Research Proposal

NASA Intelligent Systems NRA Program
TA-1 Automated Reasoning
NRA2-37143

Formal Verification Tools and Techniques for Autonomous Systems

Submitted by

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Principal Investigator:	Jeannette Wing	(412) 268-3068	wing@cs.cmu.edu
Co-Investigators:	Edmund Clarke	(412) 268-2628	emc@cs.cmu.edu
	David Garlan	(412) 268-5056	garlan@cs.cmu.edu
	Bruce Krogh	(412) 268-2472	krogh@ece.cmu.edu
	Reid Simmons	(412) 268-2621	reids@cs.cmu.edu
Administrative Contact:	Karen Faber	(412) 268-5838 Fax (412) 268-5841	faber@andrew.cmu.edu

BUDGET PROPOSAL		**Year One**	**Year Two**	**Year Three**		**TOTAL**
		01/01/01 to 12/31/01	**01/01/02 to 12/31/02**	**01/01/03 to 12/31/03**		**01/01/01 to 12/31/03**
TOTAL COST		**$ 298,850**	**$ 577,457**	**$ 602,974**		**$ 1,479,281**

Jeannette Wing Date
Professor, Computer Science

Randal Bryant Date
Department Head, Computer Science

James Morris Date
Dean, School of Computer Science

Susan Burkett Date
Associate Provost

October 2000

Table of Contents

1 Abstract

Reliable, safety-critical software is crucial to the success of most NASA missions. As both space and ground software systems become more complex, and increasingly more autonomous, guarantees of software correctness becomes crucial. While traditional approaches to validation (simulation and testing) are useful, they cannot, in general, be used to check all possible scenarios. Formal verification techniques, such as model checking, can complement such validation approaches by providing guarantees that certain system properties hold (such as absence of deadlock or resource contention).

While completely verifying large software systems is infeasible, guaranteeing certain critical components of a system can yield huge payoffs. We propose to develop software tools and novel verification techniques to enable engineers to utilize formal methods, with little or no training. The tools can be used routinely to provide semantic analyses (and explanations) of complex, time-dependent software, just as compilers now provide static syntactic analyses. By enabling engineers to verify parts of their systems early in the development process, design flaws can be caught early, and validation efforts can be focused on those areas that formal verification cannot handle.

We will focus on verification of *task executives*, an important component of autonomous systems that is responsible for executing plans, managing resources, monitoring state, and handling exceptions. While most space systems have executives, the executive for autonomous systems is typically much more complex, dealing with conditional execution, flexible plans, and complex resource constraints. We will focus on verification of three significant types of constraints needed for the successful operation of task executives - synchronization constraints (temporal relationships between tasks), resource constraints, and environmental constraints (how the system should interact with the world).

Our approach is two pronged:

1. We will create tools, usable by space engineers, that exploit existing model-checking techniques to verify discrete synchronization and resource constraints. We will develop novel methods for automatically abstracting such constraints from program code, translating to a formal model-checking language, and explaining properties that are found not to hold.
2. We will develop fundamental new model-checking techniques, and extend existing ones, to handle realistic size problems having to do with metric synchronization constraints, continuous resources, and system interaction with the environment. In particular, we will develop formal verification techniques for probabilistic and hybrid models. In these cases, we will use NASA problems to focus the research. As these new techniques mature, we will encapsulate them in tools, as described above.

The investigators have extensive expertise in formal verification of autonomous real-time distributed systems. This proposal addresses fundamental issues in Automated Reasoning related to automated verification of autonomous systems, formal specification languages for describing software, and advanced development environments to facilitate rapid (and *reliable*) development of autonomous software. This work will also have far-reaching impact in industry for the development of robust commercial autonomous and embedded real-time systems. Our focus on task executives also complements ongoing and proposed work at NASA Ames to verify the diagnostic and planning components of autonomous systems. While formal verification is not a panacea, its regular use by developers can go a long way towards helping to create reliable and robust software systems.

2 Objectives

Reliable, safety-critical software is crucial to the success of most NASA missions. Software development increasingly dominates the costs of NASA space systems. Meanwhile, these systems are being required to perform increasingly more complex tasks, and to do so with increasingly higher levels of autonomy. While traditional testing and simulation are useful validation methodologies, in general they are expensive, time consuming, and cannot provide guarantees of system correctness.

In contrast, formal verification tools like model checking can be used to provide such guarantees [13]. To date, however, such techniques have not been widely applied to space applications, due to the difficulty of creating appropriate and tractable models of software systems and of modeling their interactions with the environment. We propose to develop tools and techniques that address the difficulties of modeling and verifying real-time system interactions with uncertain environments. Our goal is to enable developers of autonomous systems to use tools based on formal methods on a routine basis. Just as engineers now rely on compilers to provide syntactic analyses of programs, we envision engineers in the future relying on formal verification tools to provide semantic analyses (and explanations) of complex spacecraft software. The results will complement traditional validation techniques, resulting in much more reliable software, at lower overall development cost.

