

Verification Tools for Autonomous and Embedded Systems

*Randal Bryant, Edmund Clarke, David Garlan,
Bruce Krogh, Reid Simmons, and Jeannette Wing*

Computer Science Department
Electrical and Computer Engineering Department
The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

April 8, 2000

1 Overview

Society depends increasingly on computers to perform tasks that affect and enrich our daily lives. These computers help run our power plants, monitor our heartbeat, drive our cars, cook our food, and detect strangers in our house. They enable us to install satellites in space and to build earthquake-sensitive buildings. Increasingly these computing systems are required to perform more complex tasks and to do so at greater and greater levels of autonomy. Each new generation of a system raises people's expectations in terms of its functionality, performance, reliability, and safety. For many of these systems, software errors can result in loss of equipment or even human life. Our responsibility as computing professionals is to provide the best possible technology to ensure that people's trust in these systems is justified.

Assuring that autonomous and embedded systems will operate correctly and safely in all situations has become increasingly difficult for several reasons. To deal with a wider range of problems and operating conditions, the decision procedures and algorithms implemented in computer software have become much more sophisticated. Decomposition and modular design have led to complex multitasking systems wrought with all the difficulties that can arise in concurrent real-time systems. The environments in which the systems operate are more diverse, and each is more complex and less predictable. As we call on autonomous and embedded systems to perform more control functions, the effects these computing systems have on the environment and the environment's subsequent reactions (the feedback loops) become more important in assessing the performance and safety of the system. Finally, these complex systems are usually not simply designed and deployed. They evolve continually. They are changed and modified to fix problems, to increase robustness, and to add new features that were never anticipated in the original design.

All of these issues have contributed to a growing concern that current methods for designing and evaluating complex autonomous and embedded systems are inadequate. Indeed, the highly publicized failures of recent NASA missions have confirmed that system designers need new tools to be able to detect and debug problems before they lead to catastrophe.

The degree of assurance we can provide today is based on extensive simulation and testing. The goal of simulation is to catch errors as early as possible in the design phase to reduce the need for more costly testing of the implemented system. In some cases, simulation is the only option; testing may be impossible because the system's intended effect is irreversible, e.g., launching a spacecraft. Both simulation and testing suffer from being incomplete: each simulation run and each test evaluates the system performance for only a single set of operating conditions and input signals. For complex autonomous and embedded systems it is impossible to cover even a small fraction of the total operating space with simulations. Finally, testing is already too expensive; today building a test harness to simulate a component's environment is more expensive than building the component itself.

The focus of our proposed research is to complement the traditional methods of system validation through simulation and testing with formal methods, that is, with methods that evaluate the performance of a system over a large

(possibly infinite) set of operating conditions without resorting to exhaustive simulation. Formal methods have been developed and applied successively to problems in computer software and hardware design, and in protocol verification. Our long-term vision is to see formal methods used regularly by designers of autonomous and embedded systems. We believe domain-specific tools based on formal methods will make it possible to design and deploy complex autonomous and embedded systems with a level of confidence that far exceeds what can be achieved today using simulation and testing. Toward this vision, there are many technical challenges, pragmatic and theoretical. There are also educational challenges. Our proposal addresses the challenges on both fronts.

1.1 Application Domains: Autonomous and Embedded System

Autonomous systems are designed to perform tasks independently, or with very limited external control. They are needed in situations where human control is either infeasible or not cost-effective. Embedded systems are situated inside a physical device, such as a car or a microwave oven. Besides enhancing the functionality of the device, embedded systems can increase the device's reliability by monitoring basic functions, such as fuel injection in an automobile. Characteristics common to autonomous and embedded systems are that (1) the systems operate in highly variable, uncertain, and time-changing environments; (2) they must meet real-time constraints to work properly; and (3) they are often interconnected with other agents, both humans and other machines. Here are four examples:

- *Spacecraft.* Due to time delays and bandwidth limitations in communication, spacecraft must take responsibility for detecting and recovering from a wide variety of potential hardware and software faults. NASA is developing more autonomous spacecraft systems, such as the Remote Agent that flew on the Deep Space One mission and the next generation of Mars rovers. The Air Force and companies such as Iridium and Teledesic are launching constellations of satellites that need to coordinate activities and monitor their status. Greater autonomy of operation is inevitable as the number of satellites grows.
- *Automobiles.* Every car now has several embedded microchips to control functionality from fuel injection to anti-lock brakes to GPS navigation. These computers will control more and more of a vehicle's operation, eventually slowing the vehicle automatically when it approaches other vehicles, warning drivers of impending accidents, communicating with other cars to form "intelligent highways," and even driving themselves.
- *Smart homes.* The home of the future will be a complex network of appliances with embedded computers. We already have examples of smart appliances from microwave ovens to fuzzy logic rice cookers and dishwashers. Tomorrow, our smart home will notify repairmen when an appliance detects it is broken, sense the presence of friends or strangers, regulate power consumption, and in general, interact with one another to coordinate activities.
- *Office and home robots.* These future robot systems will need to contend with all the complexities of sensing, planning, and acting in real time in an uncertain, dynamic environment; to interact intelligently with humans and other robot systems; and to guarantee the safety of themselves and the people they encounter. Mobile robots, such as Carnegie Mellon's Xavier and Minerva, have demonstrated their reliability through extensive experimentation, but in limited environments; we are far from giving quantitative assurances of safety that would be needed before deploying such robots more widely.

In all the above applications, the need for guarantees of safety, reliability, and overall system correctness is acute.

1.2 Formal Verification

Formal verification is the process of determining whether a system satisfies a given property of interest. Today the best known verification methods are model checking and theorem proving, both of which have sophisticated tool support and have been used in non-trivial case studies, including the design and debugging of microprocessors, cache coherence protocols, internetworking protocols, smartcards, and air traffic collision avoidance systems (see [6] for other examples). Model checking in particular has enjoyed huge success in industry for verifying hardware designs. Within the past four years, companies such as Intel, Motorola, Hewlett-Packard, and Texas Instruments have started in-house model checking groups. Companies such as Lucent, Cadence, and Synopsys market formal verification tools to hardware designers.

