Towards Automatic Verification of Autonomous Systems

Reid Simmons
Carnegie Mellon University
Pittsburgh, PA 15213
reids@cs.cmu.edu

Charles Pecheur
RIACS/NASA Ames Research Center
Moffett Field, CA 94035
pecheur@ptolemy.arc.nasa.gov

Grama Srinivasan
Wipro Technologies
Bangalore, India
srinivasan.grama@wipro.com

Abstract

While autonomous systems offer great promise in terms of capability and flexibility, their reliability is particularly hard to assess. This paper describes research to apply formal verification methods to languages used to develop autonomy software. In particular, we describe tools that automatically convert autonomy software into formal models that are then verified using model checking. This approach has been applied to MPL code for the Livingstone fault diagnosis system and to TDL task descriptions for mobile robot systems. Our long-term objective is to create tools that enable engineers and roboticists to use formal verification as part of the normal software development cycle.

1 Introduction

Autonomous systems rely on intelligent inference capabilities to be able to take appropriate actions even in unforeseen circumstances. They enable a whole range of new applications, such as sending autonomous robots to places where it is too dangerous or expensive for humans to go, and where remote human control is difficult or not even technically feasible. In particular, autonomy is a key enabling technology for the future of NASA's space exploration program and is becoming more important in terrestrial applications as embedded systems and mobile robots become more prevalent.

However, this increased capability and flexibility comes with a price: It is typically very difficult to assess the reliability of autonomy software. Traditional scenario-based testing methods fall short of providing the desired confidence level, mainly due to the combinatorial explosion of possible situations to be analyzed. Formal verification is a powerful tool for creating reliable systems. Model checking is one technique that has been used successfully to formally verify complex hardware and software systems [2, 4]. In model checking, one specifies the system in a formal language, such as SMV [2] or PROMELA [6], along with formal specifications that indicate desirable properties of the system that one wants verified. The model checker then determines whether the properties hold under all possible execution traces. Essentially, model-checking exhaustively, but intelligently, considers the complete execution tree. Counterexamples are provided for specifications that do not hold.

Our long-term objective is to make formal verification, and model checking in particular, part of the standard tool kit for designing and developing autonomous systems. The idea is to make model checking easy enough to use so that it can be employed as a regular part of the development/debugging cycle, much as compilers regularly employ extensive syntactic checking before producing object code. The hope is that by checking early in the development cycle, subsequent testing and debugging can be significantly reduced.

A large obstacle to this vision is computational complexity. Although great strides are being made in model-checking algorithms, it is still the case that verifying large software systems written using general-purpose programming languages are beyond the state of the art. Our approach to this problem is based on the fact that many autonomous systems are developed using special-purpose languages and inference engines. For instance, the NASA Remote Agent [7] uses specialized languages for each of the planner, executive, and fault diagnosis components. The advantage here is that these languages are typically at a higher level of abstraction and are typically more constrained than general-purpose programming languages. In general, simpler languages are easier to verify.

Given this approach, we can identify three classes of verification problems:

- Special-purpose languages use special-purpose interpreters (or compilers). These need to be verified.
- Application-specific programs written in these languages need to be verified for *internal* correctness. This includes checking for liveness, safety, etc.
- Application-specific programs need to be verified for *external* correctness. That is, determining whether the interaction of the software with the environment meets system requirements.

In this work, we focus only on the second class of problems. While the first class is important, verification of the correctness of an interpreter can be done once by its designers. From the point of view of engineers, the interpreter is viewed as a stable, trustable tool, just as programmers trust their compiler. While the third class is also very important, modeling the environment is very complex, and is not at all well understood at this point.

Thus, the basic approach described in this paper is to automate the translation of programs, written in special-purpose languages, into the SMV model-checking language, perform model checking using standard algorithms, and then translate counterexamples back into terms that are meaningful to the software developer. In on-going work, we are developing tools for visualizing and explaining the counterexamples.

The next section briefly describes the SMV model checker and the two languages that we used as case studies. Sections 3 and 4 describe the translators developed for those languages, and some of our early results. Section 5 presents our on-going work in this area.

