Formal Verification of Autonomy Models

From Livingstone to SMV

Charles Pecheur[1], Reid Simmons[2], and Peter Engrand[3]

[1] RIACS / NASA Ames Research Center, Moffett Field, CA, U.S.A.
pecheur@ptolemy.arc.nasa.gov
[2] Carnegie Mellon University, Pittsburgh, PA, U.S.A.
reids@cs.cmu.edu
[3] NASA Kennedy Space Center, FL, U.S.A.
peter.engrand-1@ksc.nasa.gov

DRAFT — compiled August 31, 2001

Abstract. To fulfill the needs of its deep space exploration program, NASA is actively supporting research and development in autonomous systems based on model-based reasoning. The verification and validation of such systems poses a tough challenge, due to the huge range of possible situations to be analyzed. We present the application of formal verification techniques to applications based on Livingstone, a model-based health-monitoring system from NASA. We describe a translator that converts the models used by Livingstone into the input language of SMV, a powerful model checking application from Carnegie Mellon. 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. Despite the huge state spaces that they represent (10^{55} states in some experiments), the models can still be exhaustively checked with SMV. We present both Livingstone and SMV and their modeling languages, then detail the translation process. We also give a large account of a pilot experiment using this verification tool in a real space application at NASA Kennedy Space Center.

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 [9].

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 checker, 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 describe 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 presents the Livingstone health management system and its modeling language. Section 3 introduces symbolic model checking and the SMV tool. Section 4 introduces our translator and describes the different translations that it performs. Section 5 presents its use in different NASA applications, and lessons learned from those experiences. Section 6 discusses other possible approaches to verification of model-based autonomy, Section 7 presents some comparable work, and finally Section 8 draws final conclusions.

2 Livingstone

2.1 Overview

Livingstone is a model-based health monitoring system developed at NASA Ames [21]. 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 [14], which generates flexible sequences of tasks for achieving mission-level goals, and the Smart Executive [17], 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,

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

marking the first control of an operational spacecraft by AI software [15]. Livingstone is also used in other applications such as the control of a propellant production plant for Mars missions [5], the monitoring of a mobile robot [20], and intelligent vehicle health management (IVHM) for the X-37 experimental space transportation vehicle.

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 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.

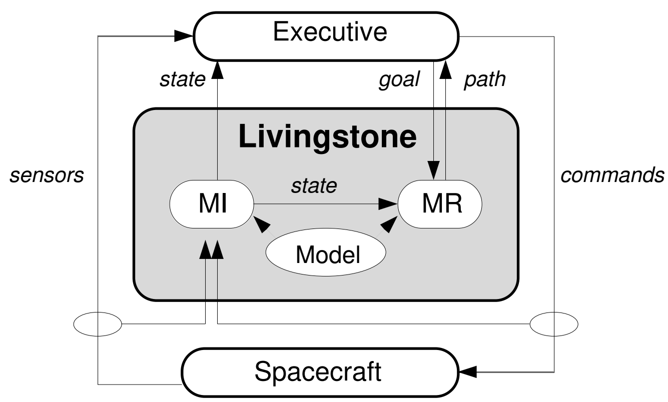

Fig. 1. Livingstone mode identification (MI) and mode recovery (MR)

This work is based on the original Livingstone 1 program that flew on DS-1 as part of the Remote Agent. Livingstone 1 is a Lisp program and uses a Lisp-style language called MPL (Model Programming Language) to describe the models. Livingstone has since then been rewritten from the ground up in C++ to address flight software requirements. The new Livingstone 2 also sports a new Java-like modeling language called JMPL.

2.2 The MPL Language

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. For example, Figure 2 sketches a simple MPL model for a valve, with a parameter `?name` used for naming purposes. It has two attributes `flow` and `cmd`, whose discrete types have been defined beforehand in the two `(defvalues ...)` declarations. It has two nominal modes `open` and `closed` and two fault modes `stuck-open` and `stuck-closed`. Its initial mode is `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`.

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

Components are combined in module declarations, with constraints expressing their interconnections. For example, the declaration in Fig. 3 models two valves `A` and `B` connected in series.

```
(defmodule valve-pair (?name)
  (:structure (valve (A ?name)) (valve (B ?name)))
  (:facts (flow-equal (flow (A ?name)) (flow (B ?name)))))
```

Fig. 3. An MPL Model for two valves connected in series

2.3 Semantics of Livingstone Models

For our purpose, an MPL model reduces to a set of variables and three sets of conditions over the values of those variables, respectively defining the initial states, the static constraints and the transition relation.

Formally, an MPL model defines a finite set X of *state variables* such that each variable $x \in X$ has an associated finite domain D_x. A *state* s assigns to each variable $x \in X$ a value $s(x) \in D_x$. We write S_X for the set of all such states.

The atomic propositions on which all logical conditions are built are variable *assignments* $x = v$, where $x \in X$ and $v \in D_x$. We write Σ_X for the set of all such assignments. A *state formula* is a boolean formula over assignments. State formulae apply to states in the natural way: a state s satisfies $x = e$ iff $s(x) = e$. We write $s \models \phi$ for "s satisfies ϕ" [2], and $[\![\phi]\!]$ for the set of states $\{s \mid s \models \phi\}$.

A *transition rule* is the form $\phi \Rightarrow \mathsf{next}\ \psi$, where ϕ and ψ are state formulae. A *transition formula* is boolean combination of transition rules. Transition rules and formulae serve to define admissible transitions between pairs of states: given states s, s', we say that $(s, s') \models \phi \Rightarrow \mathsf{next}\, \psi$ if and only if, if $s \models \phi$, then $s' \models \psi$.