Formal verification can be used to provide guarantees of system correctness. It is an attractive alternative to traditional methods of testing and simulation, which for autonomous and embedded systems, as argued above, tend to be expensive, time consuming, and hopelessly inadequate. By formal verification we mean not just the traditional notion of program verification, where the correctness of code is at question. We more broadly mean design verification, where an abstract model of a system is checked for desired behavioral properties. Finding a bug in a design is more cost-effective than finding the manifestation of the design flaw in the code.

Unfortunately, after decades of research formal verification has not become part of standard engineering practice. One reason is that techniques do not scale: code size is too large for practical program verification; the underlying mathematical formalisms (i.e., logics) do not handle all features of the programming language or all behavioral aspects of the system; and proof methods lack compositionality. Another reason is that tools do not scale: model checkers are limited by the size of the state spaces they can handle; theorem provers require too much human time and effort for too little perceived gain; and the tools are not integrated to work with others found already in the engineer's workbench. In truth, verification is simply a very hard problem. It is extremely ambitious to think that one can completely specify the behavior of an entire system formally and then embark on a complete proof of its correctness.

While verification is a very hard problem, which will not go away and will just get harder over time, we believe we can make progress on it for two reasons. First, we take a more realistic attitude toward what we expect to achieve. Rather than try to verify an entire system, we are willing to verify only the critical pieces of a system for just their key properties. If deadlock would cause a disaster, then we should model just enough of the control aspects of the system so we can check for the possibility of deadlock; we can afford to ignore the irrelevant aspects. Second, we are not trying to solve the verification problem in its generality. By focusing on autonomous and embedded systems, we can exploit the constraints of the problem space to make the verification problem more tractable.

Putting these two reasons together then, ideally, we would like to add a row of buttons to the desktop of an application-specific engineer's workstation; each button represents a check that is performed on the system being designed or tested. For example, the engineer designing the airbag switch might have one button to check that the airbag is enabled for activation only under the correct conditions (e.g., when the speed of the car is above a certain level), and another button to check that the airbag is activated only when the appropriate conditions are satisfied (e.g., when accelerometer readings are within a certain set of ranges). Toward this ideal, we propose to develop languages, techniques, and tools aimed specifically at providing engineers of autonomous and embedded systems with ways of verifying their designs. We discuss our research plans in Sections 2, 3, and 4.

Finally, another reason that formal verification and its sister, formal specification, have not been widely adopted in practice is the failure to educate scientists and engineers with state of the art specification notations and verification tools. While lip service has been paid to the training and educational aspects of transferring formal methods to the "masses" no one in the United States has taken on this task with any deep commitment. An important part of our proposed work is to address this problem head on. We have very specific ideas on how to incorporate formal specification and verification into undergraduate and graduate courses. We also are asking for funds to support doctoral students, to pass on to the next generation of scientists and educators knowledge of current formal methods and to equip them with the research skills to solve tomorrow's challenges. Thus, our proposed work also has a strong educational mission; we flesh out our plan in Section 5.

2 Three Problem Areas Common to Autonomous and Embedded System

We begin with a few real-world "war" stories, some of which one of the PIs (Simmons) has been intimately involved with.

- *Deep Space One (DS1)*. In 1999, Deep Space One flew software to plan, execute plans, and monitor system health autonomously. This was the first use of autonomous software for an inter-planetary space mission.

"Formal verification was used early in the project to detect a bug in the 'executive' code in which, essentially, a thread could be terminated without releasing a lock it had, thus preventing other threads from accessing the resource. The solution was to enclose part of the termination code in a critical region.

During flight, a bug was discovered in the autonomous software. Apparently, the problem was two threads accessing a variable 'simultaneously' and inconsistently. The problem had never appeared in testing because it came after a long period of no communication—causing a backlog of telemetry. As the telemetry data was

being sent, the telemetry thread ran longer than usual, interfering with another thread. Again, the solution was to place a bit of software in a critical region.”

- *Livingstone (Model-Based Diagnosis)*: Part of the DS1 software (called Remote Agent), Livingstone consists of symbolic models of the spacecraft. Models are hierarchies of components, where each component has a number of “modes” that are either nominal or fault modes. Livingstone tracks the state of the system and looks for discrepancies between what the model predicts and what the sensors say. It can also use the models to plan recovery actions.

“I (Simmons) have been developing tools to automatically translate from Livingstone models to SMV and then verify certain properties of the models. Properties of interest to the Livingstone developers include (a) are the models consistent, (b) are the modes of every component reachable from the initial state, (c) if the spacecraft in a fault state, does the model allow Livingstone to plan a recovery to a safe state. Livingstone has been used to model the DS1 spacecraft, parts of the Carnegie Mellon Nomad robot, Xavier, and an in-situ resource utilization (ISRU) plant.

We found many bugs in the models. Common types of bugs include (a) the transition relation for component modes is incomplete, (b) two models are consistent individually, but are inconsistent when composed. For instance, one model had two components that were constrained to share a common power source. However, the first model explicitly stated that the “power” variable of the component was always on, and the other had an “off power” mode. Thus, it was inconsistent for the second component to ever be in the “off” mode. (c) starting in a fault mode, the only way to reach a safe mode would be to transition through another fault mode. This is not good—intentionally creating one fault in order to fix another fault. It turns out that one component was missing a transition.”

- Multi-robot coordination. The Mercator Project is developing teams of robots to explore the environment and to deploy to guard an area in a coordinated fashion. There are many problems that we have had to track down, through testing and debugging output, related to synchronization of the multiple robots.

“One scenario has the robots all moving together from one position to another. In order to avoid interference, we decided to stagger the robots—except for the first robot, the others all did not start moving until their neighbor got a certain distance ahead. All worked fine, until the time that the goal position was less than that distance. Then, the trailing robots all deadlocked, waiting for the lead robot to travel far enough.