2 Background

Symbolic Model Checking: Model checking is a verification technology based on the exhaustive exploration of a system's achievable states. Given a model of a concurrent system, and an desired property of that system, a model checker will examine all possible executions of that system, including all possible interleavings of concurrent threads, and report any execution that leads to a violation of that property. The report is typically a *counterexample* to the property.

Classical, *explicit-state* model checkers such as SPIN [6] do this by generating and exploring every single state. In contrast, *symbolic* model checkers such as SMV [2] manipulate whole sets of states at once, implicitly represented as the logical conditions that those states satisfy. These conditions are often encoded in data structures called Binary Decision Diagrams (BDDs) [1], which provide a compact representation and support very efficient manipulations of Boolean formulae. While symbolic model checking has traditionally been applied to hardware systems, it is increasingly being used to verify software systems, as well.

Model-based Processing Language: MPL is used in the Livingstone system to encode hardware and software models. Livingstone is a model-based fault diagnosis and recovery subsystem developed by NASA [17]. The Livingstone inference engine observes a physical system, predicts its current state according to the MPL models, detects discrepancies between the predicted and observed states, and diagnoses potential faults if discrepancies exist. Livingstone was initially developed for the Remote Agent, an autonomous spacecraft controller demonstrated in flight on the Deep Space One (DS1) mission [7]. In addition, Livingstone is being applied to the control of a propellant plant for Mars missions [3] and execution monitoring for the Xavier mobile robot [13].

```
(defvalues on-off-values (on off))
(defvalues on-off-commands (on off no-command))

(defcomponent switch (?name)
  (:inputs ((command-in ?name) :type on-off-commands))
  (:outputs ((indicator-lamp ?name) :type on-off-values))
  (:background :initial-mode off)
  (on  :model (on (indicator-lamp ?name)) :type ok-mode
       :transitions((turn-off
                       :when (off (command-in ?name)
                       :next off)
                    (:otherwise :persist)))
  (off :model (off (indicator-lamp ?name)) :type ok-mode
       :transitions((turn-on
                       :when (on (command-in ?name)
                       :next on)
                    (:otherwise :persist)))
  (broken  :type :fault-mode :probability 0.01
           :transitions ((:otherwise :persist))))
```

Figure 1: MPL Model of a Simple Switch

MPL systems consist of sets of *components* and *modules*. Figure 1 shows a simple component written in MPL. MPL *components* consist of a set of parameters, a set of state variables (including *inputs* and *outputs*), and a set of modes, some of which are labeled as *fault modes.* Each mode has a set of guarded transitions (including self-transitions), where the guards are expected to be mutually exclusive and exhaustive. The guards are first-order formulae that involve state variables. For instance, in Figure 1, the *on* mode transitions to the *off* mode when the *command-in* variable is *on.*

Modes also consist of formulae that describe the values of state variables when in that mode. For instance, the model in Figure 1 states that when the *switch* is in the *on* mode, the *indicator-lamp* is also *on*. MPL *modules* consist of collections of components and other modules, plus constraints between them. For instance, a module can constrain the power output variable of one component to be equal to the power input of another component.

Task Description Language: TDL is an extension of C++ that includes constructs for hierarchical task decomposition and synchronization between tasks, monitoring task execution, and handling exceptions [15]. Figure 2 shows two simple tasks for multi-robot coordination written in TDL. TDL simplifies the process of specifying how concurrent robot tasks should, and should not, behave and interact. TDL, which is based on the TCA (Task Control Architecture) [12], has been used to implement the "executive" layer of several autonomous mobile robot systems (e.g., [16]). While TDL, being an extension of

```
Goal GroupDeploy (DEPLOY_PTR deployList)
{
  with (serial) {
    for (int i=0; i<length(deployList); i++) {
       spawn GroupDeploySub(i, deployList);
    }
  }
}
Goal GroupDeploySub (int phase,
                        DEPLOY_PTR deployList)
{
  with (parallel) {
    for (int j=phase; j<length(deployList); j++) {
       spawn Deploy(deployList[j].robot,
                        deployList[phase].location);
    }
  }
}
```