While completely verifying large software systems is infeasible, experience has shown that verifying certain critical components of a system can yield huge payoffs. Autonomous systems typically consist of capabilities for planning, task execution, and fault diagnosis and recovery. For this work, we will focus on the problem of verifying task executives, which execute plans, monitor state, and handle exceptions. Typical problems encountered by such executives relate to issues of deadlock and safety, with respect to synchronization constraints, potential race conditions arising out of asynchronous events, and potential conflicts in resource utilization. Most importantly, task executives must ensure that the system interacts properly with its environment to achieve the desired goals.

The main thrusts of this proposal are to:

- **Develop verification tools** that can be used routinely by engineers (non-specialists in formal verification): Here, innovative work is needed to create languages and translators that can produce formal models automatically for autonomous system software.
- **Develop verification techniques** that assist in modeling system/environment interactions: Here, fundamental research is needed to develop model-checking algorithms that can represent and verify such interactions effectively and efficiently, especially for uncertain environments.

3 Technical Approach

A critical capability of space systems is task execution - i.e., carrying out planned activities. Examples include orbital and “delta-V” maneuvers, planetary navigation, and automated science experiments on space stations. Task execution is a special concern for autonomous systems because their tasks are typically more complex, and their plans are typically more sophisticated - including

conditional execution, iteration, and dynamic handling of contingencies. Also, activities within plans are often scheduled with high degrees of flexibility, which makes it difficult to know which activities could possibly overlap and potentially interfere with each other.

We propose to focus on three classes of constraints that are critical to the successful execution of task plans:

1. Synchronization constraints. These refer to temporal constraints (both relational and metric) that must hold between activities. For instance, a Deep Space One (DS1) flight rule states "During transponder start-up or reset, the telecom subsystem must be addressed within 10 minutes."
2. Resource constraints. These refer to limitations on resource utilization (power, computation, storage, communication, etc.). For instance, DS1 flight rules state "There must be sufficient time allowed for transferring the MICAS buffer of images to files before refilling the buffer." Similarly, for a rover with a single camera, the camera should never be used for both navigation and science data collection simultaneously.
3. Environmental constraints. These refer to required, or forbidden, interactions of the mechanism with the environment. For instance, a rover should never be driven in such a way that its roll or pitch exceeds thresholds. Satisfying this requirement is complicated by the fact that the roll/pitch sensors may be noisy and often have significant latency.

Of these problems, current model-checking techniques are in principle well suited for verifying task synchronization and resource constraints. To address this class of problems, we will develop novel specification languages that make it easy for engineers to specify properties of interest to be verified, develop tools that can automatically create formal models of the software that can be used verify such properties, and develop techniques for analyzing and explaining counterexamples found by the model checker. In addition, we will extend model-checking algorithms to handle dynamic task creation and investigate efficient algorithms to deal with metric time and resources.

Much more difficult is the problem of verifying a system's interaction with its environment. Here, we will address two inherent characteristics of a space system's environment: continuous dynamics, which we will approximate using discrete models; and uncertainty, which we will accommodate using stochastic models. This part of the work is more fundamental in nature, and will have wide and far-reaching applicability for verification of autonomous and real-time software systems.

3.1 Verifying Synchronization Constraints

Most autonomous systems are constructed as loosely coupled sets of tasks. Synchronization between tasks is needed to ensure proper functioning of the system. Such synchronization constraints may be relative (e.g., "To go from autonomous mode to scripted planning mode requires going to STANDBY first") or more quantitative in nature ("Cat-bed heaters must be on for at least 90 minutes before use").

Software errors that cause a system to violate these constraints often arise from subtle interactions among concurrent tasks (e.g., race conditions, missed events, deadlocks, resource contention, etc.). Since these bugs may manifest themselves only for particular interleavings of actions, the

programmer may not be able to reproduce the problem at will. Thus, it is often very difficult to track down the sources of bugs in task executives using traditional testing and code inspection techniques.

Fortunately, such problems are particularly well suited to automated formal verification using model checking. Model checking starts with a finite state model of the software system under consideration. The model-checking tool then examines all possible interleavings of computations in the model, to expose any violations of task synchronization constraints. To the extent that the model faithfully reflects the actual system, errors found in the model represent problems with the system and *vice versa*.

Unfortunately, there are two kinds of problems with model checking-based verification of complex software systems. The first is problems of usability. To use a model checker, a designer must first create the model. For a complex system this can be a non-trivial task, since it involves manually mapping a (usually) infinite-state software system to a finite-state model. Moreover, the model must be one that is small enough that it can be checked in a reasonable amount of time. Next, the designer must figure out how to characterize the properties of interest in terms that the model checker can check. Finally, the designer must be able to interpret the results - typically a set of counterexamples showing execution paths where a property is violated. In combination these represent significant barriers for system designers and developers to use formal verification on a routine basis.

The second kind of problem is one of applicability: There are classes of properties of complex, real-time systems that are not readily checkable by today's model checking tools. Such properties include behaviors of systems where components are dynamically created or destroyed, and system properties that depend on the use of continuous-time models. In this proposal, we plan to address both kinds of problem, as we detail below.