We can now map an MPL model to an *abstract specification* structure $\langle X, I, C, T\rangle$ where:

- X is the set of *state variables*,
- I is the *initial conditions*, expressed as a set of state formulae,
- C is the *state conditions*, also expressed as a set of state formulae,
- T is the *transition conditions*, expressed as a set of transition formulae.

The sets I, C and T represent conjunctions; for example, we can write $s \models I$ for $s \models \bigwedge_{\phi \in I} \phi$. Abstract specifications are defined compositionally, following the component hierarchy. For each elementary component c, X contains the declared attributes, plus a *mode variable* m_c ranging over the modes of the component. I contains the initial conditions and C contains component constraints and conditional mode-specific constraints of the form $(m_c = m) \Rightarrow \phi$. T contains a single condition of the form

$$(\rho_1 \wedge \ldots \wedge \rho_n) \vee \rho'_1 \vee \ldots \vee \rho'_m$$

with one rule ρ_i of the form $(m_c = m) \wedge \phi \Rightarrow \mathsf{next}\ \psi$ for each transition declared in the component, and one rule ρ'_j of the form $\mathsf{true} \Rightarrow \mathsf{next}\ (m_c = m')$ for each fault mode m'. Intuitively, this formula represents, for each component, the choice between a nominal transition satisfying the transition declarations in the model, and a fault transition to any of the fault modes.

For compound modules, variables and conditions are the union of the respective variables and conditions of their sub-modules, augmented with any additional attributes and state conditions from the module itself.

[2] We omit the obvious natural extension of satisfiability from propositions to formulae. The same remarks holds for transition formulae in the next paragraph.

We then formalize the semantics of an MPL model in terms of a *transition system*[3] $\langle S, S_0, R, L\rangle$ where

- S is a set of states,
- $S_0 \subseteq S$ is the set of initial states,
- $R \subseteq S \times S$ is the transition relation, saying which states can follow each other as time progresses,
- $L : S \to \Sigma_X$ is the labeling function, saying which propositions are true in each state.

Given an abstract specification $\langle X, I, C, T\rangle$ for an MPL model, the transition system $\langle S, S_0, R, L\rangle$ for that model is given by

$$\begin{aligned} S &= S_X \\ S_0 &= \{s \in S_X \mid s \models I \wedge s \models C\} \\ R &= \{(s, s') \in S_X \times S_X \mid (s, s') \models T \wedge s' \models C\} \\ L(s) &= \{x = v \mid s(x) = v\} \end{aligned}$$

As we shall see in Section 4.2, our purpose is to translate MPL models into SMV models in a way that preserves this semantics.

For diagnosis purposes, Livingstone needs two subsets of X to be identified, corresponding respectively to the *commands* (inputs) and *observables* (outputs) of the system. A probability is also associated to each fault mode. In a nutshell, the *mode identification* (MI) problem consists in incrementally generating a (most likely) set of trajectories consistent with known initial conditions and the sequence of command and observable values so far. The *mode recovery* (MR) problem is to find command values that will achieve a given constraint in the next state, given a set of possible current states. A detailed description of how these problems are solved in Livingstone is beyond our scope, see [21] for details.

3 Symbolic Model Checking and SMV

3.1 Introduction

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 [6] and [1] for a general introduction to the theory and practice of model checking.

[3] This kind of transition system is known as a *Kripke structure*, and is the one commonly used in temporal logic theory (see e.g. [6]). In [21], the semantics of Livingstone models is formalized using a slightly different but equivalent structure.

Classical, explicit-state model checkers such as SPIN [10] 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) [2], 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.

3.2 Symbolic Model Checking

We will only sketch the intuition behind BDDs and their use for model checking here. The reader is referred to the literature for a deeper and more formal discussion. [6] has an excellent introductory presentation in chapters 5 and 6.

Binary Decision Diagrams — A BDD is a graph structure used to represent a *boolean function*, that is, a function f over boolean variables $b_1, \ldots, b_n$ with boolean result $f(b_1, \ldots, b_n)$. Each node of the graph carries a variable b_i and has two outgoing edges. It can be read as an *if-then-else* branching on that variable, with the edges corresponding to the *if* and *else* part. The leaves of the BDD are logical values 0 and 1 (i.e. *false* and *true*). For example, a BDD for $f(a, b, c) = a \vee (b \wedge c)$ is shown in Figure 4.

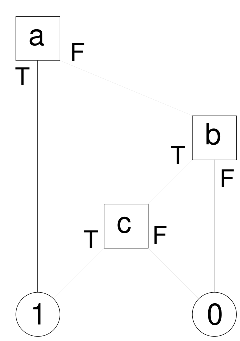

Fig. 4. BDD for $a \vee (b \wedge c)$

By fixing the order of variables in the graph, merging identical sub-graphs and removing nodes with identical children, any boolean function over a given set of variables can be represented as a unique BDD. Furthermore, all needed operations can be efficiently computed over BDDs[4]. For a given boolean function, the size of the BDDs and thus the computation costs are highly dependent on the selected ordering among variables. Finding an optimal ordering is computationally infeasible, and some functions have an exponential BDD size for any ordering. Finding a good re-ordering becomes a critical and tricky issue for large systems.