The robot systems consist of multiple processes. The software is supposed to be robust enough to enable any process to go down and then restart, and the others will notice the reconnection and do the right thing. We’ve had many problems, however, about the processes ending up in inconsistent states. Sometimes, a process will not correctly acknowledge that another process has gone down, and continues sending messages to the dead process. Sometimes, when a process reconnects, the other processes will not send it sufficient information to re-establish its internal state (or, even worse, it ends up in an inconsistent state). Often, this manifests itself in deadlock. For instance, one process waits for an acknowledgement from the second process, but it never comes. This is very dependent on what was actually happening at the time the process goes down, and results tend to be highly variable and hard to reproduce.

Related to the above, but at a higher level. In multi-robot tasks, when one robot is lost, then the others are supposed to redistribute the tasks to take up the “slack”. Very often, at least part of what the lost robot was doing is never taken up.

The system is supposed to allow tasks to be terminated (for instance, if the user changes his mind or if environmental conditions make it impossible to continue with the tasks). When a task is prematurely terminated, the system is supposed to return to some ‘known’ state—in particular, it should perform any “clean up” actions so that future tasks can be executed. Since tasks can be terminated at any time, it is very difficult to guarantee that the right thing always happens.

Related to the above, sometimes we want to *suspend* (rather than terminate) tasks. The problem is to pick up at the right point when the task is resumed, especially since arbitrary things might have happened in the interim.”

- AAI Robot Competition “Several years ago I (Simmons) ran the AAI robot competition. The task was to explore a rock-filled arena, avoiding obstacles and collecting colored balls. One thing that sticks in my mind

was the cyclic behavior of several of the entries: They would see a ball behind a rock and turn to pick it up. Before picking it up, they would bump the rock. This would activate a behavior to turn away from the bump. After turning away, the robot would see the same ball behind the same rock and turn to pick it up... In some cases, this cyclic behavior went on for five minutes before random fluctuations in the turn mechanism caused the robot to unstick itself. But, it was very embarrassing for all.”

Reid should fill in/trim/rewrite above. Not sure how to depersonalize it or if we should. See ../reid-examples.txt for the other examples.

Bruce, would you like to add an example?

These examples of autonomous and embedded systems illustrate diverse behavior, but they have two general kinds of commonalities: their classes of desired *behavioral properties* and their *software architecture*. Our hypothesis is that we can exploit these commonalities to make it tractable to apply formal analysis to the concurrent, real-time challenges they pose.

We subdivide the verification task into three problem areas: design-level verification, modeling the system’s environment, and code-level verification. The bulk of our planned research focuses on the first two, addressed in more detail in Sections 3 and 4.

2.1 Design-Level Verification

Software written for autonomous and embedded systems typically abides by general architectural patterns. For example, many robot control systems can be viewed as task trees where nodes are tasks and synchronization constraints label the edges. Communication among tasks is done via message passing. Automotive control systems can be viewed as a collection of independent features or “task buckets” that communicate via shared variables and message passing. In both cases, there are several concurrent inter-dependent tasks that must satisfy a number of synchronization and timing conditions, in addition to logical and sequential constraints, to assure that the system will work properly. There are two specification challenges to this verification problem:

- *Specifying desired system characteristics.* Specifying what “correct” means is difficult because tradeoffs between functional and “extra-functional” properties (e.g., performance and reliability) determine acceptability. E.g., it is better that an autonomous car drives off the road than hit the non-autonomous car in front of it. Or, a robot delivering mail late but to the correct recipients may be preferable to a robot delivering mail on time but to the wrong people.

In Section 3 we discuss two classes of violations of desired system characteristics: ordering violations and synchronization violations.

- *Specifying the system abstractions.* Given that we can clearly define the interface between the system and its environment, which itself is a difficult problem, we are faced with the daunting task of understanding the system itself, from the hardware level, through its interface to software, and through the different levels of software abstractions (e.g., from executable code to system architecture).

In Section 3 we present a specific three-level architecture, applicable to autonomous and embedded systems, that breaks a system down into a low-level *behavioral* substrate, a middle-level *executive*, and a top-level *planner*.

The formal specification and verification community already provides much technology for attacking the specification problems that will arise in this part of our research work. It is a matter of sitting down and doing the work, in particular, working with the domain experts to tease out the essential aspects of their systems. We have confidence that the formal techniques and tools that we already have at our disposal or that we expect to enhance over the course of this project will be immediately applicable to autonomous and embedded systems. We view this part of the proposed research as the most feasible, least risky, and most likely to lead to immediate significant results, with the greatest impact on domain-specific engineers.

2.2 Modeling the System’s Environment

Autonomous and embedded systems must operate in an unpredictable environment. It is thus impossible to characterize the environment completely. Yet, we must guarantee some degree of safety and assurance of the system when

it executes. Moreover, once an interface between the system and environment is specified, by definition, it leaves undetermined what happens when the explicit assumptions captured in the interface specification are violated. There are two fundamental scientific challenges to this modeling problem:

- *Handling continuous and discrete time.* Autonomous and embedded systems are continuous, real-time systems; they continually interact with sensors and actuators that monitor the environment. One of the major challenges of our proposed research is to define appropriate mathematical models that handle both continuous and discrete time. While some theoretical work on hybrid systems has been done, much of it is impractical, inapplicable, or too simplistic.

Instead of pursuing this theoretical line of research, we propose a new approach, which we describe in Section 4. Our approach is based on using differential discrete models to generate discrete models of the environment that then can be composed with existing discrete models of systems.

- *Modeling randomness and faults.* Autonomous and embedded systems operate in the physical world where there is inherent uncertainty. We could use nondeterminism to model uncertainty, for example with a simple fault model that allows the environment to take a generic “fault” transition any time the system can make a transition. Alternatively we could use Markov models of the environment so that a system’s state transitions are probabilistic. In practice, we may need a combination of the two, and we may need further refinement of the fault model.

In Section 4 we discuss our plans to use stochastic variables to model probabilistic events and our proposal to investigate non-Markov models to accommodate the cascading effect of dependent faults.

We view this part of our proposed research as the riskiest and most technically challenging. Progress made along this line of research undoubtedly will lead to new science.