3.1.1 Improving Usability

We plan to attack problems of usability in four ways. The first is by developing new techniques for automated extraction of task models from ordinary code. Specifically, in this project, we propose to develop new static analysis methods to extract models, called *synchronization skeletons* [11], automatically. A synchronization skeleton is a projection of the code that includes only details relevant to analysis of properties of concurrency.

Static analysis techniques have been an area of active research for several decades. However, the focus of that research has primarily been on program optimization, and only recently on program verification. Although the major successes of model checking have been in hardware, the procedure was originally developed for software [11] and, in particular, analysis of concurrent programs. Original attempts at model checking software, however, found the problem to very hard. The advances of model checking for hardware verification have prompted renewed attempts at achieving the original goal. This project will develop tools that combine static analysis and model checking, using new techniques from both areas.

There has been some previous work in this direction. VeriSoft [21] is a tool that performs a bounded state space search of concurrent C++ programs. *Bebop* is a symbolic model checker, based on the

concept of context-free reachability, that uses an intermediate representation formalism called *Boolean programs* [40]. Klaus Havelund and Willem Visser at NASA Ames implemented JAVA PathFinder [23], a tool that translates JAVA programs to Promela, the input language for the Spin model checker. The Bandera [14] project has produced a collection of tools to translate JAVA into input languages of different verification tools, which includes methods for extracting finite state models from concurrent programs in JAVA [15].

Our research will develop two complementary approaches to support automatic extraction of synchronization skeletons. The first is the use of slicing. In earlier work we applied slicing to hardware description languages [12]. In this project, we will extend these techniques to software by developing ways to use slicing techniques for extracting synchronization skeletons from concurrent programs. The second approach will be to allow formal specification of the properties that must be checked, and then use those specifications to automatically extract the code fragments that are relevant for model checking. This work is similar to that of Dawson Engler, who has investigated extraction of finite state descriptions from C programs by eliminating fragments of the program that are unnecessary for verifying a property of interest. However, in his work properties are specified operationally using C-programs and extracted manually from the source code. In the project, we intend to develop more automated techniques in which properties of interest are specified declaratively.

Our second attack on the problem of usability will focus on automated creation of models for task execution languages. In many cases, specialized languages are used to implement task executives for autonomous systems. Languages such as RAPs [18], ESL [20], and TDL [43], typically contain constructs for hierarchical goal decomposition, task synchronization, resource management, execution monitoring, and exception handling. In such languages, the synchronization constraints are explicit, so extraction and translation of constraints are facilitated. In previous research at Carnegie Mellon, in collaboration with Ames [37,44], we developed a tool to produce SMV models automatically from models written in MPL, the language used in the Livingstone fault diagnosis system [46].

We propose to create similar translators for task execution languages. The difficulty here is that such languages are typically quite expressive, and care must be taken to produce formal models that can be verified efficiently. In addition, we propose to develop specialized specification languages that enable engineers to encode properties of interest easily. We will focus on being able to encode flight rules from various missions, such as DS1, to test the approach. In some cases, we will extend the task execution languages so as to be able to support directly the inclusion of such properties, much as the “assert” property in C.

Our third approach to improving usability will be to provide automated model generators for *publish-subscribe* and *cyclic-task* software architectures. To support coordination among behaviors, while maintaining loose coupling between tasks, autonomous systems often adopt one of two forms of component integration:

1. Using publish-subscribe interaction [45] tasks are responsible for announcing (or “publishing”) significant events. Other tasks may register to be informed of (or “subscribe to”) a set of events. When one task publishes an event, it is sent to all subscribers. In this

way tasks remain independent since a publisher does not know the identity, or even existence, of other subscriber tasks.

2. Using cyclic tasks with shared variables, a similar effect is typically achieved by assigning each task to a task slot that is repeatedly run at some fixed period — say every 50 milliseconds. At each activation a task examines a set of input variables in some shared variable space and, based on those values, it writes derived values to a set of output variables. Readers of shared variables do not know which tasks set those values, or which tasks will "consume" the values of the variables to which they write. Thus, the shared variables serve the role of the events in the publish-subscribe architectures of autonomous systems, with the analogous goal of decoupling tasks while permitting communication.

While both architectures are good from an engineering point of view, reasoning about their aggregate behavior is problematic. For example, one would like to be able to guarantee that if a sensor detects that a temperature value goes over a maximum limit then some other part of the system will take appropriate corrective action to preserve overall stability. This is hard to do with such systems, however, because interactions between the parts of the system are indirect and asynchronous. In particular, the non-determinism inherent in the invocation of tasks and the communication delays inherent in a distributed publish-subscribe system (many events may be in transit) make it difficult to reason about properties such as global invariants or timely response to certain trigger events. (For example: Are sufficient events published to allow interested parties to respond in a timely fashion? Is there potential interference between components that subscribe to the same event?) Analogously, for shared-variable cyclic systems it is difficult to determine whether a given assignment of tasks to periodic buckets and the ordering of tasks within a bucket are sufficient to guarantee some property under all possible scenarios. (For example: Are there implicit ordering assumptions in the reading and writing of shared variables that may be violated?)