If the variables $\bar{b} = (b_1, \ldots, b_n)$ represent the state of some system[5], then a set of states S can be represented as the condition $S(\bar{b})$ that those states satisfy, which can be encoded as a BDD. Similarly, a transition relation T over states can be encoded as a BDD $T(\bar{b}, \bar{b}')$ over two generations of the state variables $\bar{b}$ and $\bar{b}'$.

These elements provide the basis for symbolic model checking. For example, given the BDDs $S(\bar{b})$ and $T(\bar{b}, \bar{b}')$, we can compute the BDD $\mathsf{next}(S)$ of the *successors* of S, that is, all states reachable from S through T, as

$$\mathsf{next}(S)(\bar{b}) = \exists \bar{b}'.S(\bar{b}') \wedge T(\bar{b}', \bar{b})$$

By iterating this computation from a given set of initial states until we reach a fixed point, we can compute the reachable state space of the system as a BDD. We can just as easily go backwards and compute the BDD $\mathsf{pred}(S)$ of the *predecessors* of S (from which S is reachable), by using $T(\bar{b}, \bar{b}')$ instead of $T(\bar{b}', \bar{b})$ above. That is the basis for checking CTL temporal logic formulae, as we will see shortly.

Temporal Logic and CTL — Typically, the property to be verified is provided to the model checker as a formula in *temporal logic*. As opposed to propositional or first-order logic, which reasons about a given state of the world, temporal logic allows to reason about how different states follow each other over time. In temporal logic, we can say things like "the engine will never overheat" or "all faults will eventually be detected".

In our case, we focus on the SMV tool that uses *Computation Tree Logic* (CTL). 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. In CTL, temporal operators always associate a branching quantifier E or A to a path quantifier X, U, F or G[6]. CTL operators are summarized in Table 1; their meaning is illustrated in Figure 5 (black and grey circles are states where p and q hold, respectively).

For example, the CTL formula

$$\mathsf{AG}\,(\text{request} \Rightarrow \mathsf{AF}\,\text{response})$$

[4] Needed operations are boolean operations, restrictions of variables and quantification. Good performance is achieved by caching the result of each sub-computation in a hash table.

[5] Note that other finite types, such as enumerations or (small) integer ranges, can be encoded into boolean variables.

[6] Note that F and G can be derived from U, as $\mathsf{F}\,\phi = \mathsf{true}\,\mathsf{U}\,\phi$ and $\mathsf{G}\,\phi = \neg\mathsf{F}\,\neg\phi$

Table 1. Summary of CTL operators

syntax	reads as...	ϕ holds in s if...
$\mathsf{E}\pi$	exists π	π holds on some path from s
$\mathsf{A}\pi$	always π	π holds on all paths from s

a. State formulae ϕ

syntax	reads as...	π holds on $(s_0, s_1, \ldots)$ if...
$\mathsf{X}\,\phi$	next ϕ	ϕ holds in s_1
$\mathsf{G}\,\phi$	globally ϕ	ϕ holds in s_n for all n
$\mathsf{F}\,\phi$	finally ϕ	ϕ holds in s_n for some n
$\phi\ \mathsf{U}\ \phi'$	ϕ until ϕ'	ϕ' holds in s_n for some n, and ϕ holds in s_k for all $k < n$

b. Path formulae π

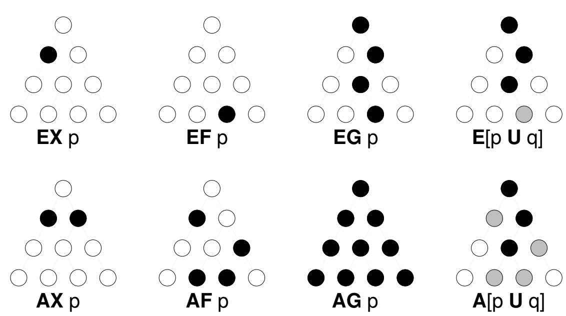

Fig. 5. CTL operators

states that from any state (**A**lways **G**lobally) where there is request, one has to reach (**A**lways **F**inally) a state where there is a response.

BDD-based Model Checking — CTL operators can be characterized recursively; this provides the basis for symbolic model checking of CTL. For example, $\mathsf{EG}\,\phi$ can be defined as the (greatest) solution for Z in the following equation:

$$Z = \phi \wedge \mathsf{EX}\,Z$$

Noticing that $\mathsf{EX}\,Z$ is nothing else than $\mathsf{pred}(Z)$, we can compute the BDD for $\mathsf{EG}\,\phi$ from the BDD for ϕ as follows: starting from $Z^0 = \mathsf{true}$, we repeatedly compute $Z^{n+1} = \phi \wedge \mathsf{pred}(Z^n)$ until we reach a fixed point. That fixed point is the set of states that satisfy $\mathsf{EG}\,\phi$.

3.3 SMV

To analyze Livingstone Models, we use the *Symbolic Model Verifier* (SMV), from Carnegie Mellon University (CMU) [3]. 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 as CTL formulas. 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.

An SMV model is organized as a hierarchy of `MODULE` declarations. A module has a name and optional parameters, and contains a sequence of declarations. There are various types of declarations, among which we list only the ones that we use here[7]:

- `VAR` var_1 : $type_1$; ...; var_n : $type_n$;
  declares each var_i as either a state variable with a finite type (either `boolean`, an enumeration $\{val_1, \ldots, val_m\}$, or an integer range), or a sub-module, where $type_i$ is a module name and parameters.
- `DEFINE` $id_1 := expr_1$; ...; $id_n := expr_n$;
  defines each id_i as a shorthand for $expr_i$. This is a macro mechanism: each $expr_i$ is substituted wherever id_i; no new BDD variable is created for id_i.
- `INIT` *form*
  declares a constraint on initial states. *form* is a boolean expression over state variables.

