Distributed Real-time Model-based Diagnosis

Seung H. Chung
Artificial Intelligence & Space Systems Laboratories
Massachusetts Institute of Technology
70 Massachusetts Ave. 37-346
Cambridge, CA 02139
617-253-8364
chung@mit.edu

Anthony Barrett
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, M/S 126-347
Pasadena, CA 91109
818-393-5372
anthony.barrett@jpl.nasa.gov

Abstract — While past flight projects involved flying single spacecraft in isolation, over forty proposed future missions involve multiple coordinated spacecraft. This paper presents an approach to onboard anomaly diagnosis that combines the simplicity and real-time guarantee of a rule-based diagnosis system with the specification ease and coverage guarantees of a model-based diagnosis system. This system also provides a clear path to distributing the diagnosis process across a number of processors on one or more spacecraft.

TABLE OF CONTENTS

1. INTRODUCTION

The past decade has seen missions with growing numbers of probes. Pathfinder has its rover (Sojourner), Cassini has its Huygens lander, and Cluster II has 4 spacecraft for multi-point magnetosphere plasma measurements. This trend is expected to continue to ever-larger fleets with increasingly difficult coordination requirements. For example, a proposal for a Terrestrial Planet Finder (TPF) mission involves an interferometer with 5 spacecraft flying in a precise formation in order to detect earth-sized planets orbiting other stars. Another proposal for a laser interferometer space antenna (LISA) mission involves 3 spacecraft flying in a precise formation to measure low frequency gravitational radiation.

As spacecraft progressively become more complex to satisfy increasingly more ambitious mission requirements, hand coding robust rule-based diagnosis software becomes progressively more arduous and prone to error. To battle this problem, the autonomy community has focused on developing model-based Mode Identification and Reconfiguration executives that control systems to satisfy commanded states. In fact, such technology is already available for spacecraft. Recently, a model-based diagnostic engine called Livingstone [1] was successfully flown and tested on NASA's Deep Space 1 mission.

Model-based systems do not require manually enumerating rules and thus no complex verification of rule interactions is necessary. Instead, engineers specify the behaviors of simple components and how components are connected to each other to form the spacecraft. Unfortunately, current systems like Livingstone cannot provide hard real-time performance guarantees. Also, such systems tend to compile their models into rules in a truth maintenance system, which does not easily lend itself to distribution. There has been work on distributed truth maintenance, but the resulting systems are fairly complicated resulting in a reluctance to fly them on both single and multiple spacecraft missions.

In this paper, we introduce Distributed Real-time Model-based Diagnosis (DRMD). DRMD combines the benefits of the model-based systems with the hard real-time guarantee of rule-based systems. DRMD, as the name suggests, offers a distributed diagnosis capability suitable for such space missions as the TPF "Free Flyer" in which the system must be diagnosed across physically separated spacecraft.

While our approach towards modeling a system was inspired by Livingstone [1], our approach towards diagnosis is based on knowledge compilation [2,3]. Livingstone's modeling language, called the Model-based Programming Language (MPL), offers the expressiveness necessary to specify a model of a system for both diagnosis and reconfiguration. Models written in MPL are both modular and reusable. Most importantly, MPL provides the means for engineers to specify a model of a system in terms of the behaviors of its components without having to reason about how all of those behaviors interact. [2] describes a model compilation technique that takes advantage of offline computational resources, shifting as much of the computation required for diagnosis offline. This technique minimizes the amount of online computation necessary for diagnosis, and further more, it provides a hard real-time guarantee for diagnosis. To combine the aforementioned benefits, we merge the use of the expressive MPL modeling language with the model compilation and diagnosis techniques of [2,3]. Finally, we

[1] 0-7803-7651-X/03/$17.00 © 2003 IEEE
[2] IEEEAC paper #1116

extend the diagnosis approach described in [2] to provide a distributed diagnosis capability. The result is Distributed Real-time Model-based Diagnosis.

