

# Model Compilation for Real-Time Planning and Diagnosis with Feedback

Anthony Barrett

Jet Propulsion Laboratory

California Institute of Technology

4800 Oak Grove Drive, M/S 126-347, Pasadena, CA 91109

anthony.barrett@jpl.nasa.gov

## Abstract

This paper describes MEXEC, an implemented micro executive that compiles a device model that can have feedback into a structure for subsequent evaluation. This system computes both the most likely current device mode from  $n$  sets of sensor measurements and the  $n-1$  step reconfiguration plan that is most likely to result in reaching a target mode – if such a plan exists. A user tunes the system by increasing  $n$  to improve system capability at the cost of real-time performance.

## 1 Introduction

Over the past decade spacecraft complexity has exploded with increasingly ambitious mission requirements. Relatively simple flyby probes have been replaced with more capable remote orbiters, and these orbiters are slowly becoming communications relay satellites for even more ambitious mobile landers like the current Mars Exploration Rover, the planned Mars Science Lab, and the suggested aerobot at Titan. With this increased complexity there is also an increased probability that components will break and in unexpected ways with subtle interactions. While traditional approaches hand-craft rule-based diagnosis and recovery systems, the difficulty in creating these rule bases quickly gets out of hand as component interactions become more subtle. Model-based approaches address this issue, but their acceptance has been retarded by the complexity of the underlying evaluation systems when compared with a simple rule evaluator whose performance is guaranteed to be linear in the number of rules [Darwiche, 2000].

This paper combines ideas from Livingston [Williams and Nayak, 1996] with results in knowledge compilation for diagnosis [Darwiche, 1998] and planning [Barrett, 2004] to create MEXEC, a micro executive that is both model-based and has an onboard evaluation system whose simplicity is comparable to that of a rule evaluator. This involves taking a device model and compiling it into a structure that facilitates determining both a system's current mode and how to reconfigure to a desired target

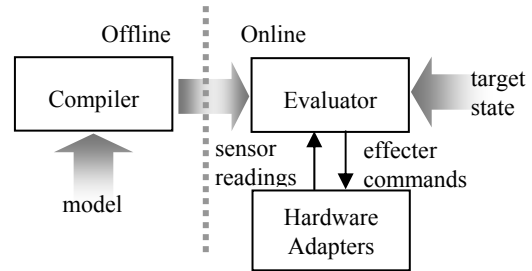


Figure 1: Online/Offline architecture for MEXEC

mode in linear time. Thus the system's architecture consists of an offline compiler and an online evaluator (see figure 1).

In addition an online performance guarantee, the global system reasoning during compilation handles feedback loops without any explicit mechanism. Also, the same compiled structure facilitates both mode identification and planning. Evaluating it one way computes the most likely current mode given  $n$  sets of measurements, and evaluating it another way computes the best  $n-1$  step reconfiguration plan given current and target modes – if such a plan exists.

This paper starts by defining our device representation language and compares it with Livingston's. It next presents a simple example device and shows how to compile it into an internal representation for linear time planning and diagnosis. The subsequent section shows how the structure is evaluated for both planning and diagnosis. To provide some realism, the implementation is described with a number of experiments. Finally the paper ends with a discussion of future work and conclusions.

## 2 Representing Devices

MEXEC's device modeling language is a simplified yet equally expressive variant of Livingstone's MPL. We model a device as a connected set of components, where each component operates in one of a number of modes. Essentially, each mode defines the relationships between a component's inputs and its outputs. More precisely, we use five constructs to define: types of connections, abstract relations, components with modes and relations between inputs and outputs, modules to define multiple component

subsystems, and the top-level system being diagnosed. The following conventions facilitate defining our syntax.

- A word in *italic* denotes a parameter, like *value*.
- Ellipsis denotes repetition, like *value*...
- Square brackets denote optional contents, like [*value*].
- A vertical bar denotes choice between options, like *false* | *true*.

With these conventions the entire language's syntax is defined in figure 2, which has constructs to respectively define connection types, well-formed formulas with arguments, user defined relations, components, modules, and a system. Just like Livingstone, system *name*'s structure is a connected set of components and modules with input/output ports, but unlike Livingstone these inputs and outputs can form feedback loops and are statically defined in :connections sensed connections (:sensors) and the commanded connections (:affectors). While the *wffs* in components can only refer to variables in local component ports, the *wffs* in :constraint entries can refer variables defined in :structure elements as well as locally defined variables.

