or a system.*Example a.*: the values stored in a given shared vector must be word-sized pointers. *Example b.*: a graph of temporal constraints [6] must contain no cycles.

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

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

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

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

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

3.1 Development

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

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

(2) Programming Language (Δ_{PL}): programming constructs, libraries, and techniques available. *Example:* The availability of a nonblocking vector [5] that can allow safe and lock-free access to shared data (and thus eliminate the hazards of deadlock, livelock, and priority inversion).

(3) Modeling Tools (Δ_{MT}): expressing design notions, automated code generation, and formal verification. *Example:* the application of the SPIN model checker [13] to exhaustively search all concurrent interleavings.

(4) Analysis Techniques (Δ_{AT}): program static and dynamic analysis. *Example:* the application of static analysis utilizing a high-level program representation to guarantee high performance in parallel systems [29].

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

3.2 Application Domains

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

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

(2) Safety-critical (Sc): establishes that a failure would lead to a catastrophic or hazardous consequences to the entire system. The DO-178B offers a hazards analysis process to assess the risk level upon a module or sub-system failure. *Example:* if the autonomous obstacle avoidance scheme fails, the rover

might crash. Thus, the system *invariants* and *guarantees* assuring the correct operation of the system are *safety-critical*.

(3) Embedded (Em): since the system is designed and optimized according to a set of pre-defined goals, its software must often control the hardware, consider strict resource constraints, and handle failures and events that may occur in the physical world. *Example:* the embedded nature and limited memory availability of the rover places the *constraint* that a goal network should not exceed 7168KB of memory space.

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

3.3 Levels of Abstraction

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

(1) Physical and Hardware (Φ): related to constraints in the hardware resources, organization, and architecture and the conditions in the physical environment. *Example:* the lack of complex atomic primitives on the flight-qualified hardware requires all nonblocking code to rely on the single-word Compare-And-Swap (CAS) atomic primitive. This demands the specification of an *invariant* that the system must eliminate the possibility of occurrence of the ABA problem [5].

(2) Algorithms and Procedures (Θ): invariants of a particular computational routine or algorithm. *Example:* the complexity of Floyd-Warshall's all-pairs-shortest-path algorithm [4] is $O(N^3)$. Due to the frequent execution of the constraint propagation scheme in a goal network the direct application of the algorithm can be prohibitively expensive. To meet the *performance* requirements a propagation scheme should execute with complexity of at most $O(N^2)$.

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

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

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

(6) Framework (Ξ): conditions related to the principle organization and design of the software development. *Example:* The Mission Data System defines the notions of states and goals. Their definition and requirements are described (independently from the implementation of a particular mission) in a number of MDS framework papers such as [21].

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

Modeling and formal verification tools such as SPIN [13], Alloy Analyzer [15], and Eclipse [26] are used to express and validate high-level domain-specific and design concerns. The challenges associated with the application of modeling and formal verification tools in the development process are:

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

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

(b) from models to implementation: the application of program synthesis techniques such as AutoFilter [8] 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. However, 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.

The semantic enhancement of the implementation can allow for the direct validation of some software invariants and guarantees and thus reduce the state space and the computational complexity required in the process of formal verification. In addition, the increased expressiveness and abstraction level of the implementation source can ease the manual or automated transition to and from the software models. Stroustrup and Dos Reis [29] present the notion of *Semantically Enhanced Library Languages(SELLs)*. As defined by the authors, a SELL is a domain-specific language derived from a general-purpose programming language by extending it with libraries defining the concepts and functionalities of the problem domain and then applying an analysis tool to guarantee the higher-level semantic invariants. The main advantages of defining and applying a SELL are founded in the availability of the maintenance, training, and tool chain of the general-purpose language that had served as its base. At the same time, a SELL's main purpose is to deliver a special-purpose language tailored to the ideals and concepts of a specific application domain. The notion of SELL is fundamental for the application of our model-based product-oriented framework for software certification.