The remainder of this paper presents our DRMD system. The next two sections, sections 2 and 3, respectively define our modeling language and describe the knowledge-compilation approach to diagnosis. Section 4 subsequently explains how to distribute the online diagnosis process across multiple spacecraft, and section 5 discusses some empirical results based on experiments with a model of the formation interferometer testbed. Finally, we conclude with some remarks on future research.

2. MODELING

DRMD'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. To define the syntax of our language we use the following conventions, where syntax definitions appear in Arial font.

- 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.

Connection types

The most fundamental object defined in a model is a connection type. As in most programming languages, all types must be defined before they can be used. Just like Livingstone, types are enumerated, and CMPL defines types in terms of finite sets of values. The syntax for type declarations is as follows, where *ctype* is the type's name and *value…* is the finite set of values.

(**defvalues** *ctype* (*value*…))

For example, the following defines the domain named "boolean" that ranges over "false" and "true":

```
(defvalues boolean (false true)).
```

Relations

Once connections are typed, defining relationships among connections becomes a matter of building Boolean equations over connection assignments. For instance, the following equation requires that two Boolean connections, `A` and `B`, have the same value.

```
(:or (:and (= A true) (= B true))
     (:and (= A false) (= B false)))
```

In general, relationships are defined by well formed formulas using the following syntactic rules, where *wff* is a well formed formula, *cname* is a connection name, and *value* is a connection value.

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

arg → *wff* | *cname* | *value*

While the Boolean functions have their standard semantics, the two equalities respectively require that a connection has a particular value and that two connections with the same type have the same value. Finally, *rname* denotes a user-defined relationship. Such a definition has the following form, where the parameters appear in *wff* and are replaced by the arguments in well-formed formulas where *rname* is used.

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

For instance, the following defines a widely used implication relationship.

```
(defrelation :implies (wffa wffb)
             (:or (:not wffa) wffb))
```

We finish our definition of relations with two semantic restrictions. First, any particular *wff* can only reference previously defined relations to keep users from defining recursive relations that cannot be evaluated. Second, the arguments associated with parameters in a defrelation are constrained by where the parameters appear in the wff. Thus, a user cannot mistakenly expand a relation to get structures like (**:not** *cname*) or (**=** *wff value*).

Components

Components model primitive device types and are specified as finite state machines where each state relates component inputs to outputs in a different way. The syntax for a component type declaration is as follows, where *stype* is the name of the component type, *ctype* defines the type of connection *cname*, and *mname* denotes a component state.

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

Each component state, or "mode" using Livinstone's vocabulary, has three attributes: an integer cost that intuitively corresponds to the likelihood of the component being in that mode; a *wff* model relating inputs and outputs when the component is in the mode; and a set of transitions denoting how a component evolves to the next mode. Each transition has a target mode, a *wff* precondition for evolving to the target, and an integer cost of the transition. In general, costs vary from 0 to 10000 and reflect mode or transition's likelihoods, where a high cost denotes a low likelihood. Since most modes will be nominal, they will have an inherently zeroed cost – leading to a default of zero if the cost attribute is omitted.

For instance, consider the following simple example of an and-gate. Given the nominal "ok" mode's model, this gate definition makes the output reflect a conjunct of the inputs. While the gate only has one explicitly modeled mode, there is an implicit *unknown* mode that all gates have. The *unknown* mode costs 10000 and has a model that leaves the inputs and outputs completely unrelated. In general, the goal of diagnosis is to find the component modes with the least cost and models that agree with observations.[3]

```
(defcomponent and-gate
  :inputs  ((boolean i1)(boolean i2))
  :outputs ((boolean o))
  :modes   ((ok :model
              (:and (:implies
                      (:and (= i1 true)
                            (= i2 true))
                      (= o true))
                    (:implies
                      (:not (:and (= i1 true)
                                  (= i2 true)))
                      (= o false))))))
```

Modules

A module is a specification of a network of components and/or other sub-modules. Its main purpose is to improve the readability and the simplicity of a model. The syntax for a module specification is as follows, where *stype* is the module type's name, *ctype* and *cname* denote typed connections, and *sname* denotes a named subsystem of the module.