2.3 Code-Level Verification

Autonomous and embedded systems are by their very nature real-time programs. In the long run, verifying that a system meets its most basic real-time constraints requires proving that executable code is correct. This level of verification relies minimally on the correctness of the compiler or interpreter of the source level programming language. For example, for robot control software, we use a task description language (TDL) that compiles into C code; any proof that we do about a TDL program is relative to the correctness of the TDL-to-C compiler.

Much of the research in the programming language community focuses on this level of verification; thus, it will not be our primary focus, though we may work on related pieces of this problem, e.g., how it fits in with design-level verification.

3 Design Verification: The Problem and Proposed Research

Autonomous and embedded systems take sensor input and a goal (or set of goals) and decide to perform some action that affects the environment (and/or the internal state of the system). The primary problem in developing such systems is ensuring that they correctly respond, in a timely fashion, to every possible set of sensor inputs in order to achieve their goals.

We can separate the problem of reacting to inputs into *real-time* and *decisional* problems. *Real-time* refers to the system responding in time. We further categorize systems according to their *hard* and *soft* real-time deadlines. For hard real-time, missing a deadline can have disastrous effects (e.g., failing to enter orbit around a planet). For soft real-time, missing a deadline just reduces overall utility (e.g., failing to transmit scientific data during a given communication window). Typically, hard real-time performance is harder to obtain than soft real-time performance. *Decisional* problems, on the other hand, refer to the system making the right decision for a given situation. For instance, an autonomous Mars rover must decide how to steer given perceived stereo input of the terrain; an embedded anti-brake system must decide how to react depending on environmental conditions and what the driver is currently doing. Typically, decisional problems are more pervasive, but somewhat easier to diagnose and fix, than real-time problems.

Common Architectural View of Autonomous and Embedded Systems

Much work in the area of autonomous and embedded systems has gone into researching architectural frameworks to support system development. We intend to exploit the constraints inherent in this class of architectures to make the formal verification of decisional and real-time problems more tractable.

Autonomous and embedded systems are based on a set of concurrent *behaviors* that run either periodically (typical of embedded systems) or aperiodically (typical of autonomous systems [3]). A behavior can be thought of as a stimulus-response rule, or a data-flow transfer function, that reacts in a certain way to a set of inputs. Modern architectures for autonomous systems add two additional layers to the behavioral substrate (Figure ??) [2, 20, 10, 8, 25]. The top layer is used to perform higher-level *task planning*. The middle layer is an *executive* [7, 12, 11, 26] that decomposes the goals from the planner into executable tasks, sequences tasks, monitors task execution, and handles exceptions.

A key feature of this layered architecture is that the units of (concurrent) functionality are loosely coupled in order to provide ease of extension. Since behaviors are assumed to be fairly independent we can easily add new ones to the behavioral substrate; their concurrent execution provides for good reactive, real-time behavior. The layers also provide flexibility in tradeoffs between deliberation and reactivity, which enables autonomous systems to achieve complex tasks in rich environments. In the setting of an autonomous robotics systems, we add both new behaviors and new tasks incrementally by registering them with the arbitrator, or by adding new concurrent tasks at the executive level. In the case of automotive embedded systems, we incorporate new features by adding a new processing routine (often drawn from a library) to the collection of routines that must be invoked periodically by the run-time executive.

However, unlike behaviors, these larger units of concurrency, which we refer to as *tasks*, are not entirely independent: the actions of one task may affect the actions of another. For example, a change in the position of one robot in a coordinated robot group may need to be communicated to other robots in the vicinity so they can adjust their behavior. Or, a sensor might announce measured values that may be read by many other components dependent on the state of the object being sensed. Similarly, in an automotive context, a warning situation detected by one task in the system may need to be reacted to by some other task.

Common Bugs Due to Concurrency in Autonomous and Embedded Systems

While the constraints imposed by such architectures facilitate system development, they are also the potential source of undesired or unexpected task interaction. In particular, there are special classes of decisional and real-time bugs that frequently arise in the use of such architectures. These systems have problems stemming from *ordering* violations and *synchronization* violations. These problems are particularly difficult to find using traditional testing due to the large number of situations that must be tried and because the problems often arise only after particular *sequences* of events. We will address these classes of problems in our proposed research. Our basic claim is that the canonical structure of the architectures make these problems tractable for formal analysis. Design-time tools to detect (and diagnose) such problems would go a long way to improving the reliability of autonomous and embedded systems built using these architectural styles.

Ordering violations arise at the behavioral level when several behaviors recommend conflicting actions. For instance, in an automotive control system one behavior may recommend sounding a chime to indicate an open door while another recommends a different chime to indicate an engine problem. In many embedded systems, the behaviors are implicitly prioritized by the order in which they run. In autonomous systems, there is typically an explicit *arbitration* mechanism that chooses among the different behaviors [17]. In either case, bugs arise when the priority mechanism leads to the wrong choice of action for a given set of input conditions. Typically, this is because the developer has made some implicit assumption about the external, or internal, state of the system at the time the behavior is triggered. By making these assumptions explicit, and by reasoning about the interactions between behaviors, we can detect situations in which ordering violations can occur.

Synchronization violations typically appear at the executive level when tasks are mis-synchronized. Excess synchronization can lead to deadlock. Lack of synchronization can lead to resource conflict. For instance, a Mars rover may try to drive while it is taking a panoramic image, or two tasks may read and write to global memory simultaneously. In many cases, system developers use special-purpose languages to represent synchronization constraints, including languages based on hierarchical FSA [13, 24] and those based on more expressive formulations [7, 11, 26]. With such languages, the synchronization constraints are made explicit, and so can be reasoned about separately from the rest of the software. In other instances, the synchronization aspects of a system may follow certain prototypical patterns, and so can be extracted and reasoned about. We intend to tackle both the problem of analyzing and verifying synchronization constraints for special-purpose languages as well as extracting synchronization skeletons from general purpose code.

