Practical Formal Verification of Diagnosability of Large Models via Symbolic Model Checking

Roberto Cavada

Charles Pecheur

RIACS Technical Report 03.03

January 2003

Activity Report

Practical Formal Verification of Diagnosability of Large Models via Symbolic Model Checking

Roberto Cavada, ITC-IRST

cavada@irst.itc.it

Charles Pecheur, RIACS

pecheur@email.arc.nasa.gov

RIACS Technical Report 03.03

February 2003

Activity Report

This document reports on the activities carried out during a four-week visit of Roberto Cavada at the NASA Ames Research Center. The main goal was to test the practical applicability of the framework proposed by Pecheur and Cimatti, where a diagnosability problem is reduced to a Symbolic Model Checking problem.

This work was supported by NASA ECS Task 2.2.1.1 “Validation of Model-Based IVHM Architectures”, under Cooperative Agreement NCC 2-1006 with the Universities Space Research Association (USRA) and under Contract NAS2-00065 with QSS Group, Inc. Roberto Cavada's visit at NASA Ames was supported by QSS Group, Inc., under a Training Exchange Program from AIPT.

Contents

1 Introduction

This document reports on the activities carried out during a four-week visit of Roberto Cavada at the NASA Ames Research Center. The main goal was to test the practical applicability of the framework proposed in [1], where a diagnosability problem is reduced to a Symbolic Model Checking problem.

Section 2 contains a brief explanation of major techniques currently used in Symbolic Model Checking, and how these techniques can be tuned in order to obtain good performances when using Model Checking tools.

Diagnosability is performed on large and structured models of real plants. Section 3 describes how these plants are modeled, and how models can be simplified to improve the performance of Symbolic Model Checkers.

Section 4 reports scalability results. Three test cases are briefly presented, and several parameters and techniques have been applied on those test cases in order to produce comparison tables. Furthermore, comparison between several Model Checkers is reported.

Section 5 summarizes the application of diagnosability verification to a real application. Several properties have been tested, and results have been highlighted.

Finally, section 6 draws some conclusions, and outlines future lines of research.

2 Symbolic Model Checking

This section briefly describes the two main techniques currently used in Symbolic Model Checking: *BDD-based* and *SAT-based* approaches. These techniques are both implemented in *NuSMV*, the New Symbolic Model Verifier developed at ITC-irst ([5]).

2.1 The BDD-based approach

Many Symbolic Model Checkers are currently based on *Binary Decision Diagrams*. BDDs are data structures that allow a compact and canonical symbolic representation of sets of states. Further information about BDD-based symbolic model checking can be found in [5, 1].

BDDs sometimes require a lot of tuning in order to be able to deal with large space models. Here are the most important techniques and parameters that can be used with BDDs:

Variables ordering This is the most important factor which can dramatically improve the BDD-based approach. *NuSMV* allows to dynamically reorder variables during the model construction. A good strategy consists in interactively stopping the model construction and forcing a reordering after some time, then restarting the construction using the saved ordering. The process is reiterated until the construction successfully completes within acceptable time.

Dynamic reordering In this case, the reordering is performed automatically by the model checker when BDD grows beyond some threshold size.

Building method There are essentially two building methods: *Monolithic* and *Conjunctive Partitioning*. *Conjunctive Partitioning* divides large BDDs into several pieces, which often improves overall performance. The *Conjunctive Partitioning* building method uses a threshold BDD size and a couple other parameters (see [4],[9]).

Variable dependencies detection This technique, due to Bwolen Yang, searches for unique dependencies between variables. When a dependency is found, one variable is removed and substituted by the equivalent expression. (For further details see [7, 8].) *smv-2.4b* is a modified version of the original SMV tool from CMU that implements this advanced technique, which in this context can lead to higher performances when constructing the BDD representation. Currently *NuSMV* does not provide this functionality.

Reachability calculation This technique calculates the set of reachable states (as a BDD) before performing the verification. This helps the verification phase, not the model construction phase. Unfortunately the construction phase is the most painful and critical phase in this verification context.

2.2 The SAT-based approach

The SAT-based approach does not use BDDs to construct the model. The model and the property to be verified are instead converted in a time-bounded propositional problem by the Bounded Model Checker. Then this propositional problem is submitted to a SAT solver, which searches for a possible set of assignments that *satisfies* the given problem.