The following section presents a case study describing the details of how we extend the semantics of ISO C++ with the libraries defining Temporal Constraint Networks and Shared Containers with Nonblocking Synchronization. Furthermore, we demonstrate how the applied programming and modeling techniques, formal verification, program transformation, and static analysis (in the process of validation and automatic parallelization of goal networks) relate to our classification framework.

4 A Case Study: Automatic Parallelization of Goal Networks in MDS

Mission Data System (MDS) [21] is the Jet Propulsion Laboratory's framework for designing and implementing complete end-to-end data and control autonomous flight systems. The framework focuses on the representation of three main software architecture principles (defining the highest Δ_{SA} level of development in the certification framework):

(1) System control: a state-based control architecture with explicit representation of controllable state [11]

(2) Goal-oriented operation: control intent is expressed by defining a set of goal and a goal network [1]

(3) Layered data management: an integrated data management and transport protocols [32]

In MDS a state variable provides access to the data abstractions representing the physical entities under control over a continuous period of time, spanning from the distant past to the distant future. In other words, a state variable is a programming (Δ_{PL}) and modeling (Δ_{MT}) representation of a set of constraint (κ) certification variables ($S_{\kappa_{sv}}$) expressed in the library level of abstraction (Λ). As explained by Wagner [32], the implementation's intent is to define a goal timeline overlapping or coinciding with the state variables timeline. This means that the implementation must rely on an algorithm (abstraction level Θ) that transforms the engineers intent together with $S_{\kappa_{sv}}$ into a set of invariants $S_{\eta_{sv}}$ and a set of guarantees $S_{\gamma_{sv}}$ and establish the validity and consistency of all $\eta_i \in S_{\eta_{sv}}$ and all $\gamma_i \in S_{\gamma_{sv}}$ so that the system's operations corresponds with its Rt, Sc, Em, and Au behavior.

Computing the invariants (a set of $S_{\eta_{gi}}$) necessary for achieving a goal (any $\gamma_i \in S_{\gamma_{sv}}$) might require the lookup of past states as well as the computation of projected future states. MDS employs the concept of goals to represent control intent. Goals are expressed as a set of temporal constraints (Section 4.1). Each state variable is associated with exactly one state estimator whose function is to collect all available data and compute a projection of the state value and its expected transitions. Control goals are considered to be those that are meant to control external physical states. Knowledge goals are those goals that represent the constraints on the software system regarding a property of a state variable. Not all states are known at all time. The most trivial knowledge goal is the request for a state to be known, thus enabling its estimator. A data state is defined as the information regarding the available state and goal data and its storage format and location. The MDS framework considers data states an integral part of the control system rather than a part of the system under control. There are dedicated state variables representing the data states. In addition, data states can be controlled through the definition of data goals. A data state might store information such as location, formatting, compression, and transport intent and status of the data. A data state might not be necessary for every state variable. In a simple control system where no telemetry is necessary, the state variable implementation might as well store the information regarding the variable's value history and its extrapolated states.

The representation of the data states and the data management in MDS is implemented in the Data Management Service module [32]. The problem of data management in an embedded control system (often requiring the satisfaction of Em, Rt, Au, and Sc -driven certification requirements at the same time) is one of translating the intent of remote operations into actions and then safely returning the observed information. The system should be robust to the extent of overcoming possible communication loss, hardware failures or a system reboot. The resource constraint of an embedded control systems (its collection of κ variables) dictate that command-oriented control systems typically do not retain information specific to the intent of the observations. In addition, telemetry systems process and transport data in unlabeled packages where the scientific data is often mixed with other data. In this context, Wagner argues that it is of significant importance to address the challenges of providing uniform models for managing the flow of observation and control data. The MDS Data Management Service Library implements a Catalog for organizing the storage of physical observations in terms of storage products. Its functionality is responsible for the remote transport of data products with respect to the behavior of other spacecraft components. According to the current lock-based Catalog design, locks are applied in a complex manner within the inheritance hierarchy that leads to an exponential increase of the verification state space.