[7] We use *constraint-style* definitions based on the `INIT/TRANS/INVAR` declarations. SMV also allows *assignment-style* definitions, based on assigning initial and next values to each variable. Assignment-style is the most common and preferred way, but constraint-style is more flexible and adapted when translating from Livingstone models.

- `TRANS` *tranform*
  declares a constraint on transitions. *tranform* is a formula over variables before and after a transition, denoted *var* and `next`(*var*), respectively.
- `INVAR` *form*
  declares a constraint on all states.
- `SPEC` *ctlform*
  declares a property to be verified, as a CTL formula.

The SMV program has to have a module called `main` that is the top level of the hierarchy. The different `INIT`, `TRANS` and `INVAR` constraints combine together across the module hierarchy exactly as in MPL models in Section 2.3, forming three resulting conditions $I(s)$, $T(s, s')$ and $C(s)$. Concretely, SMV computes the BDDs for those three conditions, and then uses them to verify that each `SPEC` declaration is satisfied in all initial states of the model.

4 Verification of Livingstone Models

4.1 Introduction

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. We defer the discussion of other potential approaches to Section 6. For now, let us point out that 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 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, as shown in Fig. 6:

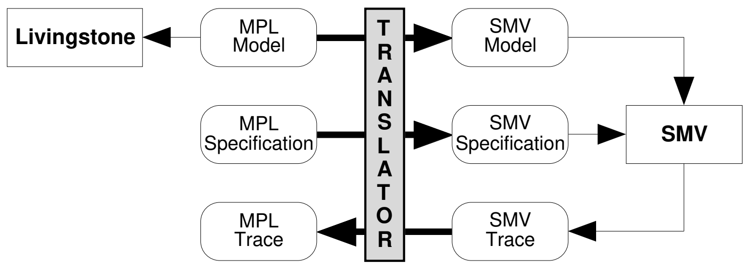

Fig. 6. Translation between MPL and SMV

– 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.

These three aspects are covered by our translator and are detailed in the following sub-sections. The translator program has been written in Lisp[8] and is about 4000 lines long.

4.2 Translation of Models

The translation of MPL models to SMV is facilitated by the strong similarities between the underlying semantic frameworks of Livingstone and SMV. As can be seen by comparing the models in Sections 2.3 and 3.3, both boil down to a synchronous transition system, defined through propositional logic constraints I on initial states, C on all states and T on transitions. Based on this, there is a straightforward mapping from MPL to SMV language elements, summarized in Table 2.

Translation of transition relations has to allow for fault transitions too. For each component, a `DEFINE` declaration is produced that defines a symbol `faults` as the set of its fault modes. Then each MPL transition from a mode `M` to a mode `M'` with condition `P` is translated into the following SMV declaration:

```
TRANS (mode=M & P) -> (next(mode)=M' | next(mode) in faults)
```

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

Table 2. Mapping of MPL to SMV elements

MPL description	MPL syntax	SMV syntax
attribute type	`(defvalues T (V ...))`	`{V, ...}`
elementary component	`(defcomponent C ...)`	`MODULE C ...`
compound module	`(defmodule M ...)`	`MODULE M ...`
component attributes	`(:attributes (A :type T) ...)`	`VAR A : {V, ...}; ...`
sub-components	`(:structure M ...)`	`VAR X : M; ...`
mode transitions	`:transitions ...`	`TRANS ...`
mode constraints	`:model ...`	`INVAR ...`
component constraints	`:facts ...`	`INVAR ...`
initial conditions	`:initial-mode ...`	`INIT ...`

If we compare this to the MPL transition formula

$$(\rho_1 \wedge \ldots \wedge \rho_n) \vee \rho'_1 \vee \ldots \vee \rho'_m$$

of Section 2.3, each ρ_i corresponds to the nominal part

```
(mode=M & P) -> next(mode)=M'
```

while the fault part

```
next(mode) in faults
```

corresponds to the whole disjunction $\rho'_1 \vee \ldots \vee \rho'_m$. On this basis, one can easily see that the conjunction of all `TRANS` declarations of a component is indeed equivalent to the transition formula above.

As an example, Figure 7 sketches the SMV translation of the MPL model in Figure 2.

```
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};
INIT mode=closed
INVAR mode=closed -> flow=off
...
TRANS (mode=closed & cmd=open) ->
        (next(mode)=open | next(mode) in faults)
...
```

Fig. 7. SMV Model of a Valve

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`. Furthermore, MPL models have loose scoping rules: any attribute can be referred to from any other component or module. In contrast, SMV enforces conventional lexical scoping, so attributes that are used out of their scope must be detected and passed along explicitly as parameters of the SMV modules that need them[9]. 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.

4.3 Translation of Specifications

The specifications to be verified with SMV are added to the MPL model using a new `defverify` declaration[10]. The `defverify` declaration also defines the top-level module to be verified. The properties to be verified are expressed in a Lisp-like syntax that extends the existing MPL syntax for logic expressions. For example, the declaration in Fig. 8 specifies a CTL property to be verified with `ispp` as the top-level module. Informally, the specification says that, from any non-fault state, a high flow in valve 1 can eventually be reached. Fig. 9 shows the top-level SMV module that is produced from that declaration, including the conversion of the CTL operators from MPL's Lisp style `(always (globally ...))` into SMV's conventional style `AG ...`.

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

Fig. 8. A sample specification in MPL

CTL is very expressive but requires a lot of caution and expertise to be used correctly. To alleviate this problem, the translator provides different features for expressing model properties.

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

[9] This is appropriate, as SMV passes parameters by reference.

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

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

Fig. 9. Translation of a sample specification in SMV

Specification Patterns — 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 following templates are currently supported:

- `(:specification :inconsistency)`
  Verifies the global consistency of the model. This generates `SPEC 0` (i.e. the always false specification). This specification can only be (vacuously) valid if the model is empty, i.e. if no consistent initial state exists. If the model is consistent, the specification will be false[11].
- `(:specification :reachability)`
  Verifies that all modes of all components are reachable (a variant only checks selected components).
- `(:specification :path-reachability S0 S1 S2 ...)`
  Whenever `S0` is reached, checks the existence of a path reaching `S1`, then `S2`, etc. A variant restricts the path to fault-free states.
- `(:specification :completeness)`
  Verifies whether there is *at least one* enabled transition in any mode of any component, by taking the disjunction of the conditions of all transitions of each mode.
- `(:specification :disjointness)`
  Verifies whether there is *at most one* enabled transition in any mode of any component[12], by considering pairwise conjunctions of the conditions of all transitions of each mode.
  Completeness and disjointness are complementary, together they ensure that the transition clauses are well-formed.

Note that all but the first template are more than syntactic macros, they depend on structural information, such as modes and transition conditions, being extracted from the model.

[11] Note that there is no way to have a *consistency* specification that would fail on inconsistent models, since any specification is vacuously valid on such models.

[12] Note that according to the semantics of MPL models, several enabled transitions would have to be *simultaneously* followed, likely resulting in an inconsistent next state.

Auxiliary Functions — The translator supports some auxiliary functions that can be used in CTL formulas to concisely capture Livingstone concepts such as occurrence of faults, activation of commands or probability of faults. Table 3 gives a representative sample. Some functions are translated solely in terms of existing variables using `DEFINE` declarations, while others, such as `failed`, require the introduction of new variables[13].

Table 3. Some auxiliary functions for MPL model specifications

`(broken heater)`	=	Heater is in a failed state.
`(multibroken 2)`	=	At least two components are failed.
`(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 least of order 3.

The probability analysis opens an interesting perspective. Fault probabilities are mapped to small integers, based on their order of magnitude (rounded if necessary). For example, $p = 10^{-3}$ maps to 3. We then compute the combined order of magnitude for multiple faults by integer addition[14], which SMV can do using BDDs. This is definitely an approximate method, that is no substitute for a precise approach such as Markov chain analysis. It does however provide a convenient way to separate between more and less likely fault scenarios.

Specification Variables — Verification variables can be introduced as attributes declarations in the `(defverify ...)` statement. These attributes work as universally quantified variables in the specifications. This is achieved in SMV with an undetermined constant variable, using the constraint `TRANS x = next(x)` but no `INIT` constraint. The universal quantification then comes as a by-product of the verification for all possible initial states.

Such variables are useful for relating values at different points within a CTL formula. For example, the formula

$$\forall m. \mathsf{AG}\,(\mathsf{send} = m \Rightarrow \mathsf{AF}\ \mathsf{rcv} = m)$$

expresses that any message sent is eventually received. Section 5.1 describes in more details the verification of functional dependency properties, which are based on such verification variables.

Model restrictions — An additional state or transition constraint can be added as a `(:model ...)` clause in the `(defverify ..)` statement. The constraint is simply added to the model. This is convenient for restricting the verification to

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

[14] Since $10^{-a} \cdot 10^{-b} = 10^{-(a+b)}$, assuming independence of faults.

particular conditions, in particular using the auxiliary functions above. One can, for example, limit the number of simultaneous commands or prune out combined faults below some probability threshold.

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.

4.4 Backward 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 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[15].

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 [19].

5 Applications

This section gives a fairly detailed account of the use of our verification tools by Livingstone application engineers in the In-Situ Propellant Production (ISPP) project at NASA Kennedy Space Center (KSC). It summarizes a more detailed presentation in [8]. 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. Another application, focusing on the Deep Space One spacecraft, is also briefly described. We conclude with some comments on the nature of the verification problem for autonomy models, and a short discussion of alternative model checking tools that have been or could be used.

[15] As of this writing, translation of traces is still to be implemented in our translator. It poses no fundamental technical difficulty but requires some effort in parsing SMV's output trace.

5.1 In-Situ Propellant Production

Since 1998, the NASA Kennedy Space Center (KSC) has been working with the Livingstone developers at NASA Ames to apply Livingstone to the In-Situ Propellant Production (ISPP) plant, a system intended to produce spacecraft propellant using the atmosphere of Mars [5]. This plant, built in KSC's Applied Chemistry Laboratory, is capable of producing a large amount of oxygen with a small quantity of seed hydrogen, based on the Reverse Water Gas Shift reaction (RWGS) (Figure 10). In a human Mars mission, this plant would be required to operate for 500 or more days without human intervention. The Livingstone monitoring and diagnosis software provides built-in autonomy capabilities for ISPP.

