Verification of Intelligent Control Software

Charles Pecheur
Université catholique de Louvain
place Ste Barbe 2, 1348 Louvain-la-Neuve, Belgium
charles.pecheur@uclouvain.be

Abstract

Autonomous embedded controllers are seen as a critical technology to enable new mission objectives and scale down operating costs for space applications. However, the validation of intelligent controls software poses a huge challenge, where traditional testing approaches fall short of providing the required level of confidence for such safety-critical applications. This is an overview of recent research in applying modern, analytical verification technologies and tools to the validation of autonomy software, in the context of space applications, at NASA Ames Research Center in California, with a particular focus on model-based approaches to autonomous control, and more specifically fault diagnosis systems. We have developed and experimented with two lines of tools, both related to model checking techniques. Verifying diagnosis systems and models has led to considering the issue of diagnosability, in the sense of checking whether a system provides sufficient observations to determine and track its internal state with sufficient accuracy. We discuss how this kind of question can be reduced to a modified model checking problem. Diagnosability analysis also expands to the domain of epistemic (i.e. knowledge) models and logics.

1 Overview

Embedded controllers are more and more pervasive and feature more and more advanced capabilities. For space applications in particular, the development of autonomous controllers, capable of standalone operation in unpredictable situations, all the way up to mission-level concerns, is seen as a critical technology to enable new mission objectives and scale down operating costs. On the flip side, the validation of intelligent controls software poses a huge challenge, both due to the increased complexity of the system itself and the broad spectrum of normal and abnormal conditions in which it has to be able to operate. Traditional testing approaches fall short of prviding the required level of confidence for such safety-critical applications.

This paper supports a presentation of recent research in applying modern, analytical verification technologies and tools to the validation of autonomy software, in the context of space applications, at NASA Ames Research Center in California. In particular, the work focuses on analysing model-based approaches to autonomous control, and more specifically fault diagnosis systems, as exemplified by NASA Ames' Livingstone system. The content of this paper is largely based on selected excerpts from prior publications [1, 2, 3, 4, 5].

We have developed two lines of tools and experimented with those tools on real-size problems taken from NASA applications. Under different angles, both approaches stem from the verification discipline known as model checking. The first approach, presented in [3] and in Section 4, addresses the verification of the domain models used for model-based diagnosis. The second approach, presented in [2] and in Section 3, augments classical testing approaches with fine control and automation capabilities inspired from model checking.

Verifying diagnosis systems and models has led to considering the issue of diagnosability: given a partially observable dynamic system, and a diagnosis system observing its evolution over time, how to verify (at design time) whether the system provides sufficient observations to determine and track (at run-time) its internal state with sufficient accuracy. This kind of question can be answered by looking for pairs of scenarios that are observationally indistinguishable, but lead to situations that are required to be distinguished. This search amounts to a modified model checking problem, and we have extended our tools to support that kind of analysis. This is explained in [4] and in Section 5.

Diagnosability analysis is obviously very relevant to hazard analysis in the context of space-bound fault protection systems, but is also an incarnation of the more general notion of observability, which has applications in such areas human interfaces or security protocols. When coupled to the complementary notion of controllability

(or, in the case of failures, recoverability), it expands to the domain of epistemic (i.e. knowledge) models and logics, and the verification techniques that apply to them. This is discussed in [5] and in Section 6.

2 Background

2.1 Livingstone

Livingstone is a model-based health monitoring system developed at NASA Ames [6]. It uses a symbolic, qualitative model of a physical system, such as a spacecraft, to infer its state and diagnose faults. Livingstone is one of the three parts of the Remote Agent (RA), an autonomous spacecraft controller developed by NASA Ames Research Center jointly with the Jet Propulsion Laboratory. The two other components are the Planner/Scheduler [7], which generates flexible sequences of tasks for achieving mission-level goals, and the Smart Executive [8], which commands spacecraft systems to achieve those tasks. Remote Agent was demonstrated in flight on the Deep Space One mission (DS-1) in May 1999, marking the first control of an operational spacecraft by AI software [9]. Livingstone is also used in other applications such as the control of a propellant production plant for Mars missions [10], the monitoring of a mobile robot [11], and intelligent vehicle health management (IVHM) for the X-37 experimental space transportation vehicle.

The Livingstone engine has two complementary modules. The *Mode Identification* module (MI) estimates the current state of the system by tracking the commands issued to the device. It then compares the predicted state of the device against observations received from the actual sensors. If a discrepancy is noticed, Livingstone performs a diagnosis by searching for the most likely configuration of component states that are consistent with the observations. Using this diagnosis, the *Mode Recovery* module (MR) can suggest an action to recover to a given goal configuration.