To achieve higher reliability (expressed as a set of safety invariants $S_{\eta_{safety}}$) and enhance the performance (measured in terms of speed of execution in the ϵ_{exe} variable), we consider the application of *lock-free synchronization.* As defined by Herlihy [12], 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. This means that a nonblocking technique represents an algorithmic (Θ) or library (Λ) solution to an important Sc problem, namely avoidance of the hazards of deadlock, livelock, and priority inversion (expressed as three separate Sc invariants: η_{lvlock}, η_{delock}, and η_{pinv}) while at the same time offering a significant performance boost (measured in ϵ_{exe}). The application of a library of nonblocking algorithms shifts the complexity of engineering shared data access from the user's source into the lock-free library implementation. Thus lock-free programming techniques can often help increase the comprehensibility of the concurrent interactions in the user's implementation. In the process of creating a parallel network of temporal constraints (by utilizing nonblocking synchronization), we measure the increased simplicity of the code (in contrast to the application of mutual exclusion) with the certification variable σ_{ptcn}.

Lock-free systems typically utilize CAS in order to implement 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 (expressed as the nonblocking invariant η_{nb}). The system is non-blocking at the expense of some extra work performed by the contending processes. Linearizability [12] 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 (expressed as the linearizability invariant η_{lin}). The consistency model implied by the linearizability requirements is stronger than the widely applied Lamport's sequential consistency model [16]. 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).

Practical nonblocking programming techniques stand at a development level Δ_{PL} and when used properly help in assuring safe and efficient access to shared data (in a concurrent system defined by the system's Δ_{MC}) and their semantics and implementation is directly related to the atomic primitives available by the system's Φ level (such as the availability of atomic primitives like Compare-And-Swap (CAS) or Double-Compare-And-Swap (DCAS)). In hardware platforms that do not provide complex atomic primitives (involving the atomic update of more than a single-word location), the implementation of lock-free algorithms is CAS-based. Such systems impose yet another important invariant η_{aba} where the programmer must eliminate the possibility of occurrence of the ABA problem [5].

The ABA problem [19] 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, and 2. the memory allocator reuses the address of an already freed object. In these scenarios a CAS speculation can succeed despite the fact that the speculating process has experienced an interrupt and the value about to be updated had been modified by other processes. The completion of such a CAS speculation can lead to ABA and can seriously affect the semantics of the nonblocking algorithm. One approach to eliminating ABA is to strictly define the semantic usage pattern of a nonblocking algorithm (meaning that not all operations might be total at all time). Such usage rules are another example of a transformation of an invariant (η_{aba}) into a set of guarantees ($S_{\gamma_{aba}}$) variables that need to be satisfied. One possibility is the application of static analysis [29] that can check for the exclusion of interleavings leading to ABA hazards. In such a scenario the ABA problem (η_{aba}) is resolved by the application of ΔAT development tools.

In our case study we describe the integration of a nonblocking vector in a 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.

4.1 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) [21] state-based and goal-oriented unified architecture for testing and development of mission software. A TCN consists of a set of temporal constraints (TCs) and a set of time points (TPs). In this model of goal-driven operation, a time point is defined as an interval of time when the configuration of the system is expected to satisfy a property predicate. The width of the interval corresponds to the temporal uncertainty inherent in the satisfaction of the predicate. Similarly, temporal constraints have an associated interval of time corresponding to the acceptable bounds on the interactions between the control system and the system under control during the performance of a specific activity. A TCN graph topology represents a snapshot at a given time of the known set of activities the control system has performed so far, is currently engaged in, and will be performing in the near future up to the horizon of the 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 ($S_{\eta_{tcn}}$).

(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 ($S_{\eta_{tcn}}$) 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.