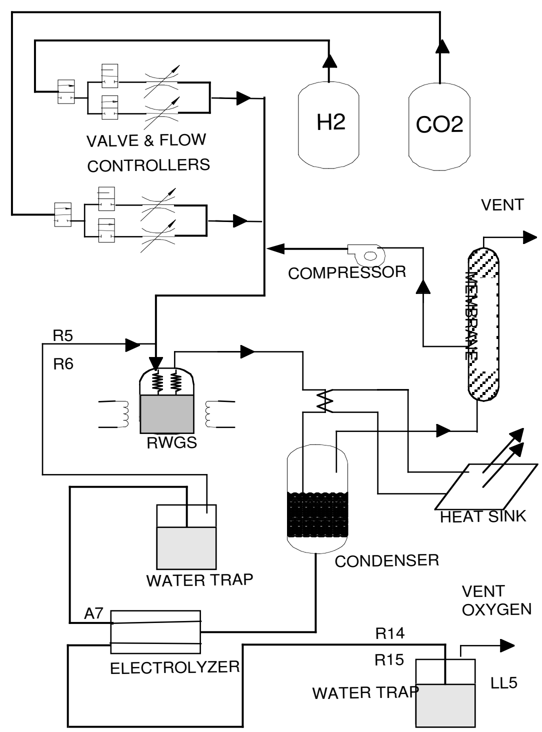

Fig. 10. RWGS process for ISPP

Verification Methodology — One of the fundamental issues for the effective verification of models deals with the construction of meaningful specifications that validate the model (where a rough and ready definition of "meaningful" is a temporal form which captures some global system behavior, such as something stated as a high level design specification).

Our effort focused on the observation of modeling error "phenomena" (i.e. observable effects of those errors) and then the investigation of the underlying causes of the errors phenomena so that specification properties could be inferred that could lead to future re-usable specification patterns. These observations consisted of two modeling errors which had been previously encountered and diagnosed by the modelers. Nevertheless, it should be stressed that as a result of the investigation of the second error, two additional modeling errors were detected which had gone unnoticed during the normal testing of the model by the developers.

First Error — The first error to be investigated dealt with a mis-modeled fluid flow behavior[16]. The model involves a component called a *sorption pump*, which absorbs the carbon dioxide (CO2) from the atmosphere, and a complex set of flow branches through which that CO2 is then sent for chemical transformation. For each branch, the model defines its admittance[17] and the flow that traverses it, both in terms of qualitative values {`off`, `low`, `high`}. Model constraints capture the relations between the different flows and admittances.

The modeling error occurred when the developers omitted the constraint that ties the global flow to the global admittance and the CO2 pressure in the sorption pump. This omission led to the model manifesting flow through a flow branch with a zero admittance (e.g. with a closed valve on the path).

In order to exhibit this error, we stated in CTL the property that *if the outlet valve of the sorption pump is blocking, then there is no flow through the branch.* In the MPL model, this was expressed as:

```
(all (globally (implies (off (admittance outlet))
                        (off (flow z-flow-module)))))
```

When the model and specification were translated and processed by SMV, a counter-example was returned, showing a violating state where `(flow z-flow-module)` was `high` although `(admittance outlet)` was `off`. The model developers acknowledged that that information would have likely allowed them to isolate their error much more quickly.

Second Error — The second error was related to modeling flows and flow differences in a component `rwgs`. A set of constraints aims at defining an attribute `(relative-flow rwgs)` as the (qualitative) difference between the incoming and outgoing flows `(flow-in rwgs)` and `(flow-out rwgs)`. As the result of an oversight during code re-use, the following constraint fragment was used

```
(when (on (flow-in O2))
      (high-in (relative-flow rwgs)))
```

[16] This experiment was performed on the model of an earlier version of the ISPP system, based on the so-called Sabatier-Electrolysis chemical process.

[17] Admittance is a measure of fluid impedance of pipes and orifices.

where `O2` should have been replaced by `rwgs`. The result found by the developers during their normal testing was an anomalous behavior of the model, where `rwgs` was filling up while the `O2` component connected to it was also filling up. Though the error seems obvious, this "mis-naming" proved to be difficult to detect both using testing by the developer and model checking with SMV.

Functional Dependency — We used this second error as an opportunity to study and demonstrate the verification of *functional dependency* properties using SMV. We say that a variable y functionally depends on variables $x_1, \ldots, x_n$ if and only if any given values for $x_1, \ldots, x_n$ uniquely determine the value of y. In practice, y could in this case be defined as a function of $x_1, \ldots, x_n$, but Livingstone provides no way to do this. Instead, the dependency has to be expressed as general propositional constraints, a potentially error-prone task.

We can express functional dependency of y w.r.t. $x_1, \ldots, x_n$ using CTL with universally quantified variables as

$$\begin{aligned}\forall v_1, \ldots, v_n, v.\ &(\mathsf{EF}\ x_1 = v_1 \wedge \ldots \wedge x_n = v_n \wedge y = v) \Rightarrow \\ &(\mathsf{AG}\ x_1 = v_1 \wedge \ldots \wedge x_n = v_n \Rightarrow y = v)\end{aligned}$$

In other words, if for some values v_i for each x_i you get some value v for y, then for the same v_i you always get the same v.