(**defmodule** *stype*
  [**:inputs** ((*ctype cname*)…)]
  :outputs ((*ctype cname*)…)
  :connections ((*ctype cname*)…)
  :structure ((*stype sname* ([*cname*…]) (*cname*…))…))

While inputs and outputs denote interfaces to a module, connections reflect internal signals between subsystems. A structure entry defines the subsystem network as a set of subsystems with each subsystem's inputs and outputs.

For example, consider the following definition of a three input and-gate, where the structure has typed components as defined in the previous example.

```
(defmodule and3i
  :inputs ((boolean i1)(boolean i2)(boolean i3))
  :outputs ((boolean out))
  :connections ((boolean wire))
  :structure ((and-gate a1 (i1 i2) (wire))
              (and-gate a2 (i3 wire) (out))))
```

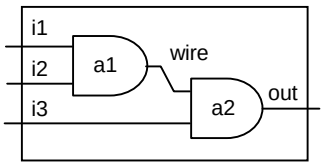

Figure 1 – Graphical representation of `and3i` module

We finish our definition of modules with two semantic restrictions within a structure entry. First, a module definition can only refer to previously defined module types – making recursive definitions illegal. Second, the output of a subsystem cannot directly or indirectly feed back into one of its inputs – to keep a designer from representing feedback loops that can cause continual mode switching.

Systems

A system is a specification of a network of components and modules. Unlike Livingstone, CMPL is extended to enable the modeling of distributed systems. In CMPL, each distributed element in the system is defined as a subsystem. Each subsystem is associated with a list of commands and observables that are local to that subsystem. The idea is that the location of the physical device is not important. For distributed diagnosis, and also for distributed commanding, the physical location of the devices' sensors and the physical location at which the commands are generated are what is important. The syntax for a system specification is as follows, where the connections and structure fields mirror the same fields in a module definition.

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

For instance, the following example defines a simple test system for our previously defined `and3i` gate. For simplicity, this example only has one subsystem and one gate. All of the top-level Boolean connections are sensed, and there are no commands feeding into the subsystem.

```
(defsystem tst1
  :subsystems
    ((sub nil (a b c o)))
  :connections ((boolean a) (boolean b)
                (boolean c) (boolean o))
  :structure ((and3i gate (a b c)(o))))
```

3. MODEL COMPILATION

While CMPL was strongly influenced by the Livingstone system, our approach to diagnosis is based on knowledge compilation [2,3]. As such, model compilation based diagnosis is a three-step process: (1) expand the system into a network of processed components at compile time; (2) compute the Boolean equation at compile time; and (3) iteratively evaluate the equation at run time.

Expanding the network

Expanding the network is a simple matter of taking a defsystem and using defmodules to expand named module elements within the structure list until only components remain. For instance, the `tst1` system mentioned above has a single module element that expands into the following two components.

[3] While our examples deal with static mode costs, these costs dynamically change when folding in transition costs from the currently perceived state. See discussion of future work in the conclusion for more on this.

```
((and-gate gate*a1 (a b) (gate*wire))
 (and-gate gate*a2 (c gate*wire) (o)))
```

As this example implies, name substitution occurs during the expansion. Inputs and outputs are replaced by actual parameter names – `i1`, `i2`, `i3`, and `out` respectively became `a`, `b`, `c`, and `o`. Subsystem names within a module are prefixed by the module's name to assure unique component names – `a1` and `a2` respectively became `gate*a1` and `gate*a2`. Similarly, connection names are also prefixed resulting in the `gate*wire`.