Figure 2: TDL Specification of a Multi-Robot Deployment Strategy (Simplified)

```
MODULE switch
VAR command-in : {on_, off_, no-command_};
    indicator-lamp : {on_, off_};
    _mode : {on_, off_, broken_};
DEFINE _faults := {broken_};
       _broken := (_mode in _fault_modes);
INIT (_mode = off_)
TRANS (((_mode = on_) & (command-in = off_))
    -> (next(_mode) in (off_ union _faults))
TRANS (((_mode = on_) & !(command-in = off_)
    -> (next(_mode) in (on_ union _faults)))
TRANS (((_mode = off_) & (command-in = on_))
    -> (next(_mode) in (on_ union _faults)))
TRANS (((_mode = off_) & !(command-in = on_)
    -> (next(_mode) in (off_ union _faults))
TRANS ((_mode = broken_) ->
       (next(_mode) = broken_))
INVAR ((_mode = on_) ->
       (indicator-lamp = on_))
INVAR ((_mode = off_) ->
       (indicator-lamp = off_))
```

Figure 3: SMV Translation of Switch Model

C++, is a general-purpose programming language, here we deal with verifying just the parts of TDL that are concerned with task decomposition and synchronization.

3 Translating MPL to SMV

The first step in translating a language to a formal representation, such as SMV, is to model the semantics of the language. In the case of MPL, this means formalizing how the Livingstone engine infers modes and state transitions. A first-order formalization is fairly straightforward, since MPL is based on synchronous, concurrent transition networks, which is the same formalism underlying SMV. Unfortunately, understanding exactly how Livingstone works is difficult, and research at NASA Ames is endeavoring to capture this precisely in a formal way.

In general, a translator is developed that converts code in one language to an SMV model that incorporates the semantics of the formal model of the interpreter. For MPL, the fact that Livingstone and SMV have similar semantics means that the original MPL code and the transformed SMV models have similar syntactic properties. In addition to the explicit state variables in MPL, we add a new variable to each component that represents the mode of the component. Each guarded transition in MPL becomes a transition statement (TRANS) in SMV. For instance, Figure 3 shows the SMV translation of the MPL code in Figure 1. The translator is described in more detail in [10].

The main difficulty in performing the translation comes from differences in variable naming conventions between the flat, global name space of MPL and the hierarchical name space of SMV. For instance, in MPL one could directly refer to "(indicator-lamp switch2)" from within a higher-level module. In SMV, however, one would need to address that variable hierarchically as "widget1.switch-module.switch2.indicator-lamp". To handle this, the translator constructs a *lexicon* that indicates how to translate MPL names to SMV names. The lexicon is generated by starting at the root ("main") module and walking the tree of modules and components, expanding the SMV names as necessary when a subpart is encountered. The translator uses a simple hash table to find corresponding names. In addition, the lexicon is used in the inverse direction to translate SMV counterexamples back into terms of the original MPL code, in order to make the counterexamples more understandable to engineers.

Another difficulty in this enterprise is how to specify the properties to be verified. SMV supports the powerful temporal logic CTL (Computation Tree Logic) to express such properties [4]. CTL enables specifications that can quantify over time and over all (or some) of the execution paths that a system can take. For example, *AG (flow=high)* states that for all execution paths, globally (in each state along the path) the flow is high. We have extended the MPL syntax to enable users to encode CTL formulae directly in MPL, using a Lisp-like style that is consistent with the rest of the MPL syntax.

Unfortunately, while CTL is expressive, it is also difficult for novices to use correctly. As a simpler alternative, the translator supports pre-defined specification patterns for common properties, such as consistency, completeness, and reachability of given component modes. Consistency and completeness check whether the guarded transitions are mutually exclusive and exhaustive. Reachability checks whether a legal path of mode transitions exist from the initial state to some given mode. These pre-defined properties are automatically translated into equivalent CTL formulae, which can then be verified by SMV.

The translator also provides auxiliary predicates that can be used to test for various important characteristics of a system. For instance, there are predefined predicates representing transitions to fault modes, presence of failed components, and occurrences of commands to the system. The probability of a component entering a given fault mode is captured using predicates that encode order-of-magnitude probabilities (likely, unlikely, etc.). These predefined predicates make it easier to write concise CTL specifications.