In Sections 3.1 and 3.2 we discuss how we plan to address each of these classes of problems. Section 3.3 cuts across both classes by focusing on how to make tools that can be used to analyze all levels of the system architecture more usable by the domain engineer.

3.1 Ordering Violations

To support coordination among behaviors, while maintaining loose coupling between tasks, autonomous systems typically adopt a form of publish-subscribe interaction. 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.

In embedded systems, such as automotive control systems, a similar effect is typically achieved using periodic scheduling and shared variables. Specifically, each task is assigned to a “bucket” of tasks that are 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 such systems are good from an engineering point of view, reasoning about their aggregate behavior is problematic. One would like, for example, 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 the stability of the whole system. However, this is quite hard to do with such systems because interaction between the parts of the system is indirect and asynchronous.

For publish-subscribe systems, the non-determinism inherent in the invocation of tasks and the communication delay inherent in a distributed 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. Analogously, for concurrent embedded 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.

To address this class of problem we propose to extend and apply our recent research on reasoning about implicit invocation systems [DGJN98a,DGJN98b] (Implicit invocations systems are a generalization of publish-subscribe and cyclic shared-memory systems.) In that research we developed formal framework for reasoning about such loosely coupled systems using linear time temporal logic and compositional reasoning. The basic idea underlying the work is that one can associate 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 infer the desired property of the overall system.

While this work provides formal underpinnings for reasoning about such systems, in its present form it is only the starting point for practical reasoning about autonomous and embedded systems. In particular, it has only been applied to small problems, it does not deal with timing properties, and it requires mathematical sophistication to reason with. Hence, to make it useful we propose to do three things: (1) Test the theory on more complex examples. The first step will be to understand how well we can capture and reason about realistic systems and properties that are of concerns to the developers of those systems. (2) Extend the theory. We need to extend the logic to account for real time properties, and further to accommodate other properties that come to light in our case studies. (3) Support its use through domain-specific tools. We plan to make the reasoning apparatus available to other by providing a set of “canned” properties useful to engineers working in robotics and automotive control systems, and support the checking of those properties using automated tools. We have already begun developing a model checker for a subset of the theory, and believe that we can extend this work to handle many of the kinds of properties that come up in practice.

3.2 Synchronization Violations

Ed: this is where the section on synchronization skeletons goes. The following is from the preproposal.

Suppose a designer wants to use model checking to verify whether a distributed implementation of an autonomous or embedded system satisfies a synchronization-related property such as absence of deadlocks. To perform the verification task the designer first must construct a finite-state model of the system. The step of extracting a finite-state model from the implementation is currently performed manually.

By focusing on just the process synchronization aspect of the system, we hope to be able to extract this model automatically. We call this finite-state abstraction of the system its *synchronization skeleton* [5]. We want to investigate techniques based on static analysis and abstract interpretation for constructing synchronization skeletons from a description of the system in a conventional programming language such as concurrent C++ or Java. In many cases an infinite state system will have a finite-state synchronization skeleton because process synchronization is often independent of the data structures used in the program (e.g., the synchronization required for a message queue does not usually depend on the content of the messages).

Some work on deriving synchronization skeletons exists, but we want to develop techniques for deriving synchronization skeletons that are "property-driven." For example, suppose we are interested in proving that a resource controller does not cause a race condition. We will develop techniques that only construct the skeleton of the system relevant to the specific property of the resource controller under consideration (i.e., the race condition). Our preliminary investigation reveals that both deriving the synchronization skeleton and verifying it become easier if the techniques and the underlying algorithms are driven by the property of interest. Moreover, counterexamples discovered in synchronization skeletons derived in this manner are substantially easier to explain and relate back to the original program; this closer connection to code will make it easier to automate the "explanation" of the validation step discussed in the section below.

3.3 Making Tools Useful for the Domain Expert

To make the techniques described above accessible to the engineers who develop autonomous and embedded systems, we need to provide ways for them to easily express properties to be verified, and to understand counter-examples produced by the model checkers. We propose to develop special-purpose specification languages that would make it easy to specify invariant and temporal properties of a system related to its decisional and real-time characteristics. In prior work, we extended the Livingstone model-based fault diagnosis language [29] to include high-level properties of interest to developers, including completeness and consistency of component models and reachability of states [28, 23]. These properties were automatically expanded from the Livingstone language syntax to equivalent temporal logic formulae that were then input to the SMV model checker. We propose to develop similar types of specifications for properties of interest in detecting ordering and concurrency. For instance, one might want to specify that only one active behavior at a time should ever try to access the robot's camera, or that when a given task is terminated it leaves the world in a known state before it completes. In order to create a useful and comprehensive set of such properties, we need a fundamental understanding of how autonomous and embedded systems typically fail. In this, our long history of work in autonomous and embedded systems will be critical [18, 25, 27, 19, 9, ?].

Detecting bugs is only half the battle – an engineer still needs to understand *why* the bug occurs. To help engineers in the process of diagnosing the cause of bugs, we propose to develop techniques to generate textual explanations for counter-examples. Since this is very difficult, in general, we will focus on explaining the types of predefined properties described above. In particular, we will use semantic knowledge of what the properties represent to produce specialized explanation techniques tuned to the different properties.

For instance, we are currently developing "explanation" techniques for the reachability property that we added to the Livingstone system [28, 23]. In Livingstone, reachability means that there is a series of mode transitions that lead from the initial state of the system to some "desired" mode of a particular component. The associated CTL formula template is "EG (component.mode = desired)." **Reid: please check formula. Something got lost in my latexing.** While existential counter-examples are very difficult to explain, in general, we use our understanding of how Livingstone works to develop a reasonably effective explanation algorithm. In particular, we note that the mode of a component may not be reachable if (a) the mode itself is inconsistent, (b) there is no consistent transition to that mode, (c) all modes that could transition to the desired mode are inconsistent, or (d) no mode that could transition to the desired mode is itself reachable. These criteria suggest a search-based algorithm: Check if the mode or all potential transitions to it are inconsistent and, if not, recursively apply this to the modes that define transitions to the desired mode. To implement this, we first convert the SMV model to clausal form and instantiate it into a Truth Maintenance System (TMS) [21]. We then assert the desired mode and use the TMS to determine if the mode is inconsistent. If not, we analyze the original Livingstone model to find all potential transitions, check each for consistency, and then recursively check all the modes that are associated with those transitions until we find an inconsistency. At this point, starting with the inconsistent mode/transition, we trace back through the dependencies recorded by the TMS to essentially create a causal explanation for why the desired mode is not reachable. We anticipate that, due to

their constrained nature, similar strategies will work for many of the other types of predefined properties that will be developed during the proposed research.