Once the components are determined, their mode definitions are converted into a Boolean expression in conjunctive normal form (CNF). Converting mode definitions into a Boolean expression involves building an equation with the following form, where *sname* is the components name, and each disjunctive entry is for a different mode *mname* defined in terms of model *wff*.

```
(:and (:or (:not (= mode*sname mname)) wff)… )
```

For example, the equation for the `gate*a1` is the following.

```
(:and (:or (:not (= mode*gate*a1 ok))
           (:and (:implies
                    (:and (= a true) (= b true))
                    (= gate*wire true))
                 (:implies
                   (:not (:and (= a true)
                               (= b true)))
                   (= gate*wire false))))
      (:or (:not (= mode*gate*a1 unknown))
           true))
```

Each component's Boolean equation is subsequently expanded with the defrelations, converted into CNF, and then each `(:not (= c v))` in a disjunct gets replaced with the set of entries {`(= c` v_i`)`…}, where v_i varies through the legal values of `c` not equal to `v`. Thus the above equation expands into.

```
(:and
  (:or (= mode*gate*a1 unknown)
       (= a false)(= b false)(= gate*wire true))
  (:or (= mode*gate*a1 unknown)
       (= a true)(= gate*wire false))
  (:or (= mode*gate*a1 unknown)
       (= b true)(= gate*wire false)))
```

Network expansion ends with all components instantiated with their CNF equations. While this is the internal form used for diagnosis in Livingstone [4], we have an extra step to improve runtime performance.

Building the equation

Building the equation out of a network of CNF components is a complex process that has been previously defined in detail [3]. Unfortunately, our networks violate a restriction made by previous compilation work that limits components to have only one output. We circumvent this limitation by replacing multiple-output components with multiple single-output components. There are a number of ways to do this, and our current approach can be illustrated in Figure 2.

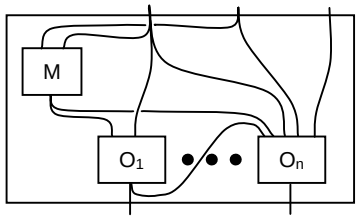

Figure 2 – Graphical representation of a multiple output gates replacement structure

In this approach each O_i corresponds with an output, and M corresponds to the original component's mode. Each of these new components has its own CNF formula, which is computed by partitioning the original component's disjuncts and then adding a few overhead disjuncts. The partition simply assigns disjuncts that mention no outputs to M and disjuncts to O_i if they refer to O_i's output but no output for O_j with `j > i`. Given the disjuncts assigned to the gates, the inputs to each gate are determined by which variables that gate's disjuncts contain.

Unfortunately, two gates cannot refer to the same unobservable mode, which is the primary reason for the M gate. This gate also contains an overhead disjunct of the following form for each of the original component's modes $mname_i$, where $mname_j$ entries are for modes other than $mname_i$.

```
(:or (= M-output*sname mname_i)
     (= mode*sname mname_j)…)
```

This collection of disjuncts force `M-output*`*sname*, which is M's output, to be equal to `mode*`*sname*, and the O_i gates have the mode variable replaced with M's output to avoid sharing an unobservable mode assignment that is not reflected in the network arcs.

Run time evaluation

After taking care of multiple output gates Darwiche's algorithm can process the component network to generate a diagnosis equation for onboard evaluation. For instance, the equation generated for our simple `tst1` example is the following.

```
(:or (:and (:or (:and (= a true) (= b true))
                (= mode*gate*a1 unknown))
           (:or (:and (= c true) (= o true))
                (:and (= c false) (= o false))
                (= mode*gate*a2 unknown)))
     (:and (:or (= a false) (= b false)
                (= mode*gate*a1 unknown))
           (:or (= o false)
                (= mode*gate*a2 unknown))))
```

At first inspection, this equation looks like any other Boolean equation, but it has several properties that simplify finding the cheapest assignment to mode variables satisfying the equation given current observations. First, the expression is an and-or graph formally called negation normal form (NNF). Second, note that the variables mentioned in each sub-expression of an `:and` expression do not appear in sibling sub-expressions. This form is formally

called decomposable negation normal form (DNNF) [5] and facilitates finding minimal cost diagnoses in time linear in the equation size.

With a DNNF equation, finding the minimal diagnoses is a matter of determining the minimum possible satisfaction cost of each sub-expression and then identifying the sources of the top level expression's cost. For instance, suppose that our `tst1` example outputted a true value when `a`, `b`, and `c` are respectively `true`, `true`, and `false`. As illustrated in Figure 3, the algorithm starts by costing the leaf nodes, where mode nodes get mode costs and sensed nodes get either zero or infinite costs depending on observations. In this case the checked nodes get zero costs. Once leaf costs are known each and/or node cost is the sum/min of its child nodes' costs. In the example, these costs appear above the nodes. Once the root node's cost is known, diagnosis goes back down the structure pruning nodes that do not contribute to the cost to determine what mode assignments can contribute to the cost. In this case the mode assignment is that component `gate*a2` is broken.

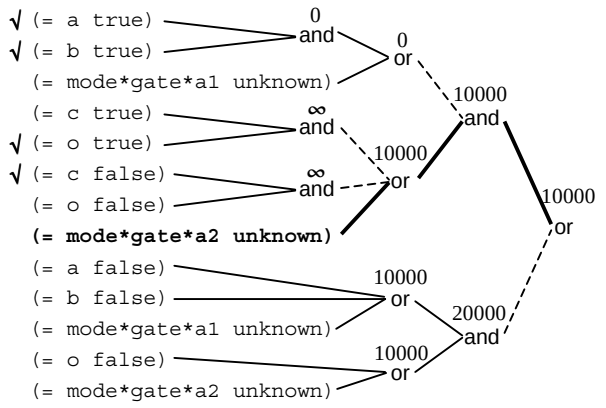

Figure 3 – Diagnosing example `tst1` system when `c` is false and all other inputs/outputs are true

The reason for the linear performance is that the cost of an and node can be determined by just looking at its children's costs. If the leaves of any and node's sub-expressions shared common variables, then the sources of costs would not be independent and the sum would be an over estimate of the actual and node's cost. Computing the actual cost would then become a lot more complex, and require time that was exponential in the sum of the sub-expressions' sizes. This is the ultimate motivation for computing DNNF Boolean expressions.

4. DISTRIBUTED SYSTEM DIAGNOSIS

Not only does the use of DNNF Boolean expressions facilitate real-time diagnosis, but it also facilitates distribution across multiple subsystems, like spacecraft. A CMPL system definition includes a specification of how the system is distributed into subsystems. Each subsystem specification defines the set of observables and commands that are local to each subsystem. We use this information to distribute the expression. We will use the following system example to explain the distribution method. This example differs from tst1 in two ways. It has two `and3i` modules and it partitions the sensed values across tree subsystems.

```
(defsystem tst2
  :subsystems
    ((s1 () (a b))
     (s2 () (d e f p))
     (s3 () (c o)))
  :connections ((boolean a) (boolean b)
                (boolean c) (boolean o)
                (boolean d) (boolean e)
                (boolean f) (boolean p))
  :structure ((and3i gate1 (a b c)(o))
              (and3i gate2 (d e f)(p))))
```

For this example the tree and its partitioning across the three subsystems appears in Figure 4. Deriving the partitioning starts with attributing a number to each subsystem, denoting its location in the subsystems list. Given these numbers, we associate each observable leaf with the subsystem that observes it. For instance `(= a true)` is associated with a 1, which is subsystem `S1`'s position. When the children of an and/or node are either unobservable or have an assigned number, the and/or node is assigned the maximum of the numbers assigned to its children. This process continues until each and/or node has an assigned number, and these numbers tell which subsystem is responsible for each node during diagnosis.