We have applied the MPL translator to several real-world applications that use Livingstone. The most extensive application, and the one for which Livingstone was initially developed, is the diagnosis and recovery component of the Remote Agent architecture on the DS1 spacecraft. The full MPL program for the spacecraft runs to several thousand lines of code. Using the translator, we have automatically constructed SMV models and verified several important properties, including consistency and completeness of the mode transition relations, and reachability of each mode. Using the translator, we were able to identify several (minor) bugs in the DS1 models, after the models had been extensively tested by more traditional means [8].

The translator was also applied to MPL programs describing the locomotion system of both an indoor and outdoor mobile robot. Those programs, which had not been extensively tested, were found to contain several significant bugs, including transition guards that were inconsistent, mode transitions that were incomplete and not mutually exhaustive, inconsistent constraints on state variables, and non-reachable modes. The latter stemmed from several factors, including modes in one component being inconsistent with all modes of another component, guards on transitions never becoming true, and the constraints associated with certain modes being inconsistent.

Finally, the translator is being used at NASA Kennedy Space Center by the developers of a Livingstone model for the In-Situ Propellant Production (ISPP) system, which is designed to produce spacecraft propellant using the atmosphere of Mars [3]. Early experiments have shown that SMV can easily process the ISPP models (translation and verification takes just several seconds) to verify several useful properties, such as reachability of normal operating conditions and recoverability from failures.

4 Translating TDL to SMV

As described in the previous section, the first step in creating a translator is to formalize the semantics of the language. Currently, we are concerned only with the task decomposition and task synchronization aspects of TDL. A *task* in TDL is a piece of C++ code that can operate on the environment and/or spawn subtasks. TDL tasks are synchronized using a language of relational and metric temporal constraints. The constraints can refer to various *aspects* of the task:

- The *handling* of a task refers to the time it takes to execute the C++ code associated with the task.
- The *expansion* of a task refers to the time it takes to completely expand the TDL subtree rooted at that task, that is, the handling of all non-leaf-node tasks.
- The *execution* of a task refers to the time it takes to handle all leaf-node tasks.

Each aspect of a task is formalized as a finite state machine with four states: *disabled*, *enabled*, *active* and *completed*. Not all 64 combinations of the three aspects are legal, however. There are intra-node constraints and inter-node constraints that reduce the number of legal transitions. The intra-node constraints represent things such as the handling of a task cannot become enabled until its expansion is enabled. Figure 4 shows the intra-node constraints as a state-transition diagram for a single TDL task.

Inter-node constraints connect parent, children and sibling tasks. For instance, in TDL the execution of a child task cannot be active until the execution of the parent is enabled. Similarly, the execution of the parent task cannot be completed until all the children tasks have their execution completed. In addition, developers can explicitly add temporal constraints between sibling tasks, such as saying that one task must be executed sequentially after another, or that one task is enabled when another becomes active.

Each TDL task is represented as an SMV module with separate state variables for the handling, expansion and execution aspects. The non-leaf-node tasks also have variables representing children tasks and encode constraints detailing the synchronization relationships between parent, children and sibling tasks. We have constructed both synchronous and asynchronous versions of the SMV models. While the asynchronous version is simpler, it tends to take longer to verify. Since we are not modeling metric time, both the synchronous and asynchronous versions utilize *fairness* constraints in SMV to model that the handling time of a task can be arbitrarily long, but finite.

The way TDL is used, a compiler translates TDL code into pure C++ code, which includes calls to a task management library [15]. A command line option enables the same compiler to produce an intermediate representation of just the task decomposition and task synchronization constraints. This intermediate language is what the TDL translator actually uses to produce SMV models.