The SAT-based approach is basically oriented towards bug detection, but it also allows a induction-based approach, which can be used to validate invariance properties. For more general properties, only LTL is currently allowed. This means that the bug-detection for properties that contain the existential quantifier is not allowed.

3 Structure of the models

We apply model checking to models used in NASA Ames' Livingstone diagnosis system. Livingstone confronts these models against observed response from the *plant* (i.e. the physical system being diagnosed) to estimate states and diagnose faults.

Livingstone models are written in JMPL (Java-based Model Programming Language). JMPL is a modeling language designed to be easier to use for modelers, and it is well-integrated in the Livingstone diagnosis system (see [6]).

3.1 JMPL models, and conversion into SMV models

Plants are modeled in JMPL, as a hierachy of modules and components. The plant model can be split across several JMPL source files, typically with one file for each module and component. The complete plant is denoted by the highest module in the hierarchy. Visible variables (commands and observables) are listed in a separate file, named the *Harness file* (*hrn* from here on).

In order to be verified by a SMV-based model checker (*smv-2.4b* and *NuSMV* are two examples), the JMPL source files set must be compiled into one or more SMV files. This operation is carried out by the *jmpl2smv* tool (see [6]). This compiler gets the JMPL sources files, the hrn, a set of options, and produces a single SMV source file.

One important option (*-twin*) allows to compile either a single model, or a twin model, as used for verification of diagnosability in [1].

3.2 Structure of JMPL models

Models are constituted by hierarchically connected modules. Terminal nodes of the hierarchy are *Components*. Each component has a private state and a possible visible state (a subset of the interface of the component). A component also has an initial state, which can be partially non-deterministic. Components have a set of static constraints and a transition relation. Constraints can be classified as follows:

- Connecting invariants. These are used to connect the module interfaces. If two components A and B are logically connected, then a part of the space of each one will be used to realize the connection. For example A could contain variable x, and B the variable y, and the invariant `x=y` could connect A and B.
- Behavior constraints. These are invariants on the physical behavior of the model.

As explained in [1], the *twin model* is obtained by instantiating two identical copies of the model, and then constraining their visible parts to be identical.

3.3 Structure of SMV models

Several techniques have been used in order to reduce the state space of the generated SMV models, and to improve model checker performances.

What follows here is an example of single SMV model:

```
MODULE M1
  VAR x : boolean; -- a visible variable
  VAR y : M2;      -- a contained component
  -- ...

MODULE M2
```

```
  VAR x : boolean; -- another visible variable
  -- ...

MODULE main
  VAR test : M1;
  -- ...
```

3.3.1 Connecting variables and invariants removal

As already explained, in single models components are connected by specifying invariants that equate variables of the connected components. Model checking performance can be improved by replacing these two equal variables by a single one.

We realized a script which searches for connecting invariants, finally removing them and any associated redundant variable. The script searches for invariants in the form `M.x = N.y`, where `x` and `y` variables, and `M` and `N` are to generic hierarchy paths. When one is found, it is removed, and the declaration for one of the variables, say `M.x`, is replaced by a macro definition referring to `N.y`. In effect, the macro makes `M.x` equivalent to `N.y` throughout the model.

Note that Yang's *smv-2.4b* automatically detects dependent variables, including equal ones, and performs a similar simplification at the BDD level. This explains the remarkable performance improvements it gives on Livingstone models (see [6]).

3.3.2 Twin models

Basically a *twin* model is a pair of top-level modules, whose visible parts are constrained to be equal. Here is how twin models are generated by *jmpl2smv*.

```
MODULE M1
  VAR x : boolean; -- a visible variable
  VAR y : M2;      -- a contained component
  -- ...

MODULE M2
  VAR x : boolean; -- another visible variable
  -- ...

MODULE twin
  VAR test : M1;

MODULE main
  VAR test  : M1;
  VAR _twin : twin;
```

The convention is that, given a single top-level SMV module named `test`, the twin model contains two copies of that module named `test` and `_twin.test`.[1]

3.3.3 Compacting visible parts

In twin models the visible parts must be constrained to be identical. Specifically, commands and sensors' values must be the same on both sides. One way to do this is to add a constraint that makes them equal, but a more efficient solution is to merge them into a single variable shared between the twins.