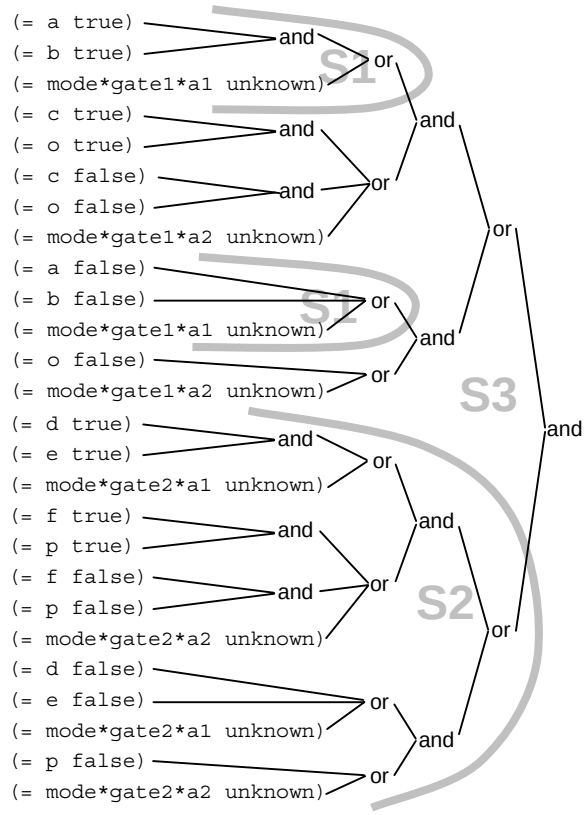

Figure 4 – Partitioning the graph for `tst2` system across three subsystems

In addition to illustrating an example distribution, Figure 4 shows an important property of the distribution in general. When two top-level modules can be diagnosed in isolation, their corresponding diagnosing sub-expressions are branches off of a top-level and node. Thus, examples like subsystem S2's being able to diagnose an entire substructure is a common occurrence, and this leads to the communications between subsystems for diagnosis being a function of subsystem interaction complexity instead of total system complexity.

5. INTERFEROMETER EXPERIMENT

In addition to testing DRMD on various circuit examples, we experimented with a Space Interferometer Mission Test Bed 3 (STB-3) model [6] 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 5, 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 also accurately points at the collector spacecraft in order to combine the starlight from the collector spacecraft with its own. This type of interferometer requires collaborative work between the two spacecraft, and due to the interdependencies between the two spacecraft, we must diagnose the system collectively. Thus, DRMD is well suited for such a system.

The FIT model has 17 components and 64 finite domain variables, where 12 variables are observable. The graph that represents the compiled FIT model consists of 4318 nodes. When distributed, the graph has 33 arcs that cross between spacecraft. With DRMD running on Allegro Lisp under Pentium III-M 750 MHz, generating the most-likely diagnoses from an observation takes between 80 and 500 msec depending on the observation. As all test scenarios verified, the number of messages passed between the combiner and collector spacecraft was linear in the number of cross spacecraft arcs. We also tested DRMD on a subset of the FIT model that represents the angular metrology of FIT. Angular metrology is responsible for properly aligning the two spacecraft, and the angular metrology model is composed of 4 components and 12 variables of which 3 are observable. When distributed, the angular metrology graph had 45 cross-spacecraft arcs. Running various diagnosis scenarios on the angular metrology system also showed that the number of messages passed between the two spacecraft was linear in the number of cross arcs. This result verifies that the number of communication among subsystems is a function of subsystem interaction complexity, not the total system complexity.

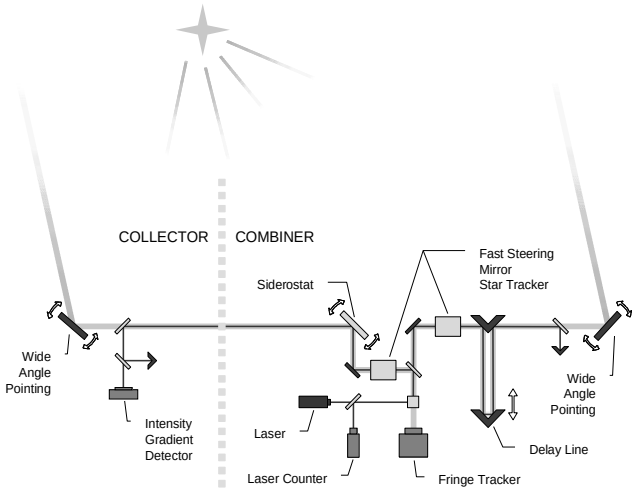

Figure 5 – Simplified schematic of the Formation Interferometer Testbed (FIT). The left side of the dotted line represents the collector spacecraft and the right side of the dotted line represents the combiner spacecraft.