While model checking would appear to be a good solution for verification of such properties, it is not immediately obvious how to map these kinds of systems into appropriate models. In previous research we have developed techniques for reasoning about such systems [17]. The basic idea underlying the work is that one can provide a precise description of the cause and effect of each event to reason locally about how a component behaves in isolation. Then, using local correctness of the components and properties of events, we can infer the desired global property of the overall system. We believe it should be possible to map these descriptions automatically into a form that can be model checked. Moreover, many of the standard problems about event ordering and interference (mentioned above) can be built-in as standard checks that are automatically verified. Initial steps towards developing such an automated translator for publish-subscribe systems is described in [19].

The fourth approach to improving usability is developing new ways to help designers understand the meaning of the counterexamples produced by a model-checker. Current work, funded by NASA, is developing algorithms that can produce causal explanations of counterexamples for MPL models [44]. This work instantiates a counterexample and its associated SMV model into a TMS (Truth Maintenance System). By propagating dependencies, a justification tree is created, which can then be manipulated to form a linear explanation. For this project, we will apply and extend this work to explain counterexamples for task executive programs. In particular, we will use techniques adopted

from AI planning to find concise explanations that focus on the most germane portions of the justification tree. We also propose to develop techniques that enable developers to view and browse the temporal evolution of counterexamples by creating virtual execution traces from the counterexamples that can be utilized by existing visualization tools for task executives.

3.1.2 Improving Applicability

While many synchronization properties are qualitative in nature ("X must occur before Y"), many more are quantitative ("X must occur at least T seconds before Y"). Although there has been considerable work already on verifying real-time systems, including research at Carnegie Mellon [6], more is needed to handle models of the complexity needed for autonomous space systems.

In understanding why a quantitative synchronization constraint has failed, it is often useful to know how far the system actually falls outside of the desired bounds. Previous research at Carnegie Mellon on model checking of discrete real-time systems has introduced the concept of quantitative timing analysis [7]. Using the new method it is possible not only to check the correctness of a system, but also to compute quantitative timing information about the behavior of the system, such as minimum and maximum times between the occurrences of events. The method can also determine the number of occurrences of an event in all paths between two other specified events. Moreover, it is also possible to identify how often a task has been blocked, or if lower priority tasks have been executing in the interval between the task's execution start and finish times. With this information it is possible not only to assert the correctness of a system, but also to understand its behavior and in many cases to identify optimizations to the design. Even more detailed information can be obtained using selective quantitative analysis, which allows the analysis of specific execution sequences in the model, instead of checking all of its behavior.

While quantitative timing analysis is a powerful technique, more research is needed before it can become a practical tool. One issue is that distributed systems are often better modeled using continuous time rather than discrete time. This is particularly true for quantitative timing analysis. Currently, it is possible to determine minimum and maximum time delays between events. However, other verification algorithms such as condition counting or selective quantitative analysis are not supported. We propose to extend quantitative timing analysis to continuous-time models and to develop model checking tools that can handle task execution systems of realistic complexity.

A second limitation of model checking technology is its inability to handle systems whose task structure varies at run time. Current techniques require one to create models that contain all possible tasks, even though some may not be active at any given moment. The creation or deletion of a new task is then modeled by activating and deactivating tasks. This approach has the disadvantage of requiring very complex models, and of not closely matching the structure of task executives, in which tasks are freely created and deleted. We propose to investigate extensions to model checking that will permit more natural representation of such systems. We will develop novel techniques that can modify transition relations of concurrent systems by dynamically creating new BDD variables when required.

3.2 Verifying Resource Constraints

Space systems are typically resource-limited, and so autonomous space systems must be concerned with resource allocation, ensuring that they do not exceed resource bounds. While the use of autonomous planning and scheduling can significantly decrease resource conflicts [35], problems can still arise [36]. Model checking is particularly well suited for detecting potential violations of resource constraints because it can examine all possible execution traces and interleavings of tasks.

Resources can be categorized into four types [16]:

1. Single sharable - a discrete, single unit, such as a camera. Sharable resources can be used by only a single task at any given time.
2. Multiple sharable - multiple discrete units, such as a team of rovers exploring a planetary surface. Multiple sharable resources may be interchangeable, but difficulties arise when the resources are interchangeable for some purposes but not others (e.g., wheeled rovers, but with different sensors).
3. Consumable - continuous resource, such as fuel. Consumable resources can be used for any task, at any time, but eventually can be used up.
4. Renewable - continuous resource that can be replenished, such as disk space or telemetry bandwidth. Renewable resources are critical in space systems, and must be very carefully managed to avoid over-subscription.

Single sharable resources can be synchronized using a mutex Boolean variable. It is straightforward to model this with model checkers. Multiple sharable resources can be represented using an integer variable with a given, finite range. While, in general, allocating multiple sharable resources is a computationally complex problem, verifying the absence of resource conflicts is no more difficult than other discrete, symbolic model-checking problems.

On the other hand, the major research issue in dealing with resource constraints is reasoning about continuous (consumable and renewable) resources. Here, we propose to adapt real-time model-checking algorithms [6,28]. In many cases, resources are consumed and renewed at a constant rate and their constraints are independent. This corresponds to multi-rate clocks, or rectangular automata, for which many verification problems of practical interest are decidable [2, 24]. Consumption and renewal rates that are not constant can be modeled in this framework using piecewise constant approximations.