Let us call the two sides in the twin couple as `L` and `R` (Left and Right, corresponding to `test` and `_twin.test` above). All the visible parts can be extracted from both the `L` and `R` models, and put into a separate hierarchy of modules that is shared between the twin models. Here is the resulting twin model, where `M1_visible` etc. contain the shared variables:

```
MODULE M1_visible:
  VAR x : boolean; -- a visible variable
  VAR y : M2_visible; -- a visible contained component

MODULE M1(visible)
  DEFINE x := visible.x; -- removed visible variable
  VAR y : M2(visible.y); -- a contained component

MODULE M2_visible
  VAR x : boolean; -- another visible variable

MODULE M2(visible)
  DEFINE x := visible.x;
  -- ...

MODULE twin(visible)
  VAR test : M1(visible);

MODULE main
  VAR test_visible : M1_visible;
  VAR test  : M1(test_visible);
  VAR _twin : twin(test_visible);
```

In the last example the visible parts are now shared between the `L` and `R` sides. This is a systematic method to reduce the state space in twin models.

We realized a script which performs this systematic transformation automatically. The script gets the SMV source of the twin model and the hrn as input, and produces a new simplified SMV source as output.

[1] This asymmetry allows to keep compatibility with single-model specifications, and easily transpose variables names to the twin model by prefixing with `_twin`.

3.3.4 Possible heuristics for extracting initial variable ordering

Models are populated with a very large number of invariants. In particular, models *X-34* and *PITEX* contain in the order of 10^3 invariants. Model checkers spend most of the time in the model construction, and in particular, for our test cases, in constructing the invariants. We know that the construction time for the BDD-based approach strongly depends on the variable ordering, so we tried to statically extract and use some information about a possible better ordering from invariants in the form:

```
V0 -> V1
V0 -> V2
...
```

A simple heuristic would make V0, V1 and V2 closed in the ordering file. We implemented a script that generates a simple ordering file by using this heuristic. But results are far long from an optimal ordering, because each variable can be referenced by more than one invariant, so there is the problem of deciding where a given variable should go.

4 Experimental evaluation

Three main models have been used for experimental evaluation:

RWGS An In-Situ Propellant Production Plant (see [2].)

X-34 The Main Propulsion System for a next-generation Reusable Launch Vehicle (see [10])

PITEX A more advanced and complex version of X-34.

These are models of large, complex and structured plants. The main verification activity centered the attention on the X-34 model. RWGS and PITEX models have been used only for scalability analysis. The *RWGS* and *X-34* models have been used to produce these comparison tables.

This section compares the *NuSMV* and *smv-2.4b* model checkers, both using the *BDD-based* approach. For *NuSMV* different BDD parameters have been tried. A table also compares the *BDD-based* and *SAT-based* approaches by using *NuSMV*.

As will be shown, the *BDD-based* approach failed when dealing with such large coupled systems. On the other hand, the *SAT-based* approach proved to be easily able to deal with even the largest *PITEX* model, using a very reasonable amount of resources.

Furthermore, the particular kind of properties required by the diagnosability verification (reachability), allows to express those properties in LTL, which is fully supported by the *BMC* module inside *NuSMV*.

4.1 Tuning the BDD package for *NuSMV*

Table 1 gives an idea on the actual impact the use of a good ordering file can have on both building and checking performances. Benchmarking has been carried out on the *RWGS* example, in a single (not twin) configuration.

1. We obtained a first construction without re-ordering. This phase took a very long time.
2. We generated a first ordering, by using the *sift* algorithm. The obtained ordering is labeled *ord1*. This phase took a relatively long time as well.
3. We generated a slightly better ordering by using the *sift_converge* algorithm. The obtained ordering is labeled *ord2*.

Times are given in seconds, while memory consumption is measured in mega byte. The verification time refers to the time spent on verifying 20 properties in the form "EF p".

Method	Var Ordering	Building		Checking	
		Time (sec)	Space (MB)	Time (sec)	Space (MB)
Monolithic	–	1016	179	165	0
Monolithic	*ord1*	22	26	9	0
Monolithic	*ord2*	14	24	6	0
Threshold	–	1020	179	165	0
Threshold	*ord2*	16	24	7	0

Table 1: Model building and checking statistics on the *RWGS* example

Table 1 highlights the great improvement that a good ordering file can actually give to model checker performances. The second row shows that the building time is dramatically reduced by using the *ord1* ordering. Before model construction, 900 seconds have been spent to obtain *ord1*. Third row shows that a further improvement can be obtained by calculating a better ordering (*ord2*). The variables ordering calculation required 25 seconds, but by using *ord2* the building time reduced from 22 to 14 seconds.

4.2 *BDD* vs *SAT*, and *NuSMV* vs *smv-2.4b*

Model building (or construction) has revealed to be the most critical phase when dealing with this kind of large system. We compared how the BDD-based and the SAT-based approaches perform during the building phase.[2]

[2] Actually, the SAT solver itself is not involved in this phase, rather, the *Bounded Model Checker* (*BMC*) generates the set of constraints to be solved by SAT. In that sense, we may be better speaking about BMC vs BDD approaches.

For the BDD approach, both *smv-2.4b* and *NuSMV* model checkers are competitors in the comparison. Table 2 shows how much the *Variable Dependencies Detection* technique implemented into *smv-2.4b* can improve performances.

Regarding the BMC-based construction time, we should point out that it is strongly influenced by invariants that can be added in order to enforce a diagnosability property.

Test-case	Twin	Coupled	*smv-2.4b* (BDD)	*NuSMV* (BDD)	*NuSMV* (BMC)
RWGS	No	No	3/351	16/24	1/15
RWGS	Yes	No	N/A	N/A	N/A
RWGS	Yes	Yes	M	T	1/26
X-34	No	No	10/370	T	3/30
X-34	Yes	No	M	T	3/30
X-34	Yes	Yes	M	M	1/17
PITEX	No	No	241/396	N/A	2/26
PITEX	Yes	No	N/A	N/A	9/50
PITEX	Yes	Yes	N/A	N/A	2/28

Table 2: Model construction time/memory with different approaches and tools (in sec/MB)

In Table 2, 'M' means that the model checker could not build the model because of size limit (over 1.5 Gb), while a 'T' means that the timings limit has been reached (24 hours). 'N/A' means that the test has not been carried out, either because the source was not available, or because we were not interested in observing the result. In particular, this is the case for the PITEX model, which we know is larger than the X-34 model. During this phase we partially used the PITEX model only for scalability analysis for the SAT-based approach.

5 Diagnosability of real models with *NuSMV*

5.1 Testing scenarios

We derived diagnosability properties from documented test scenarios used for testing the Livingstone application (see [10]). Each scenario has one or more failing components. The model describes the propellant (fuel and oxidizer) feed system of a space vehicle (tanks, pressurization circuits, etc). In all the cases we used, the failed components are stuck valves. A test is passed if the Livingstone system is able to report the correct fault, by choosing over a set of possible candidates (see [11], and [6]).

Table 3 lists the scenarios we have taken into account. The rank is the negated logarithm of the failure probability—in other words, higher ranks correspond to less likely faults. In this context, we are operating in a *fault detection* diagnosability problem (see [1]).

Scenario	Component	Failure	Rank
1	PV03	Stuck Closed	2
2	SV33	Stuck Closed	2
6	SV31	Stuck Open	4
11	SV31	Stuck Closed	2
9	VR01	Stuck Open	3
13	VR01	Stuck Closed	1

Table 3: Scenarios list

5.2 From scenarios to LTL and invariants properties

Let's call `L` and `R` the two coupled models used for verification, as already explained in section 3.3.3. Given a condition `cond` on the state of the system, we can express that `cond` is diagnosable by the invariant:

```
(L.cond -> R.cond)
```

which we verify by model checking the corresponding universally quantified LTL form `G (L.cond -> R.cond)`. For example, for the Scenario 1:

```
G (L.pv03.pv.mode=stuckClosed -> R.pv03.pv.mode=stuckClosed)
```

If the verification of this LTL property produces a counterexample, we found a case in which the property `pv03.pv.mode = stuckClosed` is not diagnosable.

5.3 Incremental approach for context enforcement

Plant models represent the physical components, not how they are operated. Operational conditions further restrict the possible behaviours. Even within that envelope, the diagnosis of a specific component may be critical or even relevant only under even more specific operational conditions.

If the model checker finds a counterexample for a given diagnosability property, the trace must be carefully analyzed in order to distinguish between real bugs in the model or in the plant, and bogus cases due to obvious, trivial or unlikely behaviors.