Another divergence from Livingstone involves the modeling of components. While the syntax is similar, the semantics revolves around the concept of cost. Essentially a mode's cost denotes how unlikely it is irrespective of any information, and a transition's cost denotes how unlikely it is when its preconditions hold. While getting costs from Livingstone's probabilities is a simple matter of taking a probability's negative log, our formalism makes users directly specify costs to reflect that the number specified is manually guessed, just like a probability.

```
(defvalues ctype ( value... ))
wff → :false | :true | (:not wff) | (:and wff... ) | (:or wff... ) |
      (= cname value) | (== cname cname) | (rname arg... )
arg → wff | cname | value
(defrelation rname ( parameter... ) wff)
(defcomponent stype
  :ports      ( (ctype cname)... )
  :modes      ( (mname [:cost int] [:model wff])... )
  :transitions (( {mname | *} -> mname wff [:cost int ])... ))
(defmodule stype
  :ports      ( (ctype cname)... )
  :connections ( (ctype cname)... )
  :structure   ( (stype sname ( cname... ))... )
  [:constraint wff ])
(defsystem name
  :sensors     ( (ctype cname)... )
  [:affectors  ( (ctype cname)... )]
  [:connections ( (ctype cname)... )]
  :structure   ( (stype sname ( cname... ))... )
  [:constraint wff ])
```

Figure 2: Syntax of device modeling language

### 3 Model Compilation

To provide an example of a modeled device, consider the following system, which has a single siderostat for tracking a star within an interferometer. This system is kept as simple as possible in order to facilitate its use as a running example in the rest of the paper. It starts by defining the values and then defines a component using the values and finally defines a system in terms of the component. One semantic restriction not mentioned in the syntax is that a definition cannot be used until after it has appeared. This keeps modelers from crafting infinite recursive definitions.

```
(defvalues boolean (false true))
(defvalues command (idle track none))
(defcomponent siderostat
  :ports ((command in) (boolean valid))
  :modes ((Tracking :model (= valid true)
          :cost 20)
         (Idling :model (= valid false)
          :cost 5)
         (unknown :cost 1000))
  :transitions
  ((Tracking -> Idling (= in idle))
   (Idling -> Tracking (= in track))
   (* -> unknown :true :cost 1000)))
(defsystem tst
  :sensors ((boolean o))
  :affectors ((command c))
  :structure ((siderostat sw (c o))))
```

#### 3.1 Model to CNF

Compiling a device model starts by taking a system definition and recursively expanding its modules using the defmodules until only components remain with an optional conjunctive constraint. Since the example lacked any defmodules, this step results in a single component called "sw" which is a siderostat in the following list, where *c* is a command effector and *o* is a Boolean observation sensor

```
((siderostat sw (c o)))
```

As the example implies, name substitution occurs during the expansion. Inputs and outputs are replaced by actual parameter names – *in* and *valid* respectively become *c* and *o*. While not visible in the example, components get unique names by prefixing each structure element name with the current module name. For instance, if *tst* were a module, *sw* would become *tst\*sw*, and the connection names would be similarly prefixed. This naming convention facilitates allowing module constraints that refer to connections and modes in substructure.

After determining components, their mode definitions are converted into a Boolean expression. This involves building an equation with the following form, where *sname* is the component's name, and each disjunctive entry is for a different mode *mname* with model *wff*. Within this form, notice the subscripts that vary from 0 to *n-1*.

```
(:and (:or (:not (= sname*mode mname)
              wffi)... )
```

For example, if  $n$  were one in our example the resulting equation would be the following.

```
(:and (:or (:not (= sw*mode0 Tracking))
           (= o true))
      (:or (:not (= sw*mode0 Idling))
           (= o false)))
```

For higher  $n$ , the disjuncts are replicated for each step and transition disjuncts are added. Transition disjuncts take on the following form, where  $sname$  is the component name,  $X$  denotes the  $X^{\text{th}}$  component transition,  $frm_x/to_x$  respectively denote the transition's source/destination, and  $wff_{x,i}$  denotes its precondition at step  $i$ . When  $frm_x$  is “\*”, the  $frm_x$  constraint is dropped to denote that the transition is always enabled.

```
(:or (:not (= sname*transi X))
      (:and (= sname*modei frmx)
            (= sname*modei+1 tox) wffx,i))
```

Finally, these user-defined disjuncts are supplemented with system-defined disjuncts for not transitioning at all. They look respectively as follows, where the noop equation's size depends on the number of transitions in order to avoid choosing no transition when some transition is enabled.

```
(:or (:not (= sname*transi noop))
      (:and (== sname*modei sname*modei+1)
            (:or (:not (= sname*modei frmx)
                  (:not wffx,i)) ... )))
```

Finally, with these constructs the compiler turns the a set of components into a single Boolean equation, conjoins that equation with the conjunction of  $wffs$ , and subsequently flatten the equation into a CNF form.

### 3.2 CNF to DNNF

Unfortunately finding a minimal satisfying assignment to a CNF equation is an NP-complete problem, and more compilation is needed to achieve linear time evaluation. Fortunately results from knowledge compilation research [Darwiche and Marquis, 2002] show how to convert the CNF representation into Decomposable Negation Normal Form (DNNF). It turns out that this form of logical expression can be evaluated in linear time to compute either the most likely diagnosis or an optimal  $n$  level plan.

While DNNF has been defined previously in terms of a Boolean expression, we make a slight extension for variable logic equations, where the negation of a variable assignment can be replaced by a disjunct of all other possible assignments to that same variable. This extends the DNNF formalism to constraint satisfaction problems.

**Definition 1:** A variable logic equation is in Decomposable Negation Normal Form if (1) it contains no negations and (2) the subexpressions under each conjunct refer to disjoint sets of variables.

Just as in the Boolean case, there are multiple possible variable logic DNNF expressions equivalent to the CNF and the objective is to find one that is as small as possible.

Since Disjunctive Normal Form is also DNNF, the largest DNNF equivalent is exponentially larger than the CNF. Fortunately much smaller DNNF equivalents can often be found. The approach here mirrors the Boolean approach to finding a d-DNNF [Darwiche, 2002] by first recursively partitioning the CNF disjuncts and then traversing the partition tree to generate the DNNF.

The whole purpose for partitioning the disjuncts is to group those that refer to the same variables together and those that refer to different variables in different partitions. Since each disjunct refers to multiple variables, it is often the case that the disjuncts in two sibling partitions will refer to the same variable, but minimizing the cross partition variables dramatically reduces the size of the DNNF equation. This partitioning essentially converts a flat conjunct of disjuncts into an equation tree with internal AND nodes and disjuncts of literals at the leaves, where the number of propositions appearing in multiple branches below an AND node is minimized.

Mirroring the Boolean compiler, partitioning is done by mapping the CNF equation to a hyper-graph, where nodes and hyper-arcs respectively correspond to disjuncts and variables. The nodes that each hyper-arc connects are determined by the disjuncts where the hyper-arc's corresponding variable appears. Given this hyper-graph, a recursive partitioning using a probabilistic min-cut algorithm [Wagner and Klimmek, 1996] computes a relatively good partition tree for the disjuncts, and generalizing this algorithm by weighting the hyperarcs with associated variable cardinalities does even better. See Figure 3 for an extremely simple example with two disjuncts and three variables whose cardinalities are 2. From the equation tree perspective, there is an AND node on top above disjuncts at the leaves. The branches of the AND node share the variable  $b$ , which is recorded in the top node's *Sep* set.

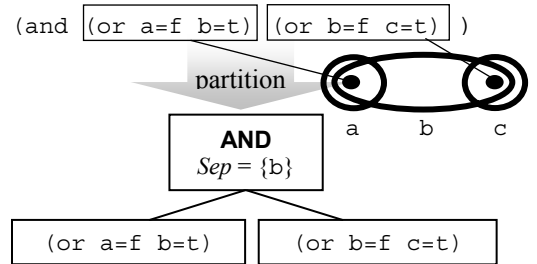


Figure 3: Example of partitioning a CNF equation

Once the equation tree is computed, computing the DNNF involves extracting each AND node's associated shared variables using the equality

$$eqn = \bigvee_{c \in \text{domain}(v)} (v = c \wedge eqn \setminus \{v = c\}),$$

where  $eqn \setminus \{v = c\}$  is an equation generated by replacing disjuncts containing  $v = c$  with *True* and removing assignments to  $v$  from other disjuncts. If a disjunct ever ends up with no assignments, it becomes *False*.