The model used by Livingstone describes the normal and abnormal functional modes of each component in the system, using a declarative formalism called MPL. The model is composed from a hierarchy of elementary *components*, assembled into compound *modules*. Each component definition may have parameters and describes the *attributes* and *modes* of the component. Attributes are the state variables of the component, ranging over qualitative, discrete values: continuous physical domains have to be abstracted into discrete intervals such as {*low*, *nominal*, *high*} or {*neg*, *zero*, *pos*}. Modes identify both nominal and fault modes of the component. Each mode specifies constraints on the values that variables may take when the component is in that mode, and how the component can transition to other modes (by definition, spontaneous transitions to any fault mode can happen from any mode). The Livingstone model thus represents a combination of concurrent finite-state transition systems.

2.2 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 expected property of that system, a model checker will run through all possible executions of that system, including all possible interleavings of concurrent threads, and report any execution that leads to a property violation. We refer the reader to [12] and [13] for a general introduction to the theory and practice of model checking.

Classical, explicit-state model checkers such as SPIN [14] do this by generating and exploring every single state. In contrast, symbolic model checking manipulates whole sets of states at once, implicitly represented as the logical conditions that those states satisfy. These conditions are encoded into data structures called *(Ordered) Binary Decision Diagrams* (BDDs) [15], that provide a compact representation and support very efficient manipulations.

Symbolic model checking can often address much larger systems than explicit state model checkers. It has traditionally been applied to hardware systems, and is increasingly being used to verify software systems as well. It does not work well for all kinds of models, however: the complexity of the BDDs can outweigh the benefits of symbolic computations, and BDDs are still exponential in the size of the system in the worst case.

We have been using the *Symbolic Model Verifier* (SMV), from Carnegie Mellon University (CMU) [16]. SMV was the first and is still one of the most widely used symbolic model checkers. The SMV software takes as input a model to be verified, written in a custom modeling language. This model also contains the properties to be verified, written in a variant of temporal logic known as *Computation Tree Logic* or CTL. Upon completion, SMV reports a true/false answer for each specified property, along with an execution trace witnessing the

property violation, in case of a false answer. Internally, SMV uses BDD-based symbolic model checking as described in the previous section. SMV does not reorder variables by default, but provides options to apply reordering and to export and import variable orderings in text files that can be modified by the user. A new, upward-compatible version of SMV, called *NuSMV* [17], has been re-built from scratch at IRST (Trento, Italy), in collaboration with CMU. NuSMV features much improved performance, and a cleaner, well documented and modular source code amenable to customization.

3 Simulation-Based Verification

This section is based on [2], and is joint work with Tony Lindsey (QSS).

This section discusses a flexible framework for simulating, analyzing and verifying autonomous controllers. The proposed approach applies state space exploration algorithms to an instrumented testbed, consisting of the actual control program being analyzed embedded in a simulated operating environment. This framework forms the foundation of *Livingstone PathFinder* (LPF), a verification tool for autonomous diagnosis applications based on NASA's Livingstone model-based diagnosis system.

The approach we follow in the work presented is a composite between conventional testing and model checking, here referred to as *simulation-based verification*. Similar to conventional testing, it executes the real program being verified rather than an abstract model derived from the system. In order to support its interactions, the program is embedded in a testbed that simulates its *environment*. On the other hand, as in model checking the execution ranges over an entire graph of possible behaviors as opposed to a suite of linear test cases. In the optimal setting, each visited state is marked to avoid redundant explorations of the same state and can be restored for backtracking to alternate executions. Furthermore, sources of variation in the execution, such as external events, scheduling or faults, are controlled to explore all alternatives.

The rationale behind simulation-based verification is to take the advanced state space exploration algorithms and optimizations developed in the field of model checking and apply them to the testing of real code. By doing so, we avoid the need for developing a separate model for verification purposes, and more importantly for scrutinizing each reported violation to assess whether it relates to the real system or to a modeling inaccuracy. Of course, simulation-based verification will in general be significantly less efficient and scalable than model checking an abstract model of the same program. However, it should be seen as an evolutionary improvement to traditional testing approaches, with important potential gains in scalability, automation and flexibility.