4 Modeling the Environment: The Problem and Proposed Research

Autonomous and embedded dynamic systems operate in the physical world, which is governed by the laws of continuous dynamics. These environments are usually modeled with differential and algebraic equations. In contrast, tools for embedded system design and verification 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 in this part of our project is to create tools for analyzing models of physical dynamic systems to obtain behavioral attributes crucial to the verification of the embedded computing system.

4.1 Handling Both Continuous and Discrete Time

Embedded systems interacting with continuous dynamic environments are 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 robotic and mechanism dynamics depending on whether or not certain surfaces are in contact. Recently, there has been considerable interest in the modeling and analysis of hybrid dynamic systems among researchers in both control theory and computer science. Central issues in hybrid dynamic systems research 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.

This latter research on formal verification is most germane to the present proposal. Theoretical research has established clear boundaries between problems that are tractable, or even decidable, and those for which no algorithm exists that can guarantee an answer to verification questions. In contrast to model checking for finite-state systems, algorithmic verification of hybrid dynamic systems is not possible for even very simple classes of dynamics. Consequently, the best one can hope for is the verification of properties of conservative models that represent outer or inner approximations to the families of exact system behaviors.

Based on our experience and the experience of others in building tools to verify properties of hybrid dynamic systems, we propose to take a new approach to these problems in the project. Rather than attempt to extend the verification tools for embedded systems to incorporate continuous dynamics directly, we propose to use the differential equation models to generate discrete models of the environment that can be composed as needed with the existing discrete models of the computing system. This approach has been taken in the past, but the discrete models have been created manually and in an ad hoc manner. We believe the tools for hybrid system modeling and analysis can be used to generate discrete models automatically for which it can be assured that all the important behaviors of the environment have been captured correctly.

There are several research issues that need to be addressed in this approach to verifying autonomous and embedded systems that interact with continuous dynamic environments. Appropriate formal representations need to be developed to define the discrete abstractions and compositions with the models of computation. Our work on discrete-state approximate quotient transition systems will form the basis for these models. To address problems of control, it is necessary that these models capture correctly the input-output behavior of the physical dynamics. Error properties of the numerical routines for generating the models need to be analyzed so that the conservativeness of the models can be guaranteed. The key to this approach will be the ability to select the appropriate scope and dimension 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.

4.2 Probabilistic Models of the Environment

We can model uncertainty of sensors and the environment using stochastic variables or abstraction. For example, suppose a temperature sensor reports that the environment temperature is T degrees; then we can assume that the true

temperature of the environment is a uniform random variable in the range $[T-\epsilon, T+\epsilon]$, where ϵ is related to the imprecision of the sensor. Alternatively, we can use abstraction and nondeterminism to model uncertainty. For example, we can abstractly represent readings from a temperature sensor as symbolic values such as cold, cool, warm, or hot.

We plan to investigate new languages to express real-time probabilistic properties such as “With 90% probability a certain event is handled within 10 milliseconds.” [1]. However, given that in practice numerical values for probabilities are hard to estimate, we also will investigate extending verification techniques such as model checking to support symbolic values for probabilities. We have already obtained some preliminary results incorporating both approaches [16], resulting in a hybrid state machine that has a mix of probabilistic and nondeterministic state transitions.

While there is a vast amount of literature on the verifying probabilistic systems, most results are largely theoretical in nature and use the “Markov assumption,” i.e., the probability of an event depends only on the current state of the system and is independent of its history. From our experience with autonomous and embedded systems, the Markov assumption is not always valid; in such complex distributed networks of nodes events are related or dependent. 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. We plan to identify the practical and realistic environmental conditions under which the Markov model would apply; more ambitiously, we propose to develop fundamentally new models to incorporate dependent events.

5 Educational Activities

Despite many years of research and some remarkable successes, formal specification and verification has had limited impact on the practice of developing and verifying computer systems. We believe part of this failure is because we have failed to demonstrate the value of formal verification to the students and faculty involved in system development. Historically, the university community has provided a supply of “early adopters” for new technologies such as programming languages (e.g., Lisp, TCL), operating systems (e.g., Unix), computer-aided design (e.g., Spice, Magic), and networked services (e.g., X Windows, WWW). These university-based user communities have been important for refining the technology and for making other potential users aware of the new possibilities.

In the case of formal verification, the early adopters have been large corporations such as IBM, Intel, and Motorola. These companies have developed in-house tools, based on university-developed programs such as SMV. They have created large support groups and invested significant resources in putting these tools into practice. By contrast, our faculty colleagues are largely unaware of the capabilities of formal verification. University courses on computer architecture, operating systems, distributed systems, and software engineering still present *ad hoc* simulation and testing as the main approaches to verifying systems. When these students take jobs at companies with in-house verification tools, they must be trained in their use. When they go to other companies, they remain ignorant of the possibilities for more rigorous approaches to verification.

We believe that developing course materials that can be incorporated into existing system design courses could have significant impact on the adoption of formal verification by industry and academia. Using the controlled environment of course projects will allow us to demonstrate the utility of formal specification and verification in detecting design errors and correcting them. This material will reach an important strategic audience—the students learning to design systems and the faculty instructing them. The students will then be prepared to have impact on industrial practice once they graduate. The faculty will see the benefits of formal verification and apply it to their research projects. This will lead to the university-based user community that can provide a driving force for future innovations and improvements.

