Automating Model Checking for 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

Abstract

While autonomous systems offer great promise in terms of capability and flexibility, their reliability is particularly hard to assess. This paper describes research in the use of model checking to support the development of reliable autonomy software. In particular, it presents tools and techniques that we are developing to facilitate the integration of model checking into the main software development cycle. The basic approach is to translate high-level models used by autonomy systems into the specification language of the SMV model checker, verify them using SMV, translate diagnostics back to the source language and visualize and explain those diagnostics. This approach has been applied to MPL models for the Livingstone fault diagnosis system and to TDL task descriptions for mobile robot systems.

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 in 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 robotics 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, because of the huge number of scenarios that have to be considered. Formal verification is a powerful tool for creating reliable systems [2, 3]. Model checking is one technique that has been used successfully to formally verify complex hardware and software systems. In model checking, one specifies the system in a formal language, such as SMV or PROMELA, 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. Counter-examples are provided for any specification that does not hold.

While powerful, to date model checking, and other formal verification techniques, have had little impact on the development of autonomous systems. There are several reasons for this. For one, developers of autonomous systems are not typically versed in model-checking techniques. For another, one typically has to manually translate the system into the model-checking language, a process which is both tedious and error prone. Then, the counter-examples produced by the model checker must be interpreted, in the context of the original system, in order to understand what actually is going wrong. Finally, model checking tends to be computationally expensive which, in general, tends to limit the size of software programs that can be verified.

Our goal 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 and some semantic checking before producing object code. The hope is that by checking early in the development cycle, subsequent testing and debugging can be significantly reduced.

The basic approach is to automate the translation of autonomous system software and specifications to the SMV model-checking language, perform model checking using standard algorithms, translate counter-examples back into terms that are meaningful to the software developer, and then provide tools for visualizing and explaining the counter-examples.

Probably the biggest problem is computational complexity. Although great strides are being made in the model-checking algorithms themselves, it is still the case that large software systems written using general-purpose programming languages are beyond the state of the art. Our solution to this problem is based on the fact that many

```
(defcomponent switch (?name)
  (:inputs ((command-in ?name)
            :type on-off-command))
  (: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

autonomous systems are developed using special-purpose languages and inference engines. For instance, the NASA Remote Agent [5] uses specialized languages for each of the planner, executive and fault diagnosis components. The advantage here is that these languages tend not to be Turing complete, and so are simpler to verify than general-purpose programming languages. This reliance on a set of relatively simple representation languages is key to our approach.

2 Background

Currently, we have focused on two languages used for autonomous systems. MPL (Model-Based Processing Language) 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 [11]. The Livingstone inference engine observes the physical system, predicts its current state according to the MPL model, detects discrepancies between the predicted and observed states, and diagnoses potential faults if discrepancies exist. MPL has been used in the fault diagnosis component of the Remote Agent architecture. Figure 1 shows a simple component written in MPL.

TDL (Task Description Language) is an extension of C++ that includes constructs for expressing hierarchical task decomposition and synchronization constraints between tasks, monitoring task execution, and handling exceptions [10]. TDL simplifies the process of specifying how concurrent robot tasks should, and should not, behave and

```
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)

interact. TDL, which is based on the TCA (Task Control Architecture) [7], has been used to implement the "executive" layer of several autonomous mobile robot systems. While TDL, being an extension of C++ *is* Turing complete, we actually just deal with verifying the parts of TDL that are concerned with task management. Figure 2 shows a simple task specification written in TDL.

We perform the verification using the SMV model checker [2]. SMV uses symbolic computation techniques based on Binary Decision Diagrams (BDDs) [1] to compactly and efficiently manipulate sets of system configurations. This allows SMV to tackle very large state spaces of 10^{100} states and beyond.

3 Translation for Model Checking

The first step in automating the translation of these higher level languages to SMV is to formalize the inference engine.[1] Essentially, this involves formalizing the state transitions that occur as the languages are interpreted by the inference engine. For MPL, a first-order formalization is fairly straightforward, since MPL is based on 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