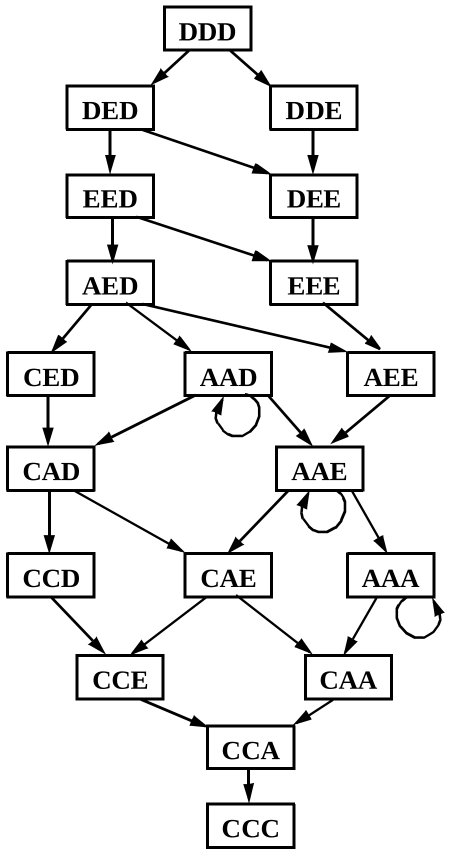

Figure 4: State Transitions Within a TDL Task
Each node is labeled with the status of the *handling*, *expansion* and *execution* aspects, in that order, with status being one of **D**isabled, **E**nabled, **A**ctive, or **C**ompleted. Thus, **CAE** represents that the handling is completed, the expansion is active, and the execution is enabled.

Currently, we translate only a subset of TDL. While we can handle all of the qualitative temporal constraints between tasks (e.g., “the expansion of task A precedes task B”), we do not handle metric constraints (e.g., “the handling of task C begins in 10 seconds”). While we can handle both unconditional and conditional spawning of tasks, we cannot handle iterative constructs (spawning of tasks within a “for” or “while” loops) or recursive invocation of tasks, all of which are supported in TDL.

While handling metric constraints is, in principle, relatively straightforward - one can discretize time and implement a global “clock” - this simple approach typically blows up the state space of the verification problem. More research is needed to find efficient means of dealing with metric time. Handling task trees with variable numbers of children tasks is also quite difficult. One simple approach is to verify each possible expansion (e.g., for one iteration, for two iterations, etc.). This, however, suffers from severe combinatorial explosion. Another approach is to infer invariants over the loops and abstract away the loop structure [11].

We can verify certain important properties of task-level control programs using the formal models of TDL that our translator generates. In particular, we can check for deadlock, liveness, and absence of resource conflict. The latter is specified by associating resources with tasks and verifying that the tasks that are associated with the same resource cannot have their handling aspect active concurrently. Currently, however, these specifications have to be encoded directly in CTL. We are currently designing extensions to TDL that would enable specifications to be described directly in the language.

Unlike the MPL translator, we have not yet used the TDL translator on real applications. We have, however, tested it on several small examples. Based on these results, we have improved the models produced by the translator to reduce the time needed by the model checker (which dominates the process as a whole).

5 On-Going Work

For MPL, we continue to use the translator in real applications. In addition, we are extending the types of pre-defined specifications, in response to feedback from users. For TDL, we are working to extend the range of TDL constructs that can be handled. In particular, we are working to model metric temporal constraints, iterative constructs, and the fact that one TDL task can terminate another task (which, in turn, terminates all its subtasks, recursively).

Another major focus is dealing with the counterexamples produced by SMV. Counterexamples are essentially symptoms of bugs in the original system. However, it is usually quite difficult to diagnose the error directly from the counterexample. One reason is that the counterexample is in the SMV language, and the translation may not directly preserve connections between the SMV model and the original language. For instance, variable names may be different and certain aspects of the hierarchical structure of the system may be lost during the translation. To handle this, we perform an inverse translation — from the SMV counterexample back to the original high-level language.

A more serious difficulty is that a counterexample merely indicates the state, or sequence of states, that led to the problem, but gives no indication of what within the state, or which particular state transitions, were really responsible. This is essentially a diagnosis problem. We are investigating two approaches to this problem.

First, we can use visualization techniques to abstract counterexamples and present the data in a way in which the cause of the problem may be more apparent. The idea is that many inference engines used in autonomous systems have associated visualization tools that distill and present data from log files of actual runs of the system. To use such tools for visualizing counterexamples, one needs to translate the state transitions in the counterexamples into the log file format that the tool expects. In essence, we are creating a "virtual" execution trace that corresponds to the false specification. In this way, all the features of the visualization tool are available to aid the user in understanding and debugging the problem.

The other way we are addressing the problem of understanding counterexamples is to produce textual explanations of the problem. Our current approach is to take the counterexample and feed it into a TMS (Truth Maintenance System), along with the original SMV model. The TMS [9] propagates the constraints of the model and the state of the system (given by the counterexample). When it detects an inconsistent proposition, it traces back through the links in the TMS. This produces a hierarchical explanation of the inconsistency that can help focus a developer on the true source(s) of the problem.