By defining a *context* we can again verify the property, and reiterate the process, looking for a real problem, or proving the refined diagnosability property.

This context-enforcement can be achieved by specifying a constraint on:

- the initial state, or
- the transition relation, as an invariant property, or
- the property itself, as an additional condition (either propositional or, in more complicated cases, involving temporal operators).

It is important to remark here that the supplied trace may denote a very complicated case, whose interpretation might require considerable expertise in the application being verified.

5.4 Results

The first attempt consisted in trying to verify all scenario properties, without specifying any context. We obtained the expected result: all properties were false, and a 2-step counter-example was found for each scenario. These were trivial results involving multiple simultaneous failures; obviously out of the operating envelope.

The next step consisted in centering the attention only on a few scenarios. We put out attention on two cases, which are described in the next sections. The most relevant result we obtained is described in the section regarding *Failure Scenario 9*.

5.4.1 Failure Scenario 1: PV03 stuck closed

This scenario checks if diagnosis can tell whether PV03 is open or stuck closed. Condition *stuck open* vs *stuck closed* is not taken into account, since we force one model to operate in a nominal context (see below for context description).

One or more failures vs. no failure: In this case the context is given from this invariant:

```
INVAR _twin.test._brokencount < 1
```

The property to be verified is:

```
G (test.pv03.pv.mode=stuckClosed -> !_twin.test.pv03.pv.mode=open)
```

Which results to be *true* by using the induction-based approach.

5.4.2 Failure Scenario 6: SV31 stuck open

This scenario checks if diagnosis system can detect whether SV31 is closed or stuck open.

The property is:

```
G !((test.sv31.sv.mode = stuckOpen) & (_twin.test.sv31.sv.mode = closed))
```

We can express different contexts:

One failure vs. no failure: In this case the context is given from this invariant:

```
INVAR test._brokencount < 2
INVAR _twin.test._brokencount < 1
```

In this case the property is *true*.

Two failures vs. no failure: `INVAR test._brokencount < 3`
`INVAR _twin.test._brokencount < 1`

In this case the failure is no more diagnosable. The supplied counterexample shows that:

- `test.sv31.sv.mode` is of course *stuckOpen*.
- The microswitch `test.sv31.openMs` is also broken; since this microswitch senses whether PV03 is open, it signals a *notOpen* position, even if the valve is open.

In this context failure rank increases to 7, which corresponds to a very unlikely event.

No openMs failure vs. no failure: Here we impose that there is no failure on the component `test.sv31.openMs`, by constraining the diagnosability property. The context is:

```
INVAR _twin.test._brokencount < 1
```

and the property becomes:

```
G !(test.sv31.openMs.mode = nominal & test.sv31.sv.mode = stuckOpen
& _twin.test.sv31.sv.mode = closed)
```

In this case the invariant is *true*, so there are no other cases which can yield non-diagnosable situations.

5.4.3 Failure Scenario 9: VR01 stuck open

Similarly to the previous scenario, this scenario checks if diagnosis system can detect whether VR01 is closed or stuck open. This turned out to provide the most interesting results.

The property is:

```
G !(test.vr01.mode = stuckOpen & _twin.test.vr01.valvePosition = closed)
```

The initial context is one failure vs. no failure, i.e.:

```
INVAR test._brokencount < 2
INVAR _twin.test._brokencount < 1
```

1. The property resulted to be *false*. *NuSMV* shows a counterexample where VR01 is stuck open on one side and closed in the other, and the diagnosis system cannot differentiate between the two.

VR01 is a venting valve, used to release excessive pressure out of a liquid oxygen tank. The ambiguity we found suggests that diagnosis may fail to notice that the valve is stuck open. Such a situation might vent all the stored oxygen in the environment, which is definitely an undesired outcome.

In order to understand what really happened, we presented these results to one of the experts who constructed the Livingstone model. Together, we discussed for a couple of hours, scrutined the produced traces, and searched for a valid explanation.

We discovered that this trace is related to a particular case where there is no pressurization in the oxygen tank, so that pressure rates get values below thresholds that were not properly anticipated for in the model.

The expert acknowledged that this situation is interesting because it captured a missing constraint in the model (rates cannot suddenly reach those value). She also commented that, on the other hand, this context is not relevant in the real operational context.

2. We forced the oxygen tank pressurization, and the property became diagnosable, as expected. Concretely, we impose the context:

```
DEFINE ctx_pressure_up := ((test.sv01.valvePosition = open)
               & (_twin.test.sv01.valvePosition = open)
               & (test.sv03.sv.valvePosition = open)
               & (_twin.test.sv03.sv.valvePosition = open));
```

6 Conclusions and future work

Basically our main result concerns the feasibility of the theoretical framework. We demonstrated that it is possible easily use formal verification techniques in a diagnosability context.

We also realized and tuned a set of tools that partially simplify the process. Most importantly, we identified a list of capabilities which should be moved at higher level (in the *jmpl2smv* compiler, for example), and a to-do list for future activities.

On the model-checking side, we discovered that essentially the BDD-based approach fails, due to the well-known space state explosion problem, given current application complexity and available processing power. On the other hand, the SAT-based approach proved to be sufficient and very efficient in all cases. This is probably due to the natureof the models, so this approach might not work in a more generic context.

About the verification activity, we did not find real bugs, but we obtained one interesting situation which required the interaction with the model specialists. We demonstrated that the model X-34 misses some constraints about non-realistic behaviors, and that a counterexample can suggest modelers to add

missing components. Above all, we demonstrated that the model checking techniques can be successfully applied for the verification of diagnosability.

In the meantime, we got some notes and feedback about the nature of models, readability of traces, and feasibility of the method.

This work can be extended in several directions:

Verification of the PITEX model PITEX model is more complex and larger than the X-34 model. Nevertheless, we do not expect significant differences in performance when using the SAT-based approach.

Extension of *jmpl2smv* compiler Any job carried out by the scripts we contextually realized, could be moved inside *jmpl2smv*.

Variable Dependencies Detection in *NuSMV* The BDD package in *NuSMV* could be extended in order to perform this functionality.

Induction approach extension The BMC module in *NuSMV* could be extended if needed, in order to deal with a more than two-steps induction approach.

Traces readability Traces are very hard to read at the moment, because they separate and do not correlate the variables from the two coupled models. A more advanced visualizer could highlight differences and similar values for Left and Right sides in twin models.

Acknowledgements

This work has been carried in close collaboration with the PITEX Experiment Team at NASA Ames. We are especially thankful to Dr. Anupa Bajwa, who provided all the needed information and support regarding the PITEX Livingstone model, and kindly devoted a significant part of her time in numerous very helpful conversations with us.

References

[1] C. Pecheur, A. Cimatti *Formal Verification of Diagnosability via Symbolic Model Checking*

[2] Daniel Clancy, William Larson, Charles Pecheur, Peter Engrand, Charles Goodrich. Autonomous Control of an In-Situ Propellant Production Plant.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. *Tools and Algorithms for Construction and Analysis of Systems*, In TACAS'99, March 1999.

[4] R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM Proceedings International Workshop on Logic Synthesis*, Lake Tahoe (NV), May 1995.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. *Proceedings of Computer Aided Verification (CAV 02)*, 2002.

[6] Charles Pecheur, Reid Simmons. From Livingstone to SMV: Formal Verification for Autonomous Spacecrafts. *Proceedings of First Goddard Workshop on Formal Approaches to Agent-Based Systems, NASA Goddard, April 5-7, 2000*. To appear in *Lecture Notes in Computer Science, Springer Verlag*.

[7] Bwolen Yang, SMV with Macro Expansion, *CAV 99*. Can be found at `http://www-2.cs.cmu.edu/~bwolen/software/`

[8] B. Yang, R. Simmons, R. E. Bryant, D. R. O'Hallaron. Optimizing Symbolic Model Checking for Contraint-Rich Models.

[9] R. Cavada, A. Cimatti, E. Olivetti, M. Pistore, M. Roveri, NuSMV 2.1 User Manual. Available at `http://nusmv.irst.itc.it`

[10] A. Bajwa, A. Sweet, The Livingstone Model of a Main Propulsion System, to appear in: Proceedings of IEEE Aerospace Conference, March 2003.

[11] Brian C. Williams, P. Pandurang Nayak, A Model-based Approach to Reactive Self-Configuring System, *Appears in the Proceedings of AAAI-96*