Often, it is important to know not just whether a resource constraint has been violated, but how close one is to a violation in the best and worst cases. This gives the engineer an understanding of potentially critical points in the design. Again, multiple clock rates can be used to model the dynamics of the consumption/renewal processes. Reachable sets of the system trajectories can be computed as fixed points of operators in the space of continuous clock values. This produces polyhedral regions that can then be introduced as linear constraints to evaluate the worst-case proximity to specified resource constraints. We propose to create tools that automate this process of computing these values for critical constraints selected by the designer.

Another significant research issue is how to specify resource constraints and utilization in a way that can be automatically processed for formal verification. We propose to develop a syntax for

specifying such constraints and embed this language in the engineer's programming environment - either by extending an existing task description language, such as ESL or TDL, or by defining them as stylized comments that can be added to general-purpose programming languages, such as C. For this, we will borrow heavily from work in planning and scheduling [35], where declarative representations of resources are commonly used.

We will then develop translators to create SMV models automatically from the resource constraint specifications, which would then be combined with the models generated for synchronization constraints (Section 3.1) to determine if the constraints hold for all possible task executions. We will also extend our work in automatic generation of explanations to handle counterexamples that demonstrate resource violations. The development of these automatic translation and explanation capabilities, together with the development of novel model-checking algorithms to handle continuous resources, will provide engineers with important and valuable new tools for verifying resource-limited space systems.

3.3 Verifying Environmental Constraints

Constraints related to the dynamic behavior of a system and its interactions with the environment are among the most difficult to incorporate into a formal verification procedure. In part, this is because traditional formal methods are based on discrete-state models that cannot easily capture the continuous parameter variations that characterize system dynamics and environmental interactions, and in part because the environment's behavior is typically uncertain and not represented well (if at all) in the models used for synchronization and resource allocation. Nevertheless, system dynamics and environmental constraints cannot be ignored when evaluating the feasibility or correctness of a planned task. For autonomous systems that must remain viable for long missions, it is essential that these constraints be represented and evaluated as faithfully as possible, so that the system can be operated to its full limits while avoiding catastrophes. Assuring that dynamic constraints are not violated by over designing the system is not an attractive option when there are stringent constraints on the available size, weight, and energy available over the mission lifetime. Unfortunately, satisfying environmental constraints via conservative planning and execution may make it impossible to carry out the mission objectives.

We propose to address the problems imposed by the continuous-time, uncertain nature of a system's environment by applying and extending recent advances in two areas: the modeling and formal verification of hybrid dynamic systems and the use of stochastic models to represent and analyze behaviors in uncertain environments. We describe at each of these approaches in turn.

3.3.1 Handling Both Continuous and Discrete State Variables

The operation of autonomous systems, and their interactions with the physical world, are governed by the laws of continuous dynamics, usually modeled by differential and algebraic equations. In contrast, current formal verification tools are based on models of computation that are inherently discrete state/event models. Timing is typically the only mechanism for incorporating features of physical environments into these models. Our goal is to create tools for analyzing models of

physical dynamic systems, by abstracting tractable behavioral models and verifying task plans with respect to environmental constraints.

Autonomous systems interacting with physical environments can be modeled as hybrid dynamic systems, that is, systems with both continuous and discrete state variables. Hybrid dynamics can also appear in the environment itself when the continuous dynamics change depending on discrete conditions, such as changes that occur in mechanism dynamics depending on whether certain surfaces are in contact. Central issues in recent research on hybrid dynamic systems include numerical methods for simulation, identifying and characterizing qualitative behaviors such as invariants and limit cycles, synthesizing controllers for various objectives such as stability and reachability, and formal verification [34,41].

This latter research on formal verification is most germane to the present proposal. In contrast to model checking for finite-state systems, algorithmic verification is possible for hybrid dynamic systems with only very simple classes of continuous dynamics [25,26,33]. Consequently, the best one can hope for is to use conservative models that represent outer or inner approximations to the families of exact system behaviors. Based on our experience and that of others in verifying properties of hybrid dynamic systems, we plan to generate discrete system models from continuous ones, and thus handle more generally continuous state variables, where time is just a special (but important) case.

We propose to use differential equations of the system to generate discrete models of the environment that can be composed as needed with existing discrete models of the computing system. This approach has been taken in the past, but the discrete models of the continuous dynamics have been either created manually and in an *ad hoc* manner [38], or generated using numerical techniques that do not guarantee the model is correct [31].

Our work on discrete-state approximate quotient transition systems will form the basis for these models [8]. The fundamental problem in obtaining discrete models of hybrid systems is the computation and representation of reachable sets for the continuous dynamics. Current methods for computing these reachable sets are effective only when there are no more than three or four continuous state variables [9,39]. To deal with this problem, we will investigate methods for decomposing the continuous dynamic models into low-dimensional interacting subsystems, each of which can be handled effectively with our computational tools. Formally, this approach corresponds to projecting the full state vector into lower-dimensional subspaces as proposed in [22]. Alternative representations of continuous dynamics will also be investigated, including ellipsoidal approximations that are possibly less complex than the polyhedral approximations we are currently using [32].

The key to this approach will be the ability to select the appropriate operating regime of the continuous dynamics that need to be approximated. Toward this end, we will create tools that make it possible for the domain expert who is familiar with the dynamics and the requirements to be verified to define the range of operation that needs to be captured.

Our approximation approach here is analogous to our approach for extracting synchronization skeletons of Section 3.1, which are yet a different abstraction of the system. It is also consistent with our focus on task-level execution, allowing us to analyze tractable models of inherently large state spaces due to continuous dynamics.

3.3.2 Modeling Uncertainty of the Environment

In many situations, absolute guarantee of correct behavior is infeasible, due to the uncertain behavior of the environment. What is required are estimates of the likelihood that properties will be violated. We can model the environmental events as nondeterministic transitions and, in some cases, estimate the probabilities of events using information like past mission data. This makes it possible to associate probabilities with the state transitions that correspond to these events. As a result, the environment of an autonomous system can often be modeled as a stochastic process.

Numerous theoretical studies have been written on the probabilistic verification problem. Algorithms for solving the problem have been given in several papers; for example, there is a model-checking algorithm for linear temporal logic that is exponential in the size of the specification and polynomial in the size of the Markov chain. However, currently no probabilistic model checkers are able to verify realistic systems. The bottleneck is the construction of the state space and the necessity to solve huge systems of linear equations. We are developing a more efficient alternative, which performs the probability calculations using Multi-Terminal Binary Decision Diagrams (MTBDDs) [10]. MTBDDs differ from Binary Decision Diagrams (BDDs) in that the leaves may have values other than 0 and 1; in this case the leaves contain transition probabilities. Preliminary results indicate that MTBDDs can be used to represent extremely large transition probability matrices.

Transition probability matrices represented as MTBDDs can be integrated with a symbolic model checker, and have the potential to outperform other matrix representations because they are so compact. For example, Hachtel and colleagues [4] have developed symbolic algorithms to perform steady-state probabilistic analysis for systems with finite state models of more than 1027 states. While it is difficult to provide precise time complexity estimates for probabilistic model checking using MTBDDs, the success of BDDs in practice indicates that this is likely to be a worthwhile approach.

For writing specifications we use Probabilistic real-time Computation Tree Logic (PCTL) [5]. PCTL augments the logic CTL (developed by Clarke and Emerson) with time and probability. Formulas are interpreted over finite state discrete-time Markov chains. The logic is very expressive and has a simple model-checking algorithm that can be implemented using MTBDD-based techniques. A model checker for this logic can be a useful tool in the dependability analysis of fault-tolerant real-time control systems, performance analysis of commercial computer systems and networks, and operation of automated manufacturing systems.

We are implementing a model checker for PCTL within Verus, a verification tool tailored for analysis of real time systems [6]. In order to model and verify stochastic systems in Verus, we have extended the specification language and implemented a subset of the PCTL operators. We have

already verified a few examples, which include a simple communication protocol, an automated manufacturing system comprising two machines, and a fault-tolerant computing system. Currently, we are verifying a railway-interlocking controller with a state space in the order of 1022 states. For this program, we will implement the full PCTL logic, continue the development of the required MTBDD operators, and evaluate how well PCTL model checking performs on NASA-relevant verification problems.

Our work on implementing an MTBDD-based PCTL model checker for a version of Verus lets us incorporate the stochastic nature of an autonomous system's environment into our models. However, it fundamentally relies on the assumption that events occur independently. In practice, events, especially failure events, do not necessarily occur independently. For example, in a spacecraft with two redundant nodes used for load balancing, if one node fails, then the other node is more likely to get overloaded and subsequently fail. Also, an exceptional or unanticipated external environmental event usually triggers a mission shutdown procedure (e.g., if the sensor's current reading exceeds the maximum threshold for safe operation). The shutdown procedure will either (1) cause further failure recovery subtasks to execute, effecting a cascade of failure events, or (2) simply terminate the current task.

In our work on modeling and analyzing survivable systems [30], we have developed a stochastic model called Constrained Markov Decision Processes (CMDPs) [1]. A CMDP can have a mix of nondeterministic and probabilistic state transitions. Consequently, a CMDP is more general than a nondeterministic state machine. CMDPs are also a generalization of Markov Chains, since the transition probabilities can depend on past history. Thus, CMDPs can be used to model a wider range of systems than the methods discussed previously.

Existing CMDP algorithms can be used to do reliability analysis via the following: (1) compute the worst-case probability that a task started will eventually finish, (2) do latency analysis, i.e., compute the worst-case expected finishing time of a task, and (3) do cost-benefit analysis, thus helping the engineer make design decisions. CMDP models have been successfully used in applications from the telecommunications industry and decision sciences.

In our preliminary work we have built a model checker that can verify properties of CMDPs [29] and have successfully applied it to two non-trivial examples in the financial services application domain. However, we have not tried to use the model checker to verify properties of autonomous systems. For this program, we will verify large-scale examples involving autonomous systems with newer versions of the tool.

