Model Compilation for Embedded Real-Time Planning and Diagnosis

Anthony Barrett
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, M/S 126-347
Pasadena, CA 91109-8099
anthony.barrett@jpl.nasa.gov

Abstract. This paper describes MEXEC, an implemented model-compilation based executive that translates a device model into an internal structure. Not only does this structure facilitate computing the most likely current device mode from *n* sets of sensor measurements, but it also facilitates generating an *n* step reconfiguration plan that is most likely to result in reaching a target mode – if such a plan exists.

1 Introduction

Over the past decade the complexity of spacecraft has rapidly grown with increasingly ambitious mission requirements. Relatively simple flyby probes were 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 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 their 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 model-compilation 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 model specified in a language like Livingston's and compiling it into an equation that can be evaluated in linear time to determine both a system's current mode from sensor readings and how to reconfigure to a desired target mode. Thus the system's architecture consists of an offline device-model compiler and an online evaluator (see figure 1).

In addition an online performance guarantee, MEXEC whittles away at the restrictions required by Livingston's real-time planner. The surprising result is that the same compiled structure can be used to for both diagnosis/mode identification and planning. Evaluating it one way facilitates computing the most likely current mode given *n* sets of measurements, and evaluating it another way facilitates computing an *n*-1 step reconfiguration plan with the highest probability of success given the current and target modes if such a plan exists. While selecting a large *n* at compile time results in a more capable onboard executive, it also results in a larger Boolean equation that takes longer to evaluate.

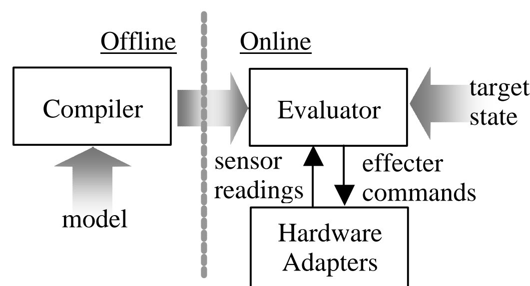

Figure 1: Online/Offline architecture for MEXEC

This paper starts by defining the device representation language and compares it with Livingston's language. It next presents a simple device example and shows how to compile it into an internal representation that can be evaluate in linear time to either plan or diagnose. 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 modeling language is called the Connection Model Programming Language (CMPL). CMPL is a simplified yet equally expressive variant of Livingstone's MPL. CMPL models 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, CMPL has 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 CMPL's 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 inputs and outputs, but unlike Livingston these inputs and outputs are statically defined in :connections. While the use of subsystems (Chung and Barrett 2003) is also a divergence from Livingston, they are outside of the scope of this paper with the exception of pointing out that the two argument lists of each subsystem respectively denote the sensed connections (sensors) and the commanded connections (effectors).

The third 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 given no prior information, and a transition's cost denotes how unlikely it is when its preconditions hold. While getting costs from Livingston's probabilities is a simple matter of taking a probability's negative log, CMPL makes users directly specify costs to reflect that the number specified is manually guessed, just like a probability.

As in other approaches to model based diagnosis (Williams and Nayak 1996; Darwiche 1998), this representation language facilitates modeling a system as a graph of interacting components, where each edge denotes a modeled signal between components. Components are simple state machines with modes denoting modeled states. Each transition associated with a mode defines a possible next mode when the transition's *wff* precondition holds among the component port signals. Finally, each mode has an associated model *wff* denoting an imposed constraint among the port signals.