6 Conclusions

Model checking is a powerful tool for verification of autonomous systems. To date, however, it has had little impact on the way autonomous systems are developed and validated. This is due to both the state explosion problem and the tedious, error-prone process of manually translating software into formal models. We are addressing these problems by focusing on high-level languages that are geared towards autonomous systems and by developing translators that produce SMV code automatically from such languages. We are also extending the original languages to enable desired verification properties to be specified directly, and at a high level of abstraction.

This paper has presented results with two such languages - MPL, used for model-based fault diagnosis and recovery, and TDL, used for task-level control. While the efforts are still on-going, and the results are still preliminary, we have confidence that this approach can significantly impact the development of autonomous systems. The result should be more reliable autonomous systems with reduced debugging/testing effort.

Acknowledgments

This work is sponsored in part by grants from NASA.

References

[1] R. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation." *IEEE Transactions on Computers,* **C-35(8)**, 1986

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond." *Information and Computation,* **98(2)**, pp. 142-170, June 1992.

[3] D. Clancy, W. Larson, C. Pecheur, P. Engrand and C. Goodrich, "Autonomous Control of an In-Situ Propellant Production Plant." In *Proceedings Technology 2009 Conference,* Miami FL, November 1999.

[4] E. M. Clarke , O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] A. R. Gross, K. R. Sridhar, W. E. Larson, D. J. Clancy, C. Pecheur, and G. A. Briggs, "Information Technology and Control Needs For In-Situ Resource Utilization." In *Proceedings of 50th IAF Congress*, Amsterdam, Holland, October 1999.

[6] G.J. Holzmann, "The Model Checker SPIN." *IEEE Transactions on Software Engineering,* **23(5)**, May 1997.

[7] N. Muscettola, P. P. Nayak, B. Pell, and B. Williams. "Remote Agent: To Boldly Go Where No AI System Has Gone Before." *Artificial Intelligence* **103(1-2)**, pp. 5-48, August 1998.

[8] P. P. Nayak, et. al. "Validating the DS1 Remote Agent Experiment." In *Proceedings of the Fifth International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS)*, ESTEC, Noordwijk, 1999.

[9] P. P. Nayak and B.C. Williams. "Fast Context Switching in Real-time Propositional Reasoning." In *Proceedings of National Conference on Artificial Intelligence*, Providence, RI, July 1997.

[10] C. Pecheur and R. Simmons. "From Livingstone to SMV: Formal Verification for Autonomous Spacecraft." In *Proceedings of the First Goddard Workshop on Formal Approaches to Agent-Based Systems*, April 2000.

[11] M. Sagiv, T. Reps and R. Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. *Transactions on Programming Languages and Systems*, **20(1)** pp. 1-50, January 1998.

[12] R. Simmons. "Structured Control for Autonomous Robots." *IEEE Transactions on Robotics and Automation*, **10(1)**, pp. 34-43, February 1994.

[13] R. Simmons, R. Goodwin, K. Zita Haigh, S. Koenig. and J. O'Sullivan. "A Layered Architecture for Office Delivery Robots." In *Proceedings of First International Conference on Autonomous Agents*, Marina del Rey, CA, February 1997.

[14] R. Simmons and G. Whelan. "Visualization Tools for Validating Software of Autonomous Spacecraft." In *Proceedings of International. Symposium on Artificial Intelligence, Robotics and Automation in Space*, Tokyo, Japan, July 1997.

[15] R. Simmons and D. Apfelbaum. "A Task Description Language for Robot Control." In *Proceedings of Conference on Intelligent Robotics and Systems*, Vancouver Canada, 1998.

[16] R. Simmons, D. Apfelbaum, D. Fox, R.P. Goldman, K.Z. Haigh, D.J. Musliner, M. Pelican, S. Thrun. "Coordinated Deployment of Multiple Heterogeneous Robots." In *Proceedings of Conference on Intelligent Robotics and Systems* (this volume), Takamatsu Japan, 2000.

[17] B. C. Williams and P. P. Nayak. "A Model-based Approach to Reactive Self-Configuring Systems." In *Proceedings of the National Conference on Artificial Intelligence*, Portland OR, August 1996.