4.2 TCN Verification and Automatic Parallelization

In this section we describe the design and application of a model-based framework for verification and parallelization of real-time C++ within JPL's MDS Framework (Figure 1). The main goal of the framework is to provide a model-driven design of a parallel temporal constraint propagation scheme where the following seven critical certification variables are satisfied: $S_{\eta_{safety}}$, ϵ_{exe}, η_{nb}, η_{lin}, η_{aba}, $S_{\eta_{tcn}}$, and σ_{ptcn}. 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 [17]. 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 the presented tool-chain 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 4.1) and *automatically* derive an implementation that can be executed concurrently. Our approach for achieving concurrent execution is based on the idea of identifying Time Phases within a goal network, which allow the parallelization of the constraint propagation algorithm. A fundamental component of our model-driven design is the construction and execution of a formal verification model in Alloy[15] that represents the implementation's core semantics and functionality. We refine a formal modeling and analysis methodology [23] that helps us analyze the logical properties of the goal network model and automatically derive a meta-model for our parallel solution.

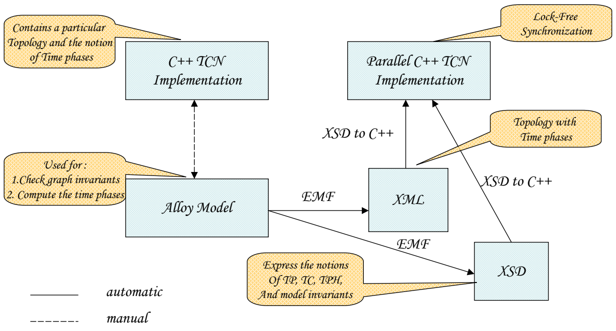
Contains a particular Topology and the notion of Time phases
C++ TCN Implementation
Lock-Free Synchronization
Parallel C++ TCN Implementation
XSD to C++
Topology with Time phases
Used for :
1.Check graph invariants
2. Compute the time phases
Alloy Model
EMF
XML
XSD to C++
EMF
XSD
Express the notions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

Figure 1. TCN Verification and Parallelization

4.3 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 [17], 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 considering only its incoming temporal constraints. Similarly, a backward pass recomputes the time windows at each time point by considering only its outgoing temporal constraints. 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.

4.4 Model-based Development and Certification

Alloy [15] is a lightweight formal specification and verification tool for the automated analysis of user-specified invariants on complete or partial models. We use the Alloy specification language [15] to formally represent and check the TCN main invariants ($S_{\eta_{tcn}}$). In our C++ goal networks implementation we have applied generic programming techniques and concepts [22], 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 addition, we utilize the Alloy Analyzer to implement our parallelization approach. Our method for 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. Having identified the time phases in our temporal constraint network specification in Alloy, the aim of the rest of our toolchain is to *automatically* derive the C++ implementation of the parallel solution through a number of code transformation techniques. Following Rouquette's methodology [23] 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.

4.5 Analysis

Table 1 provides a summary of the applied development tools that help us satisfy the seven critical certification variables in the process of TCN verification and parallelization. Each non-empty cell indicates the level of abstraction of the applied development tool. Empty cells are designated by the $\emptyset$ symbol. Below we briefly explain each entry in the table:

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

Table 1. Linking Certification Artifacts, Development Tools, and Levels of Abstraction

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

(2) ϵ_{exe}: as described in detail in [5], when used under contention a nonblocking shared vector can deliver a significant performance boost (by a factor of 10 or more) when compared with the application of the most recent and optimized mutual exclusion schemes. In the scenarios when the shared data structure access patterns show less contention, the nonblocking techniques provide a scalable and efficient solution with performance better or equal to the most optimal mutual exclusive schemes [6]. Achieving better performance and scalability of our parallel goal network is also based on the application of programming techniques at the algorithms/library level.

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

(4) η_{lin}: some operations in a shared vector require the update (in a linearizable fashion) of two or more memory locations. Such operations are $push\_back$ (need to update the tail and the size of the vector) and $resize$ (need to update the size and copy all elements). Implementing such operations in a linearizable fashion with the support of only single-word atomic primitives is notoriously difficult. We have employed a set of practical lock-free programming techniques to guarantee that the vector's operations are linearizable (such a technique is the use of Barne's-style announcement [5]).