(**defvalues** *ctype* (*value*…))

wff → (**:not** *wff*) | (**:and** *wff*…) | (**:or** *wff*…) |
(**=** *cname value*) | (**==** *cname cname*) | (*rname arg*…) |
(**:false**) | (**:true**)

arg → *wff* | *cname* | *value*

(**defrelation** *rname* (*parameter*…) *wff*)

(**defcomponent** *stype*
:ports ((*ctype cname*)…)
:modes ((*mname* [**:cost** *int*] [**:model** *wff*]
[**:transitions** ((*mname wff* [**:cost** *int*])…)])…))

(**defmodule** *stype*
:ports ((*ctype cname*)…)
:connections ((*ctype cname*)…)
:structure ((*stype sname* (*cname*…))…))

(**defsystem** *name*
[**:subsystems** ((*name* ([*cname*…]) ([*cname*…]))…)]
:connections ((*ctype cname*)…)
:structure ((*stype sname* (*cname*…))…))

Figure 2: Syntax of Connection Model Programming Language (CMPL)

Given a system model, the diagnosis problem turns into one of finding the cheapest initial modes and transitions to explain the last n sets of sensed signals in the context of the last n-1 sets of command signals. The current mode estimate derives from this computation. Similarly, the reconfiguration-planning problem turns into an effort to find n-1 sets of command signals that result in enabling the cheapest transitions from the current system state to the target state. Using the following equation to compute costs from probabilities, the cheapest diagnosis is the most probable one, and enabling the cheapest transitions results in a plan with the highest probability of success.

$$\text{cost} = -\,log(\text{probability})$$

In addition to the explicitly represented costs, modes, and transitions, each component has an implicit "unknown" mode and implicit transitions to the unknown mode. This mode imposes no constraints among signals and transitions to it have no preconditions, but the mode and transitions have high costs to prefer using specified modes.

3 Model Compilation

To provide an example of CMPL in use, 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 variable types 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 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
              :transitions
              ((Idling (= in idle))))
            (Idling :model (= valid false)
              :cost 5
              :transitions
              ((Tracking (= in track))))))

(defsystem tst
   :connections ((boolean o) (command c))
   :subsystem ((main (c) (o)))
   :structure ((siderostat sw (c o)))))
```

In the example system, the siderostat can be in one of three modes: tracking, idling, and unknown. When the mode is tracking or idling `valid` is constrained to equal true or false respectively. The example transitions have an implicit zero cost and change the mode depending on the command to either track or idle.

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. 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 effecter and `o` is a Boolean observation sensor

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

After determining components, their mode definitions are converted into a Boolean expression. This involves building an equation with the following form for each mode *mname* within component *sname* that has model *wff*. Within this form, the subscripts vary from 0 to n-1 depending on the user supplied parameter n, and subscripting *wff* is equivalent to subscripting all contained variables. This results in nM disjuncts where M is the sum of the number of modes in each component.

```
(:or (:not (= mode*sname_i mname)) wff_i)    (1)
```

For n greater than one, extra formulae are needed to characterize transitions between steps. These transitions take on the following form, where *sname* is the component name, X denotes the Xth transition in the component, frm_X and to_X respectively denote the transition's source and destination, and $wff_{X,i}$ denotes its precondition at step i. Intuitively, this formula means that the transition's occurrence at instant i implies both that the component was in the transition's source mode with the precondition holding at that instant and that the component moved to the destination mode in the following instant.

```
(:or (:not (= trans*sname_i X))              (2)
     (:and (= mode*sname_i frm_X) wff_X,i
           (= mode*sname_i+1 to_X)))
```

Finally, these user-defined formulae are supplemented with system-defined formulae for not transitioning at all and transitioning to an unknown mode. 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 (= trans*sname_i noop))           (3)
     (:and (== mode*sname_i mode*sname_i+1)
           (:or (:not (= mode*sname_i frm_X))
                (:not wff_X,i))… )))

(:or (:not (= trans*sname_i to-unknown))     (4)
     (= mode*sname_i unknown))
```

With these constructs the compiler turns a set of components into a single conjoined Boolean equation to flatten into a CNF form. This equation captures both mode dependent component interactions at n sensed instants and the mode transitions for the n-1 gaps between the instants. With this equation, the diagnosis and planning problems now involve finding the cheapest set of variable assignments that satisfies the CNF equation. For instance, for n=1 our compiler simply uses equation (1) to compile `tst` into the following CNF, where a variable's not equaling one value implies its equaling some other value.

```
(and (or (= mode*sw_0 Tracking)
         (= mode*sw_0 unknown) (= o_0 false))
     (or (= mode*sw_0 Idling)
         (= mode*sw_0 unknown) (= o_0 false)))
```

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.

DNNF has been defined previously in terms of a Boolean expression where only literals are negated and the literals appearing in sub-expressions of a conjunct are disjoint. The following definition slightly extends Boolean DNNF to variable logic equations, where the negation of a variable assignment has been replaced by a disjunct of all other possible assignments to that same variable.

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, one 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 referring to the same variables together and those referring 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 a hyper-arc connects denote those 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 hyper-arcs 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.

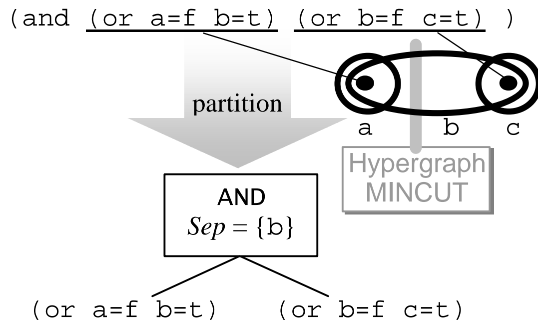

Figure 3: Example of partitioning a CNF equation by using a hypergraph-partitioning algorithm base on MINCUT.

Once the equation tree is computed, deriving the DNNF involves extracting each AND node's *Sep* set and for each shared variable v using the equality

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

where *eqn*\{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*.

More formally, the DNNF equation is recursively defined using the following two equations, where the first and second equations apply to internal and leaf nodes respectively. In the first equation *instances*(*N.Sep*,α) refers to the set of possible assignments to the vector of variables in *N.Sep* that are consistent with α. For instance, running these equations over Figure 3's partition starts by calling *dnnf*(*root*,*True*), and the instances are b=t and b=f since only b is in *root.Sep*, and both assignments agree with *True*. In general the number of consistent instances grows exponentially with *N.Sep*, leading to the use of min-cut to reduce the size of *N.Sep* for each partition.

$$dnnf(N,\boldsymbol{a}) \equiv \bigvee_{\boldsymbol{b} \in instances(N.Sep,\boldsymbol{a})} \left(\boldsymbol{b} \wedge \bigwedge_{c \in N.kids} dnnf(c, \boldsymbol{a} \wedge \boldsymbol{b}) \right)$$

$$dnnf(disj,\boldsymbol{a}) \equiv \begin{cases} True & \text{if } \boldsymbol{a} \Rightarrow disj \\ \bigvee_{\boldsymbol{b} \in disj \,\&\, \boldsymbol{a} \not\Rightarrow \neg \boldsymbol{b}} \boldsymbol{b} & \text{if } \exists \boldsymbol{b} \supset \boldsymbol{a} \Rightarrow \neg \boldsymbol{b} \\ False & \text{Otherwise} \end{cases}$$

While walking the partition does provide a DNNF equation that can be evaluated in linear time, two very important optimizations involve merging common sub-expressions to decrease the size of the computed structure and caching computations made when visiting a node for improving compiler performance (Darwiche 2002). With respect to Figure 3, there were no common sub-expressions to merge, and the resulting DNNF expression appears below.

```
(or (and b=t c=t) (and b=f a=f))
```

4 Onboard Evaluation

To illustrate a less trivial DNNF expression, consider Figure 4 for the siderostat DNNF. Actually this is a slight simplification of the generated DNNF – a third top level branch for unknown state reasoning was omitted for space reasons. This expression's top rightmost AND node has three children, and each child refers to a unique set of variables. From top to bottom these disjoint sets respectively are

{mode*sw_1}, {o_1}, and {mode*sw_0, trans*sw_0, o_0, c_0}.