5.1 Current Practice at Carnegie Mellon

We already have some experience in using formal specification and analysis tools in graduate-level courses at the Master’s and Ph.D. levels. This experience makes us optimistic that we can easily and realistically do more.

In our Professional Master’s of Software Engineering Program two of the five required core courses cover formal specification and analysis of software: Models of Software Systems and Analysis of Software Artifacts. Students in this program have worked in industry as a software engineer for at least two years. In the Models course, we teach basics of state machines and concurrent processes. We cover different specification notations, including Z, Statecharts, CSP, temporal logic, and Petri Nets, with the focus on understanding mathematical structures and proof techniques that are needed for modeling and reasoning about large systems. Homework assignments require writing formal specifications of varying degrees of complexity; examples include specifying the course’s grading system, an elevator

controller, our department's M&M dispensing machine, and a coed bathroom protocol. In the Analysis course, students use tools, including SMV, FDR, and Nitpick, to specify and analyze their models. Example homework assignments include verifying properties of the Coda file system cache coherency protocol, the alternating bit protocol, a simple database transaction service, a simple telephone service, and functions from the C string library. The emphasis of the course is to give students practice in identifying abstractions that are "small" (describable by a succinct specification), but rich enough to allow them to check a "deep" property of interest. Both of these courses have been taught since the 1993-94 academic year.

Testimonials from former students provide evidence that some of our greatest skeptics have become our greatest advocates. Essentially they report that they use formalism as a matter of course. In one example, after the student returned to his company, during a design review he wrote Z specifications on the whiteboard to clarify points of debate, and only later did it hit him that he was writing a formal specification. As another example, students applied formalisms they had learned in the Models course to a project assignment in a subsequent elective course, not because they were told to but because they felt it was the best way to communicate their ideas with each other.

We also have had some experience introducing model checking in a Ph.D.-level graduate course on computer architecture in the context of snoopy bus cache coherency protocols [15]. We found that two lectures plus one assignment was sufficient to show how to encode a protocol in the SMV language, how to specify desired properties in temporal logic, and how to use the tool to debug and verify these protocols. Encoding the protocol in SMV also served to present the protocol in a more concrete fashion than the state diagrams and descriptive text seen in the textbook. Students were able to run the model checker and get insightful results without great difficulty. They could clearly see the advantage of a tool that would analyze all possible executions of the protocol, rather than trying to hand generate a set of comprehensive tests.

Similar material could be developed for other design courses. For example, a digital logic design course could benefit from the use of both symbolic simulators [?] and word-level model checkers [4]. Standard design projects such as ALUs and simple state machines could easily be verified. When synchronization constructs, such as semaphores, are taught in courses on concurrency or operating systems, it would very natural to introduce model checkers and temporal logic specifications. Any outcomes of the research from this proposal could be used in courses on real-time and embedded systems.

5.2 What We Propose To Do

In our experience, people are much more convinced of the utility of formal verification once they have actually tried it. They find bugs that had escaped notice despite hours of scrutiny and testing. After eliminating these bugs, they gain a sense of assurance that cannot be provided by simulation and testing. Giving students a taste of formal verification as part of their normal design projects, we can hope to interest them in learning more about verification tools and the underlying technology.

Creating this educational capability more broadly from the undergraduate level through the Ph.D. level requires conscious developmental effort in tools and teaching materials. First, we must have a set of tools that fit into existing course flows. For example, most digital design classes in the United States use tools based on the Verilog hardware description language. Although several industrial model checkers accept Verilog input, its support by public domain tools is inadequate. One possibility would be to arrange to make a commercial tool available for licensing at educational institutions, much as Synopsys has for logic design tools. Our experience with commercial tools, however, is that they are generally too "heavyweight" for this sort of context. They are designed to be used for daily use by experts, rather than occasional use by novices. Consequently, they come with thousands of pages of documentation and contain a confusing array of features. In addition, the logistics of installation and license management would be excessive for a tool that will only be used for a few weeks during the entire course.

We propose developing lighter weight, public domain tools for educational use. As part of our ongoing research in formal verification, we are currently developing an integrated verification "platform" consisting of several underlying computational "engines" with a high-level language on top for constructing applications. Particular engines would include binary decision diagrams for symbolic Boolean manipulation, and basic decision procedures to enable limited forms of automatic theorem proving. This platform would be similar to the Voss system developed at the University of British Columbia [14, 22]. (Unfortunately, UBC has been unwilling to put Voss in the public domain, and hence we must recreate this capability.) Sharing this programming effort between our research work and our educational initiatives would be beneficial to both efforts.

For teaching materials, the main task is developing a set of lecture notes and accompanying assignments and documentation. For our earlier work on model checking of cache coherency protocols, the main effort would be to develop enough supporting materials so that it could be presented by an instructor who is not already an expert in model checking. For the other courses, we would attempt to identify instructors who will collaborate in identifying appropriate examples and assignments. Some of the instructors for a digital logic design course at Carnegie Mellon have already expressed interest in using formal verification tools.

We have very specific plans for enhancing the Master's of Software Engineering program. While we have had success in teaching formal methods in courses, we have yet to integrate their use by students into the program's hallmark *Studio Project*. We are asking for support for one MSE student to help with this integration effort.

We would also like to investigate how to integrate formal techniques with less formal ones, e.g., UML and message sequence charts, that are more familiar to practicing software engineers. These notations are taught in another one of the five required MSE courses so we have the opportunity to try out our ideas directly with software engineering veterans.

Finally, we would like to find ways to incorporate formal specification and verification techniques into the undergraduate computer science courses without perturbing the overall curriculum. The novelty behind our plan is not (only) to offer special electives to advanced undergraduates but to introduce formal methods in existing early courses and revisit them in existing later courses. There are opportunities for introduction in lower-level programming, data structures and algorithms, and programming methodology courses. There are opportunities for reinforcement in upper-level programming languages, compilers, operating systems, databases, and software engineering courses. Our goal is to obviate the need to change the overall undergraduate computer science curriculum, but only to enhance existing courses.

6 Cross-Cutting Themes and Summary of Proposed Work

For our proposed work, several unifying themes help us focus our efforts and differentiate our work from that of others.