6. CONCLUSION

This paper presented a knowledge-compilation based approach toward implementing an onboard model-based diagnosis system that both runs in real-time and easily distributes across a number of processors on one or more spacecraft. Past model-based diagnosis systems, like Livingstone, search for the most likely mode by testing a number of modes against an internal model representation. Our approach further processes this internal representation to determine a form that can be evaluated in linear time to find the most likely mode with an extremely simple algorithm.

While our work is based on the knowledge compilation work of Darwiche *et al.*, there has been other work on offline compilation of Livingstone models with a system called Mini-ME [7]. In this work, a model is compiled into a mapping from conjuncts of sensed values to disjuncts of mode estimates. This mapping is then used online to compute disjunctive sets of mode estimates from observations, and these sets are processed to compute the actual mode estimate that optimally satisfies all disjunctive sets. This approach is very different from the one presented here, and we have yet to make a principled comparison. Mini-ME can be faster due to computing a small number of disjunctive sets for mode estimation, but the optimal satisfaction of a number of disjunctive sets is an NP-complete problem requiring a heuristic algorithm to avoid exponential performance as much as possible. On the other hand, the equation compiled in DRMD allows a linear time evaluation to provide a mode estimate, but its size can be exponential in the number of components depending on sensor placement and the system's topology. Regardless of the relative performance expectations, DRMD was specifically targeted for distributed applications, which Mini-ME does not address.

Finally, our current implementation only uses mode costs to estimate modes given a set of observed variables, and transitions are ignored. This results in a system that ignores previous mode estimations when computing a new estimate. As future work we plan to adapt a belief update approach, like that used in a Kalman filter, to folding the last estimate into computing the next one. In this way, transitions and current mode estimations combine to provide *a priori* mode costs of the next modes to facilitate determining the next mode estimations from observations.

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 authors would also like to thank Adnan Darwiche, Daniel Dvorak, and Mitch Ingham for discussions contributing to this effort.

REFERENCES

[1] Brian C. Williams and P. Pandurang Nayak, "A Model-based Approach to Reactive Self-Configuring Systems," *In Proceedings of the National Conference on Artificial Intelligence*, 1996.

[2] A. Darwiche, "Compiling Devices: A Structure-Based Approach," *In Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 1998.

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

[4] P. Nayak and B. Williams, "Fast Context Switching in Real-time Propositional Reasoning," *In Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, Wisconsin, 1998.

[5] A. Darwiche, "Decomposable Negation Normal Form," *Journal of ACM*, July 2001.

[6] M. Ingham, B. Williams, T. Lockhart, A. Oyake, M. Clarke, and A. 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.

[7] S. Chung, J. Van Eepoel and B. Williams, "Improving Model-based Mode Estimation through Offline Compilation," *International Symposium on Artificial Intelligence, Robotics and Automation in Space*, St-Hubert, Canada, June 2001.

Seung Chung *is a Graduate Research Assistant at the MIT Space Systems and Artificial Intelligence Laboratories, working with Prof. Brian Williams. He received his Bachelor of Science degree from the Department of Aeronautics and Astronautics Engineering at the University of Washington in 1999. He is completing a Master of Science degree at MIT and has been awarded a NASA Graduate Student Research Program fellowship for his doctoral studies. His current research includes the development of Titan, a model-based executive capable of autonomously estimating the state of spacecraft, diagnosing and repairing faults, and executing commands. In 2002, he worked with the Artificial Intelligence Group at the NASA Jet Propulsion Laboratory on the development of a distributed model-based executive.*

Dr. Anthony Barrett *is a senior member of the Artificial Intelligence Group at the Jet Propulsion Laboratory, California Institute of Technology where his R&D activities involve planning, scheduling, and diagnosis applied to controlling constellations of spacecraft. He holds a B.S. in Physics, Computer Science, and Applied Mathematics from James Madison University and both an M.S. and PhD in Computer Science from the University of Washington. His research interests are in the areas of planning, scheduling, executives, and multi-agent systems.*