1. Note that we are not trying to verify the implementation of the inference engine itself. That is an important task, but since it is a one-time process, it can more profitably be done with traditional formal verification techniques.

```
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

way. TDL is organized around the execution of tasks, each of which goes through various phases. The phase transitions can be formalized as finite state machines, with constraints between different phases and between different tasks. Using this formalization, we can represent the synchronization between tasks and verify whether desired properties (such as liveness, safety, and absence of resource conflict) hold.

After the inference engine is formalized, a translator is written that produces SMV code from the higher-level language. The generated SMV code encodes the state transitions that occur as the inference engine interprets the higher-level code. For instance, SMV code for an MPL model encodes how the state of a component transitions from one mode to the next as the component receives commands. The SMV code also encodes constraints between states. For instance, SMV code for a TDL program can encode the constraint that one task may not begin execution until another task is completed. Figure 3 shows the translated SMV code for the switch component in Figure 1. Note that information that is just implicit in the original model (especially relating to state transitions) is made explicit in the SMV model.

Specifications are temporally quantified formulae that describe properties that one wants the system to exhibit. For instance, we may want to verify that some proposition will always hold after some other proposition becomes true. In SMV, specifications are represented using CTL (Concurrent Temporal Logic). We have extended MPL to enable CTL specifications to be written directly using MPL syntax. In addition, we have included some standard properties (such as completeness, disjointness, reachability) that developers have found to be useful in verifying MPL models. These properties expand into sets of CTL formulae that can then be verified along with the explicitly encoded specifications. We are currently designing extensions to TDL that would enable specifications to be described directly in the language.

4 Processing Counter-Examples

Given an SMV model and a specification written in CTL, the model checker determines whether the specification holds and, if not, produces a counter-example showing a sequence of state transitions that make the specification false. The counter-examples are essentially symptoms of bugs in the original system. However, it is usually quite difficult to diagnose the error directly from the counter-example. One reason is that the counter-example 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 counter-example back to the original high-level language.

A more serious difficulty is that a counter-example 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 counter-examples and present the data in a way in which the cause of the problem may be more apparent. The idea is that many autonomous system inference engines have associated visualization tools used to distill and present data from log files of actual runs of the system. Figure 4, for instance, shows one such tool, developed for the Remote Agent, to visualize the "Smart Executive" component [9]. It enables users to see how tasks are executed, graphically indicates when violations of constraints are detected, and enables users to interactively investigate constraints between tasks. To use such tools for visualizing counter-examples, one needs to translate the state transitions in the counter-examples into the log file format that the tool expects. In essence, we are creating a log file of a faulty run that corresponds to the false specification. Once this is done, the user can utilize all the features of the visualization tool to help understand the problem.

The other way we are addressing the problem of understanding counter-examples is to produce textual explanations of the problem. Our current approach is to take the counter-example and feed it into a TMS (Truth Maintenance System), along with the original SMV model.

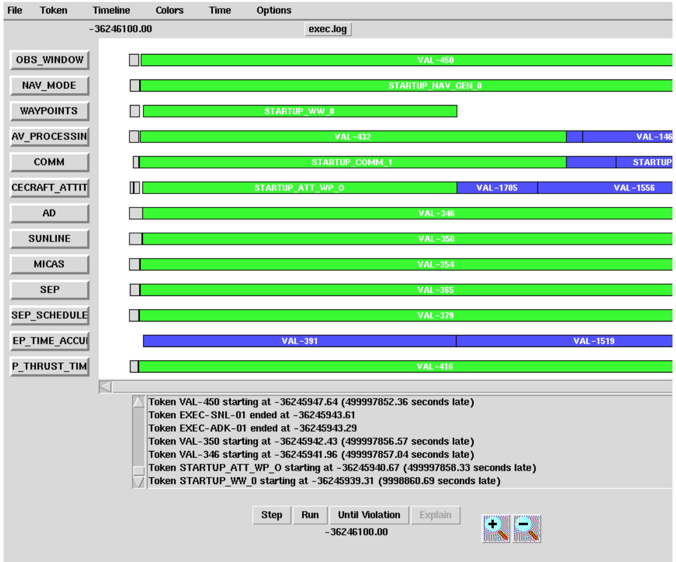

Figure 4: Executive Visualization Tool

The TMS [6] propagates the constraints of the model and the state of the system (given by the counter-example). When it detects an inconsistent proposition, we trace 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.

Figure 5 shows an example of an explanation produced for an MPL model of a motor controller for a planetary rover. The symptom is that the "*off*" mode of the "*pid-controller*" is unreachable. The explanation for this is that all of the possible modes for the "*torque-controller*" are inconsistent with the "*off*" mode because the "*off*" mode asserts that power is off, and all the "*torque-controller*" modes assert that power is on, and that the power inputs to the two controllers are the same. This information is of great use in determining what part of the model is at fault (in this particular case, an "*off*" mode was be added to the "*torque-controller*").

5 Current Status

The current state of this work is that we have formalized all of MPL and a large subset of the TDL syntax that deals with task management. We have extended MPL to enable specifications to be encoded directly in MPL. We have created automated translators for both MPL and the subset of TDL that is formalized. Currently, we are working on translating from SMV back to MPL and TDL. We have some prototype explanation software for MPL (the explanation in Figure 5 is produced automatically, directly from an SMV counter-example). Our focus is to continue investigating the generation of meaningful MPL explanations from counter-examples, for a wide range of

It is inconsistent for the *pid-controller* of *(motor-module ?motor)* to be in the *off* mode because:

1. The *torque-controller* cannot be in the *nominal* mode because
    - 1.1 The *nominal* mode of the *torque-controller* states that its *power-input* is *on*
    - 1.2 The *power-input* of the *torque-controller* equals the *power-input* of the *pid-controller*
    - 1.3 The *off* mode of the *pid-controller* states that its *power-input* is *off*
2. The *torque-controller* cannot be in the *overloaded* mode because ...
3. The *torque-controller* cannot be in the *amp-fault* mode because ...
4. The *torque-controller* must be in one of the modes: *nominal, overloaded, amp-fault*

Figure 5: Automatically Produced Explanation

CTL specifications, and to investigate the visualization of counter-examples for TDL. We also intend to complete the formalization and translation of TDL. We expect that the work on MPL will be ported to NASA for use in actual development projects.

The translator from MPL to SMV has been used in several case studies in verification of Livingstone applications, including the Remote Agent spacecraft controller [5], the In-Situ Propellant Production system (ISPP), a chemical processing unit that produces spacecraft propellant out of the Mars atmosphere, and the Xavier mobile robot [8]. Using the translator and SMV, NASA researchers checked flow properties in the Livingstone model of a portion of the ISPP system [4]. In one instance, the counter-example reported by SMV for a property violation enabled developers to identify an error in the modeling of flow equations. The latest ISPP model, with more than 100 variables and 10^{50} states, still fits within the limits of what SMV can handle.

We believe that by providing translation and explanation tools, we can make model checking and formal verification techniques available to developers of autonomous systems. The result should be more reliable autonomous systems and reduced debugging/testing effort.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

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