In addition to modeling history-dependent events and enabling reliability and latency analysis, another important advantage of using a general model like CMDPs is its ability to model cost constraints. Using costs associated with state transitions, we can perform cost-benefit analysis. For example, suppose we have a fixed budget and need to decide which nodes should be upgraded to achieve maximum increase in system reliability. We may not be able to upgrade all nodes and must choose those that would be most cost-effective. In a CMDP, cost information is represented very generally in terms of functions that assign real values to <*state, action*> pairs in the state transition model. We expect to be able to use these functions to model the kinds of resources described in

Section 3.2. This idea is still unexplored territory – in the long term, we propose to understand how the CMDP model can help us do cost-benefit analysis of resource usage.

More generally, we propose to identify the practical and realistic environmental conditions under which the Markov assumption applies. We can use tools based on MTBDDs and PCTL to analyze such examples. More ambitiously, we propose to identify and express the dependencies (especially due to failure events) resulting from dynamic interactions between the system and environment. Here, we need to develop our CMDPs methodology further to handle these more complex transitions. Finally, we plan to investigate how CMDPs can be used to model cost constraints as they relate to resource usage, thus extending the types of resource constraint problems that can be handled.

4 Research Plan

Our proposed work is divided roughly into two categories:

(1) Tools for Automating Verification of Autonomous Systems: Using technology at hand (model checking), we will apply it immediately to what is feasible - specifying and verifying synchronization and sharable resource constraints. The research here will focus on making it easier for engineers to use model checking by automating the model construction process. Hence, we will work on extracting synchronization skeletons, modeling publish-subscribe systems, and translating specialized task executive languages. We will also deal with discrete resources, develop languages for specifying properties of interest for task executives, and investigate methods for explaining counterexamples. All this proposed work is feasible and will be the focus of our attention in the first two years of the project.

(2) Novel Verification Techniques Geared Towards Autonomous Systems: We will develop brand new methods, and supporting tools, to reason about metric resources and environmental constraints. Here, in contrast to synchronization constraints, we lack a firm understanding of all the issues involved, ranging from the kinds of models appropriate for capturing behaviors of interest, to the logics with which to state the constraints. Since more fundamental research is required for this part of our proposed work, we expect to explore the ideas as outlined in Sections 3.2 and 3.3 during the first two years of the project so that by the third year we can build the relevant tools and apply them to NASA-relevant examples.

Our proposed budget ramps up according to this research plan. In the first year, our budget goes to support the short-term efforts of (1) while laying foundations needed for (2). As time progresses, the budget will go to continuing support of (1) but with primary focus on (2).

5 References

[1] E. Altman. *Constrained Markov Decision Processes*. Chapman and Hall, 1998.
[2] R. Alur, T.A. Henzinger, and P.H. Ho. Automatic symbolic verification of embedded systems.1993.
[3] R. Alur, T.A. Henzinger, and P.H. Ho. Automatic symbolic verification of embedded systems. *IEEE Trans. on Software Engineering*, 22(3), Mar 1996.

[4] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *ICCAD '9*3, Nov. 1993.
[5] H. Bansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computin*g, (6):512–535, 1994.
[6] S. V. Campos, et al. Verus: a tool for quantitative analysis of finite-state real-time systems. In *ACM Wkshop on Languages Compilers and Tools for Real-Time System*s, 1995.
[7] S. V. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Real-Time Systems Symposiu*m, 1994.
[8] A. Chutinan and B. H. Krogh. Approximate quotient transition systems for hybrid systems. In *2000 American Control Conferenc*e, June 2000.
[9] A. Chutinan and B.H. Krogh. Computing polyhedral approximations to dynamic flow pipes. *The IEEE Conference on Decision and Contro*l, 1998.
[10] E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Internl. Wkshop on Logic Synthesi*s, May 1993.
[11] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Worksho*p, LNCS, 1981.
[12] E.M. Clarke, M. Fujita, P.S. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Proc. of Charme '9*9. Springer-Verlag, 1999.
[13] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Survey*s, 24(4), December 1996.
[14] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proc. of ICSE 200*0, June 2000.
[15] J.C. Corbett. Constructing compact models of concurrent Java programs. In *Proc. of the 1998 Symp. on Software Testing and Analysi*s, March 1998.
[16] K. Currie and A. Tate. O-Plan: The open planning architecture. *Artificial Intel.*, 52, 1991.
[17] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Towards a formal treatment of implicit invocation. *Formal Aspects of Computin*g, 1998.
[18] R. James Firby. An investigation into reactive planning in complex domains. In *Proc. National Conference on Artificial Intelligenc*e, Seattle, WA, July 1987.
[19] David Garlan and Serge Khersonsky. Model checking implicit invocation systems. In *Proc. of the 10th Internl. Wkshp on SW Specification and Desig*n, San Diego, CA, Nov. 2000.
[20] Erann Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. In *Proc. of the AAAI Fall Symposium on Plan Executio*n. AAAI Press, 1996.
[21] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Symp. on Princ. of Prog. Lang*., pages 174–186. ACM Press, January 1997.
[22] M.R. Greenstreet and I. Mitchell. *Hybrid Systems: Computation and Control, 2nd Internl Workshop, LNC*S1569, Reachability analysis using polygonal projections, 1999.
[23] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT*), 2(4), April 2000.
[24] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *J. Comput. Sci*., 57(1), 1998.
[25] T.A. Henzinger. Hybrid automata with finite bimulations. In *ICALP 95: Automata, Languages, and Programmin*g. Springer-Verlag, 1995.