To enable their controlled execution, instrumentation is introduced in both the analyzed program and its environment. To perform a true model-checking search, the tool should be capable of iterating over all alternate events at each state, backtracking to previously visited states, and detecting states that produce the same behavior. Some of these capabilities may be supported only partly or not at all, depending on the nature of the different components in the testbed and the needs and trade-offs of the analysis being performed. For example, a complex piece of software may provide checkpointing capabilities on top of which backtracking can easily be built; however, state equivalence might require an analysis of internal data structures that is either too complex, computationally expensive, or infeasible due to the proprietary nature of the software. In addition, this analysis may not be worthwhile because equivalent states will seldom be reached anyway. Even if true backtracking is not available, it can still be simulated by re-playing the sequence up to the desired state. This reduces the exploration to a suite of sequential test cases, but the additional automation and flexibility in search strategies can still be beneficial.

With these capabilities, the program and its testbed constitute a *virtual machine* that embodies a fully controllable state machine, whose state space can be explored according to different strategies, such as depth-first search, breadth-first search, heuristic-based guided search, randomized search, pattern-guided search or interactive simulation. The environment portion of the testbed is typically restricted to a well-defined range of possible scenarios in order to constrain the state space within tractable bounds.

Livingstone PathFinder can be considered on three different levels:

- As a *verification approach*, LPF applies a combination of model checking and testing principles that we refer to as *simulation-based verification*.
- As a *program framework*, LPF provides an infrastructure for applying simulation-based verification to autonomous controllers.

- As a *concrete program*, LPF currently instantiates the framework to applications based on the Livingstone diagnosis system.

The architecture of the LPF tool consists of the following three components:

- *Diagnosis*: the diagnosis system being analyzed, based on the Livingstone diagnosis *engine*, interpreting a *model* of the physical system.
- *Simulator*: the simulator for the physical system on which diagnosis is performed. Currently this is a second Livingstone engine interpreting a model of the physical system. The models used in the Diagnosis and the Simulator are the same by default but can be different.
- *Driver*: the simulation driver that generates commands and faults according to a user-provided scenario script. The scenario file is essentially a non-deterministic test case whose elementary steps are commands and faults.

LPF accepts as input a Livingstone model of the physical system and a scenario script defining the class of commands and faults to be analyzed. The model is used to perform model-based diagnosis and will be used to simulate the system as well. The tool runs through all executions specified in the script, backtracking as necessary to explore alternate routes. At each step, LPF checks for error conditions, such as discrepancies between the actual simulated faults and those reported by diagnosis. If an error is detected, LPF reports the sequence of events that led to the current state.

The PITEX experiment has demonstrated the application of Livingstone-based diagnosis to the main propulsion feed subsystem of the X-34 space vehicle [18, 19], and LPF has been successfully applied to the PITEX model of X-34. This particular Livingstone model consists of 535 components and 823 attributes, 250 transitions, compiling to 2022 propositional clauses.

4 Verification of Diagnosis Models

This section is based on [3] and is joiint work with Reid Simmons (CMU) and Peter Engrand (NASA KSC).

Livingstone involves the interaction between various components: the reasoning engine that performs the diagnosis, the model that provides application-specific knowledge to it, the physical system being diagnosed, the executive that drives it and acts upon diagnosis results. There are many ways that one could attempt to provide better verification tools for all or parts of such a system, borrowing from various existing techniques, from improved testing up to computer-supported mathematical proofs. The focus of the work presented here is on the verification of the Livingstone model, by turning this model into a representation suitable for model checking. Since the model is specific to the application that it is used for, it is indeed where the correctness issues are most likely to occur during the development of a given application.

In many previous experiences in model checking of software, the system to be verified has been translated by hand from its original design to the custom input syntax of the target verification tool. This translation is by far the most complex and time-consuming part, typically taking weeks or months, whereas the running of the verification is a matter of minutes or hours thanks to the processing power of today's computers. The net result is that software model checking has been so far mostly performed off-track by formal methods experts, rather than by field engineers as part of the development process. This gap between verification and software development formalisms and tools is recognized as one of the main obstacles to the widespread adoption of formal verification by the software industry.

Our goal is to allow Livingstone application developers to use model checking to assist them in designing and correcting their models, as part of their usual development environment. To achieve that, we have developed a translator to automate the conversion between MPL and SMV. To completely isolate the Livingstone developer from the syntax and technical details of the SMV version of his model, we need to address three kinds of translation:

- The MPL model needs to be translated into an SMV model amenable to model checking.
- The specifications to be verified against this model need to be expressible in terms of the MPL model and similarly translated.

- Finally, the diagnostic traces produced by SMV need to be converted back in terms of the MPL model.

We have developed a translator that covers these three aspects.

The translation of Livingstone models to SMV is facilitated by the strong similarities between the underlying semantic frameworks of Livingstone and SMV. Both boil down to a synchronous transition system, defined through propositional logic constraints on initial states, on all states and on transitions.

