Model-based Software Tools for Integrated Vehicle Health Management

Gabor Karsai, Gautam Biswas, Sherif Abdelwahed, Nag Mahadevan, Eric Manders
Institute for Software-Integrated Systems,
Vanderbilt University
Nashville, TN 37235
{gabor,biswas,sherif,nag,ericM2}@isis.vanderbilt.edu

Abstract

Present day IVHM systems are often constructed using hand-written code that is hard to produce and difficult to verify and maintain. In this paper we introduce a suite of model-based tools that allow for the construction of embeddable IVHM applications using a model-based approach. Reusable (and potentially validated) reasoners are used in conjunction with executable code that is generated from models, thus allowing the integration of the reasoner as a component into a larger on-board system. This paper describes the toolsuite, the modeling approach used, the run-time environment, and some of the applications where the tools were used.

1. Introduction

Integrated Vehicle Health Management systems today (like the AHM on the Space Shuttle) are often built using traditional programming languages and tools that ensure performance but require significant effort in specification, design, implementation, verification, and maintenance. IVHM systems include significant software components, which may provide input to flight critical systems (e.g., engine shutoff), thus their reliability is of utmost importance. These software components are built using traditional procedural languages, and they have to go through the same V&V processes as any other flight critical software. Furthermore, any change in the underlying hardware necessitates changes in the software, resulting in a costly re-implementation and re-certification. Thus, the design, implementation, verification, and evolution of these systems can be very expensive, and the cost-benefit of onboard IVHM systems is often hard to justify.

Model-based software development that relies on the use of domain-specific modeling languages, model analysis, and automatic code generation from the models is considered one of the promising directions in embedded software development. Model-based development processes and tools are rapidly gaining acceptance in the industry, even in safety-critical systems like flight control. Modern flight control systems today are often prototyped and tested by simulation in model-based tools (Matrix-X and Simulink/Stateflow being the two most relevant examples), and then embedded code is generated from the models for deployment onboard the target platform.

In this paper we describe an approach, called Fault-Adaptive Control Technology (FACT) that uses model-based development tools, software generators, and reusable run-time reasoner components for building on-board IVHM systems. This framework provides a model-based software development approach, and it integrates with other model-based tools as well. The tools and the technology have been evaluated in a set of prototype applications that are discussed in section 4.

2. Background

Integrated Vehicle Health Management (IVHM) has been used in the aerospace industry for 30+ years. Its goal is to provide better ways for operating and maintaining aerospace vehicles using techniques, such as condition monitoring, anomaly detection, fault isolