From Livingstone to SMV
Formal Verification for Autonomous Spacecrafts

Charles Pecheur[1] and Reid Simmons[2]

[1] RIACS / NASA Ames Research Center, Moffett Field, CA 94035, U.S.A.
pecheur@ptolemy.arc.nasa.gov
[2] Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.
reids@cs.cmu.edu

Abstract. To fulfill the needs of its deep space exploration program, NASA is actively supporting research and development in autonomy software. However, the reliable and cost-effective development and validation of autonomy systems poses a tough challenge. Traditional scenario-based testing methods fall short because of the combinatorial explosion of possible situations to be analyzed, and formal verification techniques typically require a tedious, manual modelling by formal method experts. This paper presents the application of formal verification techniques in the development of autonomous controllers based on Livingstone, a model-based health-monitoring system that can detect and diagnose anomalies and suggest possible recovery actions. We present a translator that converts the models used by Livingstone into specifications that can be verified with the SMV model checker. The translation frees the Livingstone developer from the tedious conversion of his design to SMV, and isolates him from the technical details of the SMV program. We describe different aspects of the translation and briefly discuss its application to several NASA domains.

1 Introduction

As NASA's missions continue to explore Mars and beyond, the great distances from Earth will require that they be able to perform many of their tasks with an increasing amount of autonomy, including navigation, self-diagnosis, and on-board science. For example, the Autonomous Controller for the In-Situ Propellant Production facility, designed to produce spacecraft fuel on Mars, must operate with infrequent and severely limited human intervention to control complex, real-time, and mission-critical processes over many months in poorly understood environments [4].

While autonomy offers promises of improved capabilities at a reduced operational cost, there are concerns about being able to design, implement and verify such autonomous systems in a reliable and cost-effective manner. Traditional scenario-based testing methods fall short of providing the desired confidence level, because of the combinatorial explosion of possible situations to be analyzed.

Often, formal verification techniques based on model checking[1] are able to efficiently check all possible execution traces of a system in a fully automatic way. However, the system typically has to be manually converted beforehand into the syntax accepted by the model checker. This is a tedious and complex process, that requires a good knowledge of the model chacker, and is therefore usually carried externally by a formal methods expert, rather than by the system designer himself.

This paper presents the application of formal verification techniques in the development of autonomous controllers based on Livingstone, a model-based health management and control system that helps to achieve this autonomy by detecting and diagnosing anomalies and suggesting possible recovery actions. We present a translator that converts the models used by Livingstone into specifications that can be verified with the SMV model checker from Carnegie Mellon University. The translator converts both the Livingstone model and the specification to be verified from Livingstone to SMV, and then converts any diagnostic trace from SMV back to Livingstone. It thereby shields the Livingstone application designer from the technicalities of the SMV model checker.

Sections 2 and 3 respectively present the Livingstone health management system and the SMV model checker. Section 4 introduces our translator and describes its different parts. Section 5 discusses its application to several NASA projects, Section 6 develops some comments on the nature of the verification problem for autonomy model, and Section 7 draws final conclusions.

2 Livingstone

Livingstone is a model-based health monitoring system developed at NASA Ames [9]. It uses a symbolic, qualitative model of equipment to infer its state and diagnose failures. Livingstone is one of the three parts of the Remote Agent (RA), an autonomous spacecraft controller developed by NASA Ames Research Center conjointly with the Jet Propulsion Laboratory. The two other components are the Planner/Scheduler, which generates flexible sequences of tasks for achieving mission-level goals, and the Smart Executive, 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 [6]. Livingstone is also used in other applications such as the control of a propellant production plant for Mars missions and the monitoring of a mobile robot.

The functioning of Livingstone is depicted in Fig. 1. 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 set of component

[1] As opposed to those based on theorem proving, which can provide even more general results but require an even more involved and skilled guidance.

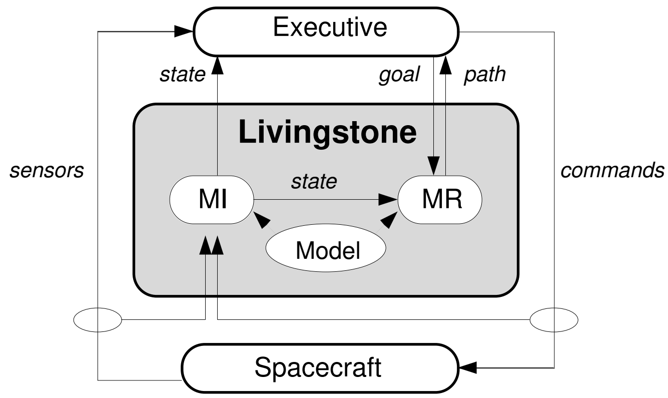

Fig. 1. Livingstone mode identification (MI) amd mode recovery (MR)

mode assignments that are consistent with the observations. Using this diagnosis, the *Mode Recovery* module (MR) can compute a path 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. Livingstone describes components using a declarative formalism called Model Programming Language (MPL), which has a Lisp-like syntax. Components are parameterized and are described using variables taking qualitative, discrete values. For each component, a set of modes is defined identifying both its nominal and failure modes. Each mode specifies constraints on the values that variables may take when the component is in that mode, and how the component can switch to other modes (by definition, spontaneous transition to any failure mode can happen from any mode). The Livingstone model thus represents a combination of concurrent finite-state transition systems. For example, Fig. 2 presents a simple MPL model for a valve, with a variable `flow` ranging over {`off`, `low`, `nominal`, `high`}. It has two nominal modes `open` and `closed` and two failure modes `stuck-open` and `stuck-closed`. The `closed` mode enforces `flow=off` and allows a transition `do-open` to the `open` mode, triggered when the `cmd` variable has value `open`.

3 Symbolic Model Checking and SMV

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.

Classical, explicit-state model checkers such as SPIN [5] 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

```
(defvalues flow (off low nominal high))
(defvalues valve-cmd (open close no-cmd))
(defcomponent valve (?name)
  (:inputs (cmd :type valve-cmd))
  (:attributes ((flow ?name) :type flow) ...)
  (closed :type ok-mode :model (off (flow ?name))
    :transitions ((do-open :when (open cmd) :next open) ...))
  (open :type ok-mode ...)
  (stuck-closed :type fault-mode ...)
  (stuck-open :type fault-mode ...))
```

Fig. 2. A simple MPL Model of a valve

that those states satisfy. These conditions are encoded into data structures called Binary Decision Diagrams (BDDs) [1], that provide a compact representation and support very efficient manipulations. Typically, a BDD of the current set of states is combined with a BDD of the transition relation to obtain a BDD of the next set of reachable states. Symbolic model checking can address much larger systems than explicit state model checkers, but does not work well for all systems: 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. While symbolic model checking has traditionally been applied to hardware systems, it is increasingly being used to verify software systems, as well.

SMV, from Carnegie-Mellon University (CMU) [2], is one of the most popular symbolic model checkers. An SMV specification uses variables with finite types, grouped into a hierarchy of module declarations. Each module states its local variables, their initial value and how they change from one state to the next. Properties to be verified can be added to any module. The properties are expressed in CTL (Computation Tree Logic). CTL is a branching-time temporal logic, which means that it supports reasoning over both the breadth and the depth of the tree of possible executions. For example, the CTL formula `AG flow=high` states that **A**lways (for all executions) **G**lobally (all along each execution), the flow is high.

4 Verification of Livingstone Models

The Livingstone engine performs complex computations using large-size data structures capturing its knowledge about the model. In order to apply model checking to the autonomous controller as a whole, we would need an SMV specification of the Livingstone engine and its data structures, including the Livingstone model. Producing such a specification would be an arduous and error-prone task. Furthermore, the size of the data structures involved would severely limit the tractability of model checking.

Alternatively, the autonomy model can be considered as a high-level program that is "executed", in a somewhat unusual way, by Livingstone. The Livingstone

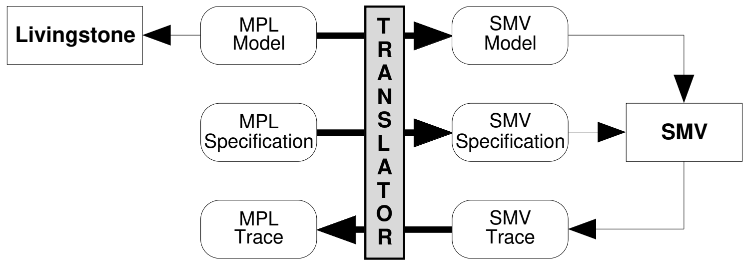

Fig. 3. Three kinds of translation between MPL and SMV

program itself is a more complex, but also more stable and better understood part, built around well-documented algorithms. Since it remains basically unchanged between applications, the verification of its correctness can be done once and for all by its designers and is not addressed here. From the point of view of its users, Livingstone is viewed as a stable, trustable part, just as C programmers trust their C compiler.

Therefore, the focus of this article is on the verification of a Livingstone model with respect to the real system that this model represents. This can be addressed by turning this model into a representation suitable for model checking. Since this model is specific to the application that it is used for, it is indeed where the correctness issues are most likely to occur.

In many previous experiences in model checking of software, this translation has been done by hand. This is usually the most complex and time-consuming part, typically taking weeks or months, whereas the running 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 is currently mostly performed off-track by formal methods experts, rather than by field engineers as part of the development process.

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, as shown in Fig. 3:

- 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.

```
MODULE valve
VAR     mode: {open,closed,stuck-open,stuck-closed};
        cmd: {open,close,no-cmd};
        flow: {off,low,nominal,high};
DEFINE faults:={stuck-open,stuck-closed};
INVAR mode=closed -> flow=off
TRANS (mode=closed & cmd=open) ->
        (next(mode)=open | next(mode) in faults)
```

Fig. 4. SMV Model of a Valve

These three aspects are covered by our translator and are detailed in the following sub-sections. The translator program has been written in Lisp[2] and is about 4000 lines long.

4.1 Translation of Models

The translation of MPL models to SMV is facilitated by the strong similarities between Livingstone models and SMV specifications. In particular, both have synchronous concurrency semantics. The main difficulty in performing the translation comes from discrepancies in variable naming rules between the flat name space of Livingstone and the hierarchical name space of SMV. Each MPL variable reference, such as `(flow valve-1)`, needs to be converted into a SMV qualified variable reference w.r.t. the module hierarchy, e.g. `ispp.inlet.valve-1.flow`. To optimize this process, the translator builds a lexicon of all variables declared in the MPL model with their SMV counterpart, and then uses it in all three parts of the translation. Figure 4 presents the SMV translation of the MPL model in Figure 2.

4.2 Translation of Specifications

The specifications to be verified with SMV are added to the MPL model using a new **`defverify`** declaration[3]. The **`defverify`** declaration also defines the top-level module to be verified. The properties to be verified are expressed in a Lisp-like style that is consistent with the rest of the MPL syntax; their translation function is an extension of the one used for MPL's logic formulae. For example, the declaration in Fig. 5 specifies a CTL property to be verified on module `ispp`. Without entering into details of CTL, the specification says that, from any non-failure state, a high flow in valve 1 can eventually be reached. Fig. 6 shows the top-level SMV module that is produced from that declaration.

In SMV, specifications use the powerful temporal logic CTL. CTL is very expressive but requires a lot of caution and expertise to be used correctly. To

[2] Lisp was a natural choice considering the Lisp-style syntax of the MPL language.

[3] This is specific to the translator and rejected by Livingstone; an added empty Lisp macro definition easily fixes this problem.

```
(defverify
  (:structure (ispp))
  (:specification
     (always (globally (implies
       (not (broken))
       (exists (eventually (high (flow valve-1)))))))
```

Fig. 5. Specification for verification in MPL

```
MODULE main
  VAR ispp : ispp;
  SPEC AG ((!broken) ->
    EF (ispp.inlet.valve-1.flow = high))
```

Fig. 6. Specification for verification in SMV

alleviate this problem, the translator supports several alternative ways of expressing model properties.

Plain CTL — CTL operators are supported in MPL's Lisp-like syntax, as illustrated in Fig. 5.

Specification Patterns — Common properties such as reachability of given component modes can be concisely expressed using pre-defined specification patterns such as (:reachability (valve valve-1)).

Consistency and completeness are a prime source of trouble for designers of Livingstone models. For example, for all transition statements (name :when <cond> :next <mode>) associated to the same mode, it is required that exactly one of the conditions <cond> hold at each step. If two transitions are enabled simultaneously, then two next modes are enforced at the same time, resulting in inconsistency. To catch these problems, the specification pattern (:disjointness <comp>) extracts the guards of all transitions of component <comp> in the model and generates, for each mode, a mutual exclusion property among its transitions. A peer pattern (:completeness <comp>) checks that at least one guard is always fulfilled.

Auxiliary Functions — The translator supports some auxiliary functions that can be used in CTL formulas to concisely capture Livingstone concepts such as occurrence of failures, activation of commands or probability of failures. Table 1 gives a representative sample. Some functions are translated solely in terms of SMV logic expressions, while others, such as failed, require the introduction of new variables[4].

[4] The latter are omitted by default, since the new variables can cause a big penalty on the performance of SMV.

Table 1. Some auxiliary functions for MPL model specifications

`(broken heater)` =	Heater is in a failed state.
`(failed heater)` =	On last transition, heater failed.
`(multicommand 2)` =	At least two commands are activated.
`(brokenproba 3)` =	Combined probability of currently failed components is at most of order 3.

The probability analysis opens an interesting perspective. Failure probabilities are mapped to small integer order-of-magnitude values (e.g. $p = 10^{-3}$ maps to 3), so that the value for multiple failures can be computed by integer addition, which is supported by SMV's BDD-based analysis. One should note, however, that this is an approximate method, which fits well with the qualitative nature of Livingstone models but is no substitute for a precise approach such as Markov chain analysis.

Observers — The translator allows the definition of *observer automata*, which are cross-breeds between modules (in that they can refer to other components or modules) and components (in that they can have modes). An observer, however, can have no internal variables, other than keeping track of mode. Observers are useful in some situations where the CTL specification language is inadequate for representing the specifications that one wants to verify.

4.3 Translation of Traces

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 is translated back in terms of the variables of the original MPL model. This is achieved 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. Two approaches to this diagnosis problem are currently being investigated. One is based on using visualization tools to expose the trace, the other one uses a truth maintenance system to produce causal explanations [8].

5 Applications

5.1 Deep Space One

Livingstone was originally developed to provide model-based diagnosis and recovery for the Remote Agent architecture on the DS1 spacecraft. The full Livingstone model for the spacecraft runs to several thousand lines of MPL 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. We are also developing specialized declarations to enable us to verify path reachability properties, such as the ability of the system to transition from a fault mode to a known "safe" mode. Using the translator, we were able to identify several (minor) bugs in the DS1 models (this was after the models had been extensively tested by more traditional means) [7].

5.2 ISPP

The translator is being used at NASA Kennedy Space Center by the developers of a Livingstone model for the In-Situ Propellant Production (ISPP), a system that will produce spacecraft propellant using the atmosphere of Mars [3]. First experiments have shown that SMV can easily process the ISPP model and verify useful properties such as reachability of normal operating conditions or recoverability from failures. The current version of the ISPP model, with 10^{50} states, can still be processed in less than a minute using SMV optimizations (re-ordering of variables). 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.

6 Lessons Learned

Concrete applications have shown that the nature of the verification problem for Livingstone models is quite distinct from the verification of a more conventional concurrent application. A typical concurrent system is a collection of active entities, each following a well scoped algorithm. In contrast, a typical Livingstone module describes a passive component such as a tank, valve or sensor; it states how this component reacts to external commands but hardly ever imposes any kind of order of operations in the component itself. On top of that, failures amount to unrestricted spontaneous transitions in every component that allows them.

This results in state spaces that have a very peculiar shape: a huge branching factor, due to all the command variables that can be set and all the failures that can occur at any given step, but a very low depth, due to the very little inherent sequential constraints in the model. In other words, a typical reachability tree for an MPL model is very broad but very shallow, with every state reachable from the initial one within a few transitions.

This also affects the kind of properties that are useful to verify. Looking for deadlocks makes no sense in the presence of spontaneous failure transitions, though more focused reachability properties can reveal inconsistencies in the model. More typically, though, one is interested in consistency and completeness properties, because the declarative nature of MPL makes it very easy to come up with an over- or under-constrained model.

7 Conclusions

Our MPL to SMV translator allows the Livingstone-based application developers to take their MPL model, specify desired properties in a natural extension of their familiar MPL syntax, use SMV to check them and get the results in terms of their MPL model, without reading or writing a single line of SMV code. This kind of separation is an important step towards a wider adoption of verification methods and tools by the software design community.

SMV seems to be very appropriate for certifying Livingstone models for several reasons. First of all, the Livingstone and SMV execution models have a lot in common; they are both based on conditions on finite-range variables and synchronous transitions. Second, the BDD-based symbolic model checking is very efficient for such synchronous systems and appears to fit well to the loosely constrained behaviors captured by Livingstone models. Due to this good match and to the high level of abstraction already achieved by the Livingstone models themselves, it is possible to perform an exhaustive analysis of a direct translation of those models, even for fairly complex models. In contrast, more conventional software model checking applications almost always require some abstraction and simplification stage to make the model amenable to model checking.

This work shows that verification of Livingstone models can be a useful tool for improving the development of Livingstone-based applications. It is, however, only one piece in the larger problem of building and validating autonomous applications. It cannot establish that the Livingstone mode identification will properly identify a situation (though it can establish that there is not enough information to do it). Neither does it address the interaction of Livingstone with other parts of the system, including real hardware with hard timing issues. Other complementary approaches are needed. In this line of work, we are currently prototyping an analytic testing approach based on a controlled execution of an instrumented version of the real Livingstone program in a simulated environment.

References

1. R. E. 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)**, June 1992, pp. 142-170.
3. D. Clancy, W. Larson, C. Pecheur, P. Engrand and C. Goodrich. Autonomous Control of an In-Situ Propellant Production Plant. Technology 2009 Conference, Miami, November 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. Proceedings of the 50th IAF Congress, Amsterdam, Holland, October 1999.
5. G. J. Holzmann. The Model Checker SPIN. IEEE Transactions on Software Engineering, **23(5)**, May 1997.

6. 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,August 1998.
7. P. P. Nayak et al. Validating the DS1 Remote Agent Experiment. In: Proceedings of the 5th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS-99), ESTEC, Noordwijk, 1999.
8. R. Simmons and C. Pecheur. Automating Model Checking for Autonomous Systems. AAAI Spring Symposium on Real-Time Autonomous Systems, March 2000.
9. B. C. Williams and P. P. Nayak. A Model-based Approach to Reactive Self-Configuring Systems. Proceedings of AAAI-96, 1996.