- *Automatic, tool-based approach.* We believe in a "tool-based" approach to formal verification, where highly automated verification programs do most of the actual work. As an example, model checkers evaluate system behavior over many different operating conditions to determine whether a particular property is satisfied. Model checkers have proved successful for reasoning about highly concurrent systems and for allowing useful properties to be verified even in cases where a complete system specification is not available. Such capabilities can be extended to more abstract domains through the use of automatic decision procedures and symbolic encoding techniques.
- *Interdisciplinary team.* Our research group consists of faculty with a range of backgrounds, including automotive control, robotic control, formal hardware verification, and software specification and analysis. The six co-PIs are faculty collectively with appointments in the Computer Science Department, the Electrical and Computer Engineering Department, and the Robotics Institute at Carnegie Mellon University. With this team we combine expertise in the application domains, in the underlying computing platform, and in the software development process. We also have a great deal of collective experience in developing and applying formal verification tools.
- *Strong educational mission.* We believe that education is one of the most effective methods to transfer the research results in formal methods into regular practice. A significant portion of our funding will be directed toward graduate student support as well as developing course materials.

Formal specification and verification methods can complement traditional testing and simulation in the design and evaluation of autonomous and embedded systems. Our research is to investigate how. We plan to develop brand new methods that can apply to this growing and pervasive class of systems. Our expectation is that our results will more broadly apply to concurrent and distributed systems in general since many of our contributions-in the form of new mathematical models, logics, languages, algorithms, and data structures-will be fundamental in nature.

References

- [1] C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In *24th International Colloquium on Automata, Languages, and Programming (ICALP '97)*, volume LNCS 1256. Springer-Verlag, July 1997.
- [2] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. A proven three-tiered architecture for programming autonomous robots. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2), 1997.
- [3] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1), 1986.
- [4] Y.-A. Chen, E. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao. Verification of all circuits in a floating point unit using word-level model checking. In *Formal Methods in Computer-Aided Design*, volume LNCS 1166. Springer-Verlag, November 1996.
- [5] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs Workshop*, volume 131. Springer-Verlag, May 1981.
- [6] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *Computing Surveys*, 24(4):626–643, December 1996.
- [7] R. James Firby. An investigation into reactive planning in complex domains. In *Proc. National Conference on Artificial Intelligence*, pages 202–206, Seattle, WA, July 1987.
- [8] R. James Firby. Building symbolic primitives with continuous control routines. In *AI Planning Systems (AIPS-92)*, College Park, MD, June 1992.
- [9] Ed Gamble and Reid Simmons. Software architectures for spacecraft autonomy. *IEEE Intelligent Systems*, 13(5), 1998.
- [10] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proc. National Conference on Artificial Intelligence*, pages 809–815, San Jose, CA, July 1992.
- [11] Erann Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. In Louise Pryor, editor, *Procs. of the AAAI Fall Symposium on Plan Execution*. AAAI Press, 1996.
- [12] Mike Georgeff and Amy Lansky. Reactive reasoning and planning. In *Proc. National Conference on Artificial Intelligence*, pages 972–978, Seattle, WA, July 1987.
- [13] David Harel. Statecharts: A visual approach to complex systems. Technical Report CS84-05, Weizmann Institute of Science, February 1984.
- [14] S. Hazelhurst and C.-J. H. Seger. A simple theorem prover based on symbolic trajectory evaluation. 14(4), April 1995.
- [15] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, (Second Edition)*, chapter 8. Morgan-Kaufmann, San Francisco, 1996.
- [16] S. Jha, R. Linger, T. Longstaff, and J. Wing. Analyzing survivability properties of specifications of networks. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2000. To appear.
- [17] Leslie Pack Kaelbling. An architecture for intelligent reactive systems. In M. Georgeff and A. Lansky, editors, *Reasoning About Actions and Plans*. Morgan Kaufmann, 1987.
- [18] Eric Krotkov and Reid Simmons. Perception, planning, and control for autonomous walking with the ambler planetary rover. *International Journal of Robotics Research*, 15(2):155–180, April 1996.

- [19] Stewart Moorehead, Reid Simmons, and William Whittaker. Autonomous navigation field results of a planetary analog robot in antarctica. In *Proceedings 1999 International Symposium on Artificial Intelligence, Robotics and Automation in Space*, Noordwijk, Holland, June 1999.
- [20] David Musliner, Edmund Durfee, and Kang Shin. CIRCA: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), 1993.
- [21] Pandu Nayak and Brian Williams. Fast context switching in real-time propositional reasoning. In *Proceedings National Conference on Artificial Intelligence*, pages 50–56, Providence RI, July 1997. AAAI.
- [22] J. O’Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying ieeec compliance of floating-point hardware. *Intel Technology Journal*, 1999.
- [23] Charles Pecheur and Reid Simmons. From livingstone to smv: Formal verification for autonomous systems. In *Goddard Workshop on Formal Methods*, April 2000.
- [24] S. Schneider, V. Chen, G. Pardo-Castellote, and C. Wang. Controlshell: A software architecture for complex electromechanical systems. *The International Journal of Robotics Research*, 14(4), April 1998.
- [25] Reid Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, February 1994.
- [26] Reid Simmons and David Apfelbaum. A task description language for robot control. In *Proc. International Conference on Intelligent Robots and Systems*, Vancouver Canada, October 1998.
- [27] Reid Simmons, Rich Goodwin, Karen Zita Haigh, Sven Koenig, and Joseph O’Sullivan. A layered architecture for office delivery robots. In W. Lewis Johnson, editor, *Proc. Autonomous Agents ’97*, pages 245–252, Marina del Rey, CA, February 1997. ACM Press.
- [28] Reid Simmons and Charles Pecheur. Automating model checking for autonomous systems. In *AAAI Spring Symposium on Real-Time Autonomous Systems*, Stanford, CA, March 2000.
- [29] Brian Williams and Pandu Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings National Conference on Artificial Intelligence*, pages 971–978, Portland OR, August 1996. AAAI.