The CTL logic used for SMV specifications is very expressive but requires a lot of caution and expertise to be used correctly. To alleviate this problem, the translator provides pre-defined specification templates for some common classes of properties that are generally useful to check, independently of any given application. The user can refer to them by their meaningful name, and the translator automatically produces the corresponding CTL specifications. The translator also supports some additional features, such as auxiliary functions that concisely capture Livingstone concepts such as occurrence of faults, activation of commands or probability of faults.

When a violated specification is found, SMV reports a diagnostic trace, consisting of a sequence of states leading to the violation. This trace is essential for diagnosing the nature of the violation. The states in the trace, however, show variables by their SMV names. To make sense to the Livingstone developer, it has to be translated back in terms of the variables of the original MPL model. This can done using the lexicon generated for the model translation in the reverse direction. A more arduous difficulty is that the diagnostic trace merely indicates the states that led to the violation but gives no indication of what, within those states, is really responsible.

The translator has been used by Livingstone application engineers in the In-Situ Propellant Production (ISPP) project at NASA Kennedy Space Center (KSC). The purpose of this experience was not only to experiment with the translator and SMV, but also to study the potentials and challenges of putting such a tool in the hands of application practitioners. The target was the In-Situ Propellant Production (ISPP) plant, a system intended to produce spacecraft propellant using the atmosphere of Mars [10].

The size of the ISPP Model was 10^{17} States. Although this is a relatively large state space, SMV needed less than a minute to return a result. The size of the latest ISPP Model is on the order of 10^{55} states, and can still be processed in a matter of minutes using an enhanced version of SMV [20]. The Livingstone model of ISPP features a huge state space but little depth (all states can be reached within at most three transitions), for which the symbolic processing of SMV is very appropriate. The experience has also been a steep learning curve for the application developer in charge of the formal verification experiment at KSC.

5 Verification of Diagnosability

This section is based on [4] and is joint work with Alessandro Cimatti (IRST) and Roberto Cavada (IRST).

Diagnosability is the possibility for an ideal diagnosis system to infer accurate and sufficient run-time information on the behavior of the observed system. Diagnosability has been studied by several authors in the domain of discrete systems [21, 22, 23, 24, 25] and timed systems [26].

The diagnosability problem can be formally characterized, using the idea of *context*, that explicitly takes into account the run-time conditions under which it should be possible to acquire certain information. A diagnosability condition for a given plant is violated if and only if a *critical pair* can be found. A critical pair is a pair of executions that are indistinguishable (i.e. share the same inputs and outputs), but hide conditions that should be distinguished (for instance, to prevent simple failures to stay undetected and degenerate into catastrophic events). Given a plant being diagnosed, a *coupled twin model* of the plant can be defined and used to search for critical pairs. The diagnosability problem can then be recast in terms of temporal logic formulae, and reduced to a model checking problem over the coupled twin model.

We have developed a platform able to generate formal models for the twin plant, starting from Livingstone models, absed on the Livingstone-to-SMV translator. Several diagnosability problems corresponding to interesting scenarios from real-world applications were tackled by means of the NuSMV model checker [17]. An experimental analysis shows that the verification of diagnosability can be practical: large Livingstone models of space transportation systems are automatically analyzed within seconds by means of SAT-based symbolic model checking techniques.

6 From Diagnosis to Knowledge

This section is based on [5] and is joint work with Franco Raimondi (UCL) and Alessio Lomuscio (KCL).

Recently, various extensions of model checking techniques and tools have been investigated for the verification of richer modal logics that include modal operators to reason about time, knowledge, beliefs, and strategies. The model checker MCK [27] is based on ordered binary decision diagrams (OBDDs), and supports the verification of epistemic and temporal properties for systems defined on interpreted systems semantics. Similarly, the tool MCMAS [28] uses OBDDs and allows to reason about time, knowledge, and correct behaviour of agents. The development of these tools is motivated by the interest in the automatic verification of various scenarios in which it seems more natural to reason about epistemic properties of multi-agent systems [29], as in communication and security protocols.

In particular, we have investigated how *diagnosability* and *recoverability* can be expressed as temporal-epistemic *specification patterns*. For instance, *diagnosability* can be naturally expressed by ascribing a form of knowledge to a diagnoser: a diagnoser is able to diagnose a fault f iff the diagnoser always knows whether f or $\neg f$. Diagnosability and recoverability properties of a system correspond, respectively, to the feasibility of diagnosing and recovering from faults in that system, given available observable and controllable variables (sensors and actuators). In this sense, they are particular forms of observability/controllability properties found in classical control theory. As opposed to the approach discussed in the previous section, in this case we represent diagnosability by using epistemic and temporal properties of *agents*. Following this, we have used a MCMAS [28] to verify a Livingstone model. MCMAS is a model checker for the verification of temporal, epistemic, and correctness properties of agents. MCMAS extends to more complex logics the traditional model checking algorithms for CTL [12] and uses OBDDs [15] as an efficient encoding technique.