(5) η_{aba}: The ABA problem is fundamental to all CAS-based systems and can affect the semantics of the nonblocking algorithms. In systems allowing complex atomic primitives such as CAS2 or DCAS, ABA can be easily avoided by attaching a version counter to each value. In such a case we would have had an algorithmic solution with a strong support from the hardware architecture. We cannot assume the availability of such complex atomic primitives in the hardware architecture of the flight-qualified embedded hardware. Our solution to the ABA problem is the application of a library for program analysis [29] that can help us eliminate hazardous interleavings.

(6) $S_{\eta_{tcn}}$: to guarantee the correct operation of our autonomous goal-driven application, we have build a framework (Figure 1) that relies on modeling, program analysis, and program transformation programming techniques.

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

5 Conclusion

We introduced an innovative framework for model-based product-oriented certification founded on the concept of source code enhancement and analysis. We offered a classification of the certification artifact types, the development and validation tools and techniques, the application domain-specific factors, and the levels of abstraction used in our certification platform. We used our certification platform to analyze the model-driven development of a parallel propagation scheme of the MDS temporal constraint network module. In our analysis we identified seven critical certification artifact: 1. providing the safety of the concurrent interactions (by eliminating the hazards of deadlock, livelock, and priority inversion), 2. achieving better scalability and overall system performance, 3. allowing nonblocking synchronization, 4. having linearizable operations on the shared data, 5. eliminating the possibility of ABA corrupting the concurrent operations' semantics, 6. establishing the correctness of the core TCN graph invariants, and 7. having simpler to analyze and maintain parallel processes. In our discussion we explained the relationships among these seven certification artifacts and the underlying hardware architecture, the applied programming techniques, and program analysis, modeling, and transformation techniques. Our certification framework helped us formulate, express, and analyze the process of product-oriented certification for a complex computer-based system, such as the model-driven development tool-chain of parallel autonomous goal networks.

References

[1] A. Barett, R. Knight, J. Morris, and R. Rasmussen. Mission Planning and Execution Within the Mission Data System. In *Proceedings of the International Workshop on Planning and Scheduling for Space*, 2004.

[2] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, R. Washington, and W. Visser. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. In *Formal Methods in Systems Design Journal*, September 2005.

[3] Columbia Accident Investigation Board. Columbia Accident Investigation Board Report Volume 1.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[5] 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.

[6] 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.

[7] E. Denney and B. Fischer. Software Certification and Software Certification Management Systems. In *SoftCement05. Proceedings of the 2005 ASE Workshop on Software Certificate Management*, 2005.

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

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

[10] D. Dvorak, G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reiinholtz. Project Golden Gate: Towards Real-Time Java in Space Missions. In *IEEE ISORC*, 2004.

[11] D. Dvorak, R. Rasmussen, and T. Starbird. State Knowledge Representation in the Mission Data System. In *Proceedings of IEEE Aerospace Conference*, 2002.

[12] 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.

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

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

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

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

[17] J. Lou. An Efficient Algorithm for Propagation of Temporal Constraint Networks. *NASA Tech Brief Vol. 26 No. 4 from JPL New Technology Report NPO-21098*, April 2002.

[18] 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.

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

[20] C. Perrow. *Normal Accidents*. Princeton University Press, September 1999.

[21] 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.

[22] G. D. Reis and B. Stroustrup. Specifying C++ Concepts, ISO WG21 N1886, 2005.

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

[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 *Proceedings of the 2006 IEEE Aerospace Conference*, 2006.

[26] M. Sherriff and L. Williams. DevCOP: A Software Certificate Management System for Eclipse. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 375–384, Washington, DC, USA, 2006. IEEE Computer Society.

[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 *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 *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing, LCPC 2005*, 2005.

[30] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty Architecture for Robotic Autonomy. In *IEEE Aerospace Conference*, March 2001.

[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 *Proceedings of the IEEE System, Man, and Cybernetics Conference*, 2005.