Given that DNNF AND nodes have branches that refer to disjoint sets of variables, choosing variable assignments to satisfy one branch has no effect on the other branches. This property dramatically simplifies the search for optimal satisfying variable assignments into a three-step process:

1. associate costs with variable assignments in leaves;
2. propagate node costs up through the tree by either assigning the min or sum of the descendents' costs to an OR or AND node respectively; and
3. if the root's cost is 0, infinity, or some other value then respectively return default assignments, failure, or descend from the root to determine and return the variable assignments that contribute to its cost.

4.1 Mode Estimation

Evaluating a DNNF structure to determine component modes for the tst example starts by assigning costs to the mode*$name_0$ variables, where these costs come from the **:cost** entry associated with each mode in the original model, and missing cost entries are assumed to be zero. For instance, none of the transitions have associated costs in the model, resulting in assigning zero to the trans*$name_0$ leave costs. Finally, sensed values are assigned either zero or infinity depending on the value

sensed. In this case the sensed values for o_0 and o_1 were both true.

Following the simple propagation step, the associated node costs appear above the nodes in Figure 4. Note that the cost is of the top level node is 20. This value is used to prune the search when descending down the tree to determine the assignment to mode*sw_1, which is the most likely siderostat mode that matches the observations.

While this approach assumes forgetting of old state information, it can be enhanced to either remember the most likely last state or a set of likely last states using a particle filter approach. Since the only difference between such approaches revolves around leaf cost assignments, the requisite changes are very manageable.

16 node unknown modes branch 10000
0: [mode*sw_1 = Tracking]
0: [o_1= true]
20: [mode*sw_0 = Tracking]
0: [trans*sw_0 = noop]
0: [o_0 = true]
inf: [c_0 = track]
0: [c_0 = none]
5: [mode*sw_0 = Idling]
inf: [c_0 = track]
inf: [o_0 = false]
0: [trans*sw_0 = 2]
0: [mode*sw_1 = Idling]
inf: [o_1 = false]
5: [mode*sw_0 = Idling]
0: [trans*sw_0 = noop]
inf: [o_0 = false]
inf: [c_0 = idle]
0: [c_0 = none]
20: [mode*sw_0 = Tracking]
inf: [c_0 = idle]
0: [o_0 = true]
0: [trans*sw_0 = 1]

Figure 4: Evaluating an n=3 DNNF structure to determine the siderostat mode from two sets of observations.

4.2 Reconfiguration Planning

When evaluating a DNNF structure for a reconfiguration plan, a cost is assigned to each variable using a number of planning dependent preferences. First, not performing an action has zero cost. This results in associating zero with all leaves that set transitions to noops. Second, leaves denoting other transitions are assigned costs that come from :cost entries associated with transitions. In the example all of these costs are assumed to be zero. Finally mode*$name_0$ and mode*$name_n$ entries have costs that depend respectively on the current and target mode. Those leaf assignments that are consistent with these modes will cost zero and inconsistent leaves get an infinite cost. For instance, Figure 5 documents the evaluation to take a currently tracking siderostat and make it idle. In this case the cost is propagated up and then it is used to guide the descent to find the desired cost of c_0, the effecter variable.

16 node unknown modes branch inf
inf: [mode*sw_1 = Tracking]
0: [o_1= true]
0: [mode*sw_0 = Tracking]
0: [trans*sw_0 = noop]
0: [o_0 = true]
0: [c_0 = track]
0: [c_0 = none]
inf: [mode*sw_0 = Idling]
0: [c_0 = track]
0: [o_0 = false]
0: [trans*sw_0 = 2]
0: [mode*sw_1 = Idling]
0: [o_1 = false]
inf: [mode*sw_0 = Idling]
0: [trans*sw_0 = noop]
0: [o_0 = false]
0: [c_0 = idle]
0: [c_0 = none]
0: [mode*sw_0 = Tracking]
0: [c_0 = idle]
0: [o_0 = true]
0: [trans*sw_0 = 1]

Figure 5: Evaluating an n=3 DNNF structure to compute a reconfiguration plan.