## 6 Related Work

While others have made the leap to applying compilation techniques to both simplify and accelerate embedded computation to determine a system's current mode of operation, they are more restricted than MEXEC. First, DNNF equation creation and evaluation was initially developed in a diagnosis application [Darwiche, 1998], but the resulting system restricted a component to only have one output and that there cannot be directed cycles (feedback) between components. MEXEC makes neither of these restrictions. The Mimi-ME system [Chung *et al.*, 2001] similarly avoided making these restrictions, but it lacks hard real-time guarantees by virtue of having to solve an NP-complete problem, called MIN-SAT, when converting observations into mode estimates. MEXEC supports hard real-time performance guarantees.

The closest related work on real-time reconfiguration planning comes from the Burton reconfiguration planner used on DS-1 [Williams and Nayak, 1997] and other research on planning via symbolic model checking [Cimatti and Roveri, 1999]. In the case of Burton our system improves on that work by relaxing a number of restricting assumptions. For instance, Burton required the absence of causal cycles (feedback), no two transitions within a component can be simultaneously enabled, and that each transition must have a control variable in its precondition. MEXEC has none of these restrictions. On the other hand, our system can only plan  $n$  steps ahead where Burton did not have that limitation. Similarly, the work using symbolic model checking lacked the  $n$ -step restriction, but it compiled out a universal plan for a particular target state. Our system uses the same compiled structure to determine how to reach any target state within  $n$  steps of the current state.

## 7 Conclusions

This paper presented the MEXEC system, a knowledge-compilation based approach to implementing an offline domain compiler that enables embedded hard real-time diagnosis and reconfiguration planning for more robust system commanding. This system also enables feedback reasoning by virtue of global analysis during compilation.

## Acknowledgements

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The author would also like to thank Alan Oursland, Seung Chung, Adnan Darwiche, Daniel Dvorak, and Mitch Ingham for discussions contributing to this effort

## References

[Barrett, 2004] Anthony Barrett. Domain Compilation for Embedded Real-Time Planning. In *Proceedings of the Fourteenth International Conference on Automated*

- Planning & Scheduling*, Whistler, British Columbia, Canada, June 2004. AAAI Press.
- [Chung *et al.*, 2001] Seung Chung, John Van Eepoel, and Brian Williams. Improving Model-based Mode Estimation through Offline Compilation. In *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space*, St-Hubert, Canada, June 2001.
- [Chung and Barrett, 2003] Seung Chung and Anthony Barrett. Distributed Real-time Model-based Diagnosis. In *Proceedings of the 2003 IEEE Aerospace Conference*, Big Sky, MT. March 2003.
- [Cimatti and Roveri, 1999] Alessandro Cimatti and Marco Roveri. Conformant Planning via Model Checking. In *Recent Advances in AI Planning, 5th European Conference on Planning*, Durham, UK: Springer
- [Darwiche, 1998] Adnan Darwiche. Model-based diagnosis using structured system descriptions. *Journal of Artificial Intelligence Research*, 8:165-222.
- [Darwiche, 2000] Adnan Darwiche. Model-based diagnosis under real-world constraints. *AI Magazine*, Summer 2000.
- [Darwiche, 2002] Adnan Darwiche. A Compiler for Deterministic Decomposable Negation Normal Form. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*. 627-634. Edmonton, Alberta, Canada: AAAI Press.
- [Darwiche and Marquis, 2002] Adnan Darwiche and Pierre Marquis. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research* 17:229-264.
- [Ingham *et al.*, 2001] Michel Ingham, Brian Williams, Thomas Lockhart, Amalaye Oyake, Micah Clarke, and Abdullah Aljabri. Autonomous Sequencing and Model-based Fault Protection for Space Interferometry, In *Proceedings of International Symposium on Artificial Intelligence, Robotics and Automation in Space*, St-Hubert, Canada, June 2001.
- [Kushmerick *et al.*, 1994] Nicholas Kushmerick, Steve Hanks, and Daniel Weld. An Algorithm for Probabilistic Least-Commitment Planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA: AAAI Press. 1994.
- [Wagner and Klimmek, 1996] Frank Wagner and Regina Klimmek. A Simple Hypergraph Min Cut Algorithm. Technical Report, b 96-02, Inst. Of Computer Science, Freie Universität Berlin. 1996.
- [Williams and Nayak, 1996] Brian Williams and P. Pandurang Nayak. A Model-based Approach to Reactive Self-Configuring Systems. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR: AAAI Press. 1996.
- [Williams and Nayak, 1997] Brian Williams and P. Pandurang Nayak. A Reactive Planner for a Model-based Executive. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, Nagoya, Japan: Morgan Kaufmann. 1997.