[26] T.A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Scienc*e. IEEE Computer Society Press, 1996.
[27] T.A. Henzinger, P.H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Trans. on Automatic Contro*l, 43(4), April 1998.
[28] T.A. Henzinger, X. Nicolin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computatio*n, 111(2), 1994.
[29] S. Jha and J. Wing. Survivability analysis of networked systems. Submitted to *International Conference on Software Engineerin*g, August 2000.
[30] S. Jha, J.M. Wing, R. Linger, and T. Longstaff. Analyzing survivability properties of specifications of networks. In *Proc. of the Internl. Conference on Dependable Systems and Networks, Workshop on Dependability Despite Malicious Faul*t, June 2000.
[31] S. Kowalewski, S. Engell, and O. Stursberg. *Advances in Contro*l, chapter Verification of logic controllers for continuous plants. Springer, 1999.
[32] A. B. Kurzhanski and P. Varaiya. Ellipsoidal techniques for reachability analysis. In *Hybrid Systems: Computation and Control, 3rd International Workshop, LNCS* 1790. Springer, 2000.
[33] J. S. Miller. Decidability and complexity results for timed automata and semi-linear hybrid automata. In *Hybrid Systems: Computation and Control, 3rd Internl Wkshp LNC*S1790, 2000.
[34] A. S. Morse, C. C. Pantelides, S. Sastry, and J. M. Schumacher (editors). A special issue on hybrid systems. *Automatic*a, 35(3), March 1999.
[35] Nicola Muscettola. *HSTS: Integrating Planning and Schedulin*g. Morgan Kaufmann, 1994.
[36] Pandu Nayak, *et al.* Validating the DS1 Remote Agent experiment. In *Proc. Internl Symp. on AI, Robotics and Automation for Space (i-SAIRAS*), The Netherlands, June 1999.
[37] Charles Pecheur and Reid Simmons. From Livingstone to SMV: Formal verification for autonomous systems. In *Goddard Workshop on Formal Method*s, April 2000.
[38] H. A. Preisig. A mathematical approach to discrete-event dynamic modeling of hybrid systems. *Computers and Chemical Engineerin*g, 20(pt. B), 1996.
[39] J. Preußig, O. Stursberg, and S. Kowalewski. Reachability analysis of a class of switched continuous systems by integrating rectangular approximation and rectangular analysis. In *Hybrid Systems: Computation and Control, 2nd International Workshop*,,1999. Spr-Verlag.
[40] Sriram Rajamani and Thomas Ball. Bebop: A symbolic model checker for Boolean programs. In *SPIN 2000 Workshop on Model Checking of Softwar*e, August 2000.
[41] Elsevier Science. Spec. issue on hybrid systems. *Theoretical Comp. Sci.*, 138(1), Feb 1995.
[42] I. Silva and B. H. Krogh. Formal verification of hybrid systems using CheckMate: A case study. In *2000 American Control Conferenc*e, June 2000.
[43] Reid Simmons and David Apfelbaum. A task description language for robot control. In *Proc. International Conference on Intelligent Robots and System*s, October 1998.
[44] Reid Simmons and Charles Pecheur. Automating model checking for autonomous systems. *AAAI Spring Symposium on Real-Time Autonomous System*s, Stanford, CA, March 2000.
[45] K. Sullivan and D. Notkin. Reconciling environment integration and software evolution. In *Proceedings of SIGSOFT '90: 4th Symp. on SW Development Environment*s, Dec.1990.
[46] B. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings National Conference on Artificial Intelligenc*e. AAAI, August 1996.

Management Plan

Jeannette Wing (PI) will have overall management responsibility for the project. She will also be responsible for the probabilistic and quantitative timing analysis techniques.

Edmund Clarke (co-I) will have primary responsibility for extracting synchronization constraints and metric model checking (for temporal and resource constraints).

David Garlan (co-I) will have primary responsibility for formalizing publish-subscribe systems and developing related tools. He will share responsibility for developing languages for specifying properties.

Bruce Krogh (co-I) will be responsible for issues relating to verification of hybrid systems. In addition, he will provide expertise in real-time systems.

Reid Simmons (co-I) will have primary responsibility for modeling task executive languages and explaining counterexamples. He will share responsibility for developing languages for specifying properties. He will also provide expertise in autonomous systems.

Several of the investigators have already collaborated on research, which should facilitate joint research and integration of results. We will have regular group meetings to discuss research progress, latest developments in the field, and NASA relevance. We will submit regular reports to NASA and make available all tools developed, along with test data.

We already have a good collaborative relationship with the verification group at NASA Ames, and expect this will continue. In fact, this relationship should be strengthened as plans progress for a Carnegie Mellon presence on, or near, the Ames campus.