In the case of the second error above, `(relative-flow rwgs)` was meant as a function of `(flow-in rwgs)` and `(flow-out rwgs)`, so we added verification variables `some-flow-in`, `some-flow-out` and `some-relative-flow` and checked the property

```
(implies
  (exists (finally
    (and (= (flow-in rwgs_trap) some-flow-in)
         (= (flow-out rwgs_trap) some-flow-in)
         (= (relative-flow rwgs) some-relative-flow))))
  (always (globally
    (implies
      (and (= (flow-in rwgs_trap) some-flow-in)
           (= (flow-out rwgs_trap) some-flow-in))
      (= (relative-flow rwgs) some-relative-flow)))))
```

SMV reported this property as false, as expected, but some further steps were needed to get a meaningful diagnosis. Indeed, the property amounts to a disjunction[18], so a complete counter-example has two sides, one for each half of the property, giving the two different values of `(relative-flow rwgs)` for the same `(flow-in rwgs)` and `(flow-out rwgs)`. However, by design SMV only gives a single counter-example trace, corresponding to the first half, in this case:

```
(off (flow-in rwgs_trap))
(off (flow-out rwgs_trap))
(equal (relative-flow rwgs))
```

[18] As $(\mathsf{EF}\ \phi) \Rightarrow (\mathsf{AG}\ \psi)$ is equivalent to $(\mathsf{AG}\ \neg\phi) \vee (\mathsf{AG}\ \psi)$.

To obtain the second half of the counter-example, the second half of the property was instantiated as a separate specification using the values from the first counter-example:

```
(always (globally
  (implies
    (and (off (flow-in rwgs_trap))
         (off (flow-out rwgs_trap)))
    (equal (relative-flow rwgs)))))
```

Verification of this new specification generated the second counter-example pinpointing the problem exposed above:

```
(off (flow-in rwgs_trap))
(off (flow-out rwgs_trap))
(high-in (relative-flow rwgs))
```

Although we finally achieved our goal of detecting and identifying a functional dependency violation, the method to follow is fairly involved and cumbersome. Functional dependency properties are prime candidates for a better automated support in the translator. Complete support will require more than specification patterns, though, as results have to be extracted from the first SMV run to prepare for the second one.

Experimental Notes — The size of the RWGS Model used which contained the second error is 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 RWGS Model is of the order of 10^{55} states, and can still be processed in a matter of minutes using an enhanced version of SMV [22]. 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. This adds further evidence that writing a correct temporal specification can be a subtle an error prone task, part of which consists in verifying the specification itself in order to check that it truly conveys the property one wants to check for.

5.2 Deep Space One

Livingstone was originally developed to provide model-based diagnosis and recovery for the Remote Agent architecture on the DS1 spacecraft. It is also in this context that the translator to SMV was originally developed. 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. 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) [16].

5.3 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, faults 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 faults 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.

Because Livingstone models are so loosely constrained, they are also hit head-on by the combinatorial state space explosion, as the multiple configurations of each component essentially multiply with each other. This leads to huge state spaces (10^{55} for the latest ISPP model), that preclude any explicit state enumeration approach. For BDD-based symbolic processing, though, the complexity is rather related to the logical complexity of the model, and the ones we have seen so far fit (sometimes barely) within reach of the available tools. In particular, SMV needs only a few fixpoint iterations for each property because the models are so shallow.

This also affects the kind of properties that are useful to verify. Looking for deadlocks makes no sense in the presence of spontaneous fault 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.

5.4 Alternative Model Checkers

Most of the experiments with the translator presented here have used the SMV package from Carnegie Mellon to perform the verification itself. However, SMV has reached the limits of available memory (256Mb in most experiments) with larger models, which has forced us to consider and try different improvements and alternatives. We list the most significant ones here.

Re-ordering Variables — As explained in Section 3.3, BDD size depends critically on a fixed ordering among model variables, and SMV provides options to optimize that ordering. We had to use this technique on a first-generation model of ISPP with 10^{50} states. After some trial and error, we captured an optimized ordering that lead to successful verification of that model.

New SMV versions — A new, upward-compatible version of SMV, called *NuSMV* [4], has been recently 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. A couple quick tests with NuSMV confirmed a significant speedup but did not improve the memory performance, which has been the limiting factor in our applications.

Elimination of Variables — Certainly the most successful action towards scaling up to more complex models has been the adoption of an enhanced version of SMV developed by Bwolen Yang [22]. This version performs an analysis of the state constraints in the model to detect variables that can be expressed in terms of others, in the same sense as the functional dependency in Section 5.1. When such a variable is found, it is replaced by the corresponding expression and removed from the model, resulting in a reduction of the BDDs. This works out particularly well on Livingstone models, which are rich in state constraints expressing functional dependencies. On the latest model, this simplification reduced the BDD depth by more than a third, from 588 down to 379 binary variables. As a result, reachability of all modes, which exhausted memory with the standard SMV, completed in 29 seconds with the enhanced version.

Bounded Model Checking — The symbolic model checking community has shown a recent interest in *bounded model checking* using SAT solvers[19] instead of BDDs, such as the BMC (Bounded Model Checker) tool from Carnegie Mellon. BMC gives performs only an approximate verification, as the model is explored only down to a given depth, but never suffers from exponential space blow up, finds shortest counter-examples and is quite fast in practice (though NP-complete in theory). While we have not had the opportunity to try BMC on Livingstone models, this seems a promising approach: as the models are expected to be shallow, the limitation of depth should not be too penalizing. Indeed, an important part of an average Livingstone model resides in the static state constraints, and many interesting properties, such as functional dependencies, could be analyzed statically on those constraints using standard SAT techniques.

6 Comparison of Verification Approaches

The functionality of Livingstone results from the interaction between multiple parts of a quite different nature:

- The *Livingstone engine* that performs model-based reasoning for diagnosis purposes,
- The *Livingstone model* that provides domain-specific knowledge to the engine,
- The *plant* or physical system that is being diagnosed, including the system software layer that provides the abstraction used by Livingstone,

[19] SAT solvers implement decision procedures for propositional logic.