While a need to keep this example simple motivates not tagging transitions with costs, such tags reflect the likelihood of a transition once its preconditions are met. Thus, multiple transitions can have consistent preconditions and the underlying evaluation will actually adjust the preconditions to maximize the likelihood that the triggered transitions will result in attaining the target conditions. This implies that the planning algorithm finds n-1 step solutions to probabilistic planning problems like those of BURIDAN (Kushmerick, Hanks, and Weld 1994). From this vantage point MEXEC's compiled internal structure can be viewed as a limited policy for solving POMDP problems if a solution can be found in n-1 steps, but this perspective has yet to be fully explored.

5 Implementation and Experiments

The system is currently implemented in Allegro Common LISP with under 500 lines to compute a device's associated CNF, under 500 lines to compute a CNF equation's associated DNNF, and less than 80 lines to evaluate a DNNF equation to find all minimal cost satisfactions.

In addition to testing MEXEC on various switching circuit examples, there has been some work on developing and experimenting with models of a Space Interferometer Mission Test Bed 3 (STB-3) model (Ingham et al. 2001) as well as the Formation Interferometer Test Bed (FIT) model, which is an extension on the STB-3 model. While

STB-3 represents a single spacecraft interferometer, FIT represents a separated spacecraft interferometer. As illustrated in Figure 6, FIT is composed of combiner (right) and collector (left) spacecraft. The collector spacecraft precisely points at a star and reflects the starlight beam to the combiner spacecraft. While the combiner spacecraft also points at the star to collect the starlight, it accurately points at the collector spacecraft in order to combine the starlight from the collector spacecraft with its own.

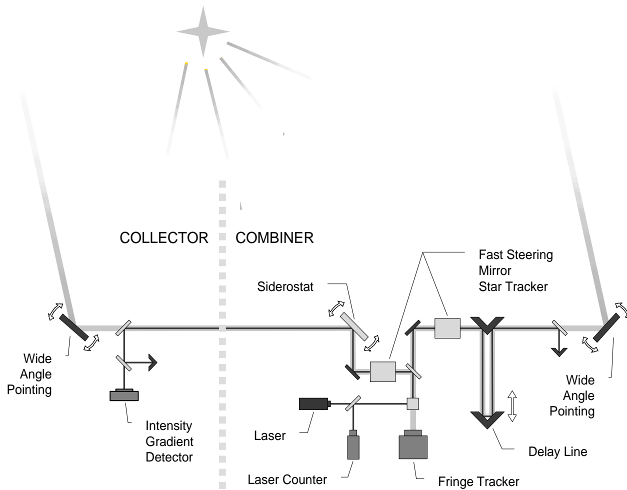

Figure 6: A simplified schematic of the Formation Interferometer Testbed (FIT). The left side of the dotted line represents the collector spacecraft and the right side of the dotted line represents the combiner spacecraft.

Device	C	V	S	$n = 1$	$n = 2$	$n = 3$	$n = 4$
`tst`	1	3	2	8	35	60	92
STB-3	7	26	13	125	851	5191	36354
FIT	17	64	12	235	1488	7620	39705

Table 1: DNNF sizes (in nodes) of interferometers with C components, V variables, S sensors, and *n* instants.

Compiling these two models and the `tst` model for instantaneous (n=1) through three transition (n=4) DNNF structures results in the generation of Table 1. The initial message to pull out of this exercise is that instantaneous DNNF structures, for diagnosis only, tend to be extremely compact, but as n increases the DNNF size grows exponentially. Still, the DNNF size for the trivial `tst` model only rose linearly and work on planning (Barrett 2004) and strict DNNF compilation (Darwiche 2002) leads one to suspect that this scaling issue can be improved.