References

[1] Tim Menzies and Charles Pecheur. Verification and validation and artificial intelligence. In M. Zelkowitz, editor, *Advances in Computing*, volume 65. Elsevier, 2005.

[2] Tony Lindsey and Charles Pecheur. Simulation-based verification of autonomous controllers with livingstone pathfinder. In *Proceedings of the Tenth International Conference on Tools And Algorithms For The Construction And Analysis Of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*. Springer Verlag, apr 2004.

[3] Charles Pecheur, Reid Simmons, and Peter Engrand. Formal verification of autonomy models: From livingstone to smv. In *Agent Technology from a Formal Perspective*, NASA Monographs in Systems and Software Engineering. Springer Verlag, 2006.

[4] Alessandro Cimatti, Charles Pecheur, and Roberto Cavada. Formal verification of diagnosability via symbolic model checking. In *Proceedings of IJCAI'03*, Acapulco, Mexico, 2003.

[5] F. Raimondi, C. Pecheur, and A. Lomuscio. Applications of model checking for multi-agent systems: verification of diagnosability and recoverability. In *Proceedings of Concurrency, Specification & Programming (CS&P)*, pages 433–444. Warsaw University, September 2005.

[6] B. C. Williams and P. P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of AAAI-96*, 1996.

[7] N. Muscettola. *HSTS: Integrating planning and scheduling*. Morgan Kaufman, 1994.

[8] B. Pell, D. E. Bernard, S. A. Chien, E. Gat, N. Muscettola, P. P. Nayak, M. Wagner, and B. C. Williams. An autonomous spacecraft agent prototype. *Autonomous Robots*, 5(1), March 1998.

[9] 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):5–48, 1998.

[10] D. Clancy, W. Larson, C. Pecheur, P. Engrand, and C. Goodrich. Autonomous control of an in-situ propellant production plant. In *Proceedings of Technology 2009 Conference*, Miami, 1999.

[11] Reid G. Simmons, Richard Goodwin, Karen Zita Haigh, Sven Koenig, Joseph O'Sullivan, and Manuela M. Veloso. Xavier: Experience with a layered robot architecture. *Intelligence*, 2001. to appear.

[12] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. Mit Press, 1999.

[13] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[14] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[15] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[16] 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):142–170, 1992.

[17] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In *Proc. of International Conference on Computer-Aided Verification*, 1999.

[18] A. Bajwa and A. Sweet. The Livingstone model of a main propulsion system. In *Proceedings of the IEEE Aerospace Conference*, 2003.

[19] C. Meyer and H. Cannon. Propulsion ivhm technology experiment overview. In *Proceedings of the IEEE Aerospace Conference*, 2003.

[20] B. Yang, R. Simmons, R. Bryant, , and D. O'Hallaron. Optimizing symbolic model checking for invariant-rich models. In *Proc. of International Conference on Computer-Aided Verification*, 1999.

[21] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, September 1995.

[22] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Failure diagnosis using discrete event models. *IEEE Transactions on Control Systems*, 4(2):105–124, March 1996.

[23] S. Jiang and R. Kumar. Failure diagnosis of discrete event systems with linear-time temporal logic fault specications, 2002. IEEE Trans. on Automatic Control.

[24] S. Jiang, Z. Huang, V. Chandra, and R. Kumar. A polynomial algorithm for testing diagnosability of discrete event systems. *IEEE Transactions on Automatic Control*, 46(8):1318–1321, August 2001.

[25] L. Console, C. Picardi, and M. Ribaudo. Diagnosis and diagnosability using pepa. In *Proceedings of the European Conference on Artificial Intelligence*, pages 131–136, Berlino, Germany, August 2000. IOS Press.

[26] Stavros Tripakis. Fault diagnosis for timed automata. In *Proceedings of FTRTFT*, 2002.

[27] P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In *Proceedings of 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 479–483. Springer-Verlag, 2004.

[28] F. Raimondi and A. Lomuscio. Automatic verification of multi-agent systems by model checking via OBDD's. *Journal of Applied Logic*, 2005. To appear in Special issue on Logic-based agent verification.

[29] M. Wooldridge. *Reasoning about Rational Agents*. MIT Press, 2000.