– The rest of the controller, including the *executive* that drives the plant and interprets diagnoses, and the human or automated environment (such as a planner/scheduler) that feeds commands to the executive.

It is clear that verifying the proper operation of Livingstone in such a context is a complex task that will require more than one method. In this section, we want to discuss the merits and limits of different methods.

Verification of Models — In Section 4, we have seen that Livingstone models lend themselves quite nicely to formal, exhaustive analysis using symbolic model checking. In doing so we should not lose sight, however, that we are not verifying a diagnosis engine but a general dynamic model of the plant. We can check whether that model has some desirable structural properties and matches our characterization of its physical counterpart, but not, in general, whether its faults will be properly diagnosed. Yet, some necessary conditions for diagnosis can be verified on the model, such as whether different faults are always differentiable based on a given set of observables.

Ideally, model-based reasoning would always infer the right diagnosis from a correct model, but in practice, performance tradeoffs may preclude that, or there could not be enough information to discriminate between different diagnoses. Thus, model-based verification does not alleviate the need for testing the model within the actual Livingstone algorithm.

Verification of the Engine — The engine is a relatively compact piece of software performing complex artificial intelligence algorithms. It is a sequential, deterministic program but it manipulates large and dynamic data structures. Doing model checking on the engine alone would make little sense, as it has a huge but mostly data-related state space; some form of formal proof of its constituting algorithms would make more sense. Nevertheless, the engine is also a 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. From the point of view of its users, Livingstone should be viewed as a stable, trustable part, just as C programmers trust their C compiler.

Verification of the Application — As explained above, we still want to verify a Livingstone application as a whole, including the engine, the model and its environment (controller and plant), or at least some abstract simulation of its relevant parts.

We could, in principle, model all that for the purpose of model checking. It would have to include a specification of the Livingstone engine algorithms and its data structures. In particular, the Livingstone model itself would appear not as a transition system but as a huge data structure used by the engine. Producing such a specification would be an arduous and error-prone task. Furthermore, the size of the data structures involved would likely render the model hopelessly intractable.

An alternative solution, that we are currently investigating, is to perform a controlled execution of Livingstone within a testbed, both instrumented to allow backtracking and state comparisons. Using this technique, a wide range of behaviors can be automatically and efficiently explored, using the same kind of state space exploration algorithm as used in explicit state model checkers.

7 Related Work

Initial experiments in applying model checking to autonomy models were performed by Penix, Pecheur and Havelund in 1998 [18] and focused on the HSTS model-based planner/scheduler [14]. HSTS models are composed of a large number of tightly coupled declarative real-time constraints whose combined effects are difficult to apprehend. The experiments were done on a small sample model of an autonomous robot. A useful subset of the planner modeling language, omitting the real-time aspects but covering the robot example, was translated into the input syntax of three mainstream model checkers: SMV [3], Spin [10] and Murphi [7]. No translation tool was built, but the translations were done by hand in a systematic way amenable to automation. The analysis identified several flaws in the model that were not identified during testing. Out of the three model checkers used, SMV allowed the most direct translation and was by far the fastest: 0.05 seconds vs. approx. 30 seconds for Spin or Murphi. The good performance of SMV can be attributed to similar factors as for Livingstone models: easy translation of declarative constraints and large loosely coupled models. Planner models also contain timing information that was ignored here since untimed model checkers were used.

In a continuing effort, Khatib [11,12] is exploring the translation of HSTS models to UPPAAL [13]. UPPAAL is a real-time model checker, and is therefore able to deal with the timing aspects of HSTS models. UPPAAL has been used successfully to verify properties on a timed version of the same robot model as used in the earlier experiments.

8 Conclusions

Our MPL to SMV translator will allow 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. Although SMV has been pushed to its limits a couple times, recent improvements such as variable elimination, or new avenues such as SAT-based model checking, will allow to push those limits even further. 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. 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.
2. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
3. 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, June 1992.
4. 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.
5. 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*, November 1999.
6. E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. Mit Press, 1999.
7. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.
8. Peter Engrand. Model checking autonomy models for a martian propellant production plant. In *Proceedings of the AAAI Symposium on Model-Based Validatin of Intelligence*, March 2001.
9. 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 the 50th IAF Congress*, October 1999.
10. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

11. L. Khatib, N. Muscettola, and K. Havelund. Verification of plan models using UPPAAL. In *Proceedings of First Goddard Workshop on Formal Approaches to Agent-Based Systems*, April 2000. To appear in Lecture Notes in Computer Science, Springer Verlag.
12. Lina Khatib, Nicola Muscettola, and Klaus Havelund. Mapping temporal planning constraints into timed automata. In *Eigth International Symposium on Temporal Representation and Reasoning, TIME-01*. IEEE Computer Society, June 2001.
13. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal of Software Tools for Technology Transfer*, 1(1-2), 1997.
14. N. Muscettola. *HSTS: Integrating planning and scheduling*. Morgan Kaufman, 1994.
15. 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.
16. 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.
17. 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.
18. J. Penix, C. Pecheur, and K. Havelund. Using model checking to validate AI planner domain models. In *Proceedings of the 23rd Annual Software Engineering Workshop*, December 1998.
19. R. Simmons and C. Pecheur. Automating model checking for autonomous systems. In *AAAI Spring Symposium on Real-Time Autonomous Systems*, March 2000.
20. 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.
21. B. C. Williams and P. P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of AAAI-96*, 1996.
22. 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.