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 disallowed directed feedback cycles between components. MEXEC makes neither of these restrictions. The Mimi-ME system (Chung, Van Eepoel, and Williams 2001) similarly avoided making these restrictions, but it can neither support distributed reasoning nor provide real-time guarantees by virtue of having to collect all information in one place and then solve an NP-complete MIN-SAT problem to convert observations into mode estimates. MEXEC supports performance guarantees that are linear in the size of the computed DNNF structure.

The closest related work on real-time reconfiguration planning comes from the Burton reconfiguration planner used on DS-1 (Williams and Nayak 1996) 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 (1) the absence of causal cycles, (2) no two transitions within a component can be simultaneously enabled, and (3) that each transition must have a control variable in its precondition. MEXEC has none of these restrictions. On the other hand, this system can only plan a set number of steps ahead where Burton did not have that limitation. Similarly, the work using symbolic model checking lacked the set step number 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 a set number of steps from the current state.

7 Conclusions

This paper presented the MEXEC system, a knowledge-compilation based approach to implementing an offline model compiler that enables embedded real-time diagnosis and reconfiguration planning for more robust spacecraft commanding. The MEXEC offline compiler translates a model with a user supplied parameter n into a DNNF equation that the MEXEC onboard evaluator uses to compute the modeled system's cheapest state that agrees with the last n sets of sensor readings and n-1 commands. If this estimated state does not match the desired state, the MEXEC onboard evaluator reuses to DNNF equation to compute the cheapest set of commands to reach the desired state if the state can be reached within n-1 steps.

As previously mentioned, one avenue for future work involves reducing the size of the compiled DNNF. Research on SATPLAN (Ernst, Millstein, and Weld 1997) showed that there are a number of different CNF encodings of a planning problem, and the same holds for CNF encodings of mode transitions. Experiments with alternative CNF encodings can lead to reducing the compiled DNNF encoding. A second mentioned avenue for future research involves maintaining old state information. Currently the system only remembers the last n sets of sensor values and n-1 sets of command signals. This can be generalized using a particle filter approach toward remembering a constant set of past states. Finally,

future representational enhancements to the modeling language will be driven by attempts to diagnose and command devices like interferometers, rovers, and robotic blimps with this system.

8 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, A. 2004. Domain Compilation for Embedded Real-Time Planning. In *Proceedings of the Fourteenth International Conference on Automated Planning & Scheduling*, Whistler, British Columbia, Canada: AAAI Press.

Chung, S. and Van Eepoel, J. and Williams, B. 2001. “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.

Cimatti, A. and Roveri, M. 1999, “Conformant Planning via Model Checking.” In *Recent Advances in AI Planning, 5th European Conference on Planning*, Durham, UK: Springer

Darwiche, A. 1998. Model-based diagnosis using structured system descriptions. *Journal of Artificial Intelligence Research*, 8:165-222.

Darwiche, A. 2000. Model-based diagnosis under real-world constraints. AI Magazine, Summer 2000.

Darwiche, A. 2002. 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, A. and Marquis, P. 2002. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research* 17:229-264.

Ernst, M. and Millstein, T. and Weld, D. 1997. Automatic SAT-Compilation of Planning Problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1169-1177. NAGOYA, Aichi, Japan: Morgan Kaufmann.

Ingham, M. and B. Williams, B. and Lockhart, T. and A. Oyake, A. and Clarke, M. and Aljabri, A. 2001. 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, N. and Hanks, S. and Weld, D. 1994. An Algorithm for Probabilistic Least-Commitment Planning, In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA: AAAI Press.

Wagner, F. and Klimmek, R. 1996. A Simple Hypergraph Min Cut Algorithm. Technical Report, b 96-02, Inst. Of Computer Science, Freie Universität Berlin.

Williams, B. and Nayak, P. 1996. A Model-based Approach to Reactive Self-Configuring Systems. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR: AAAI Press.

Williams, B. and Nayak, P. 1997. A Reactive Planner for a Model-based Executive. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, Nagoya, Japan: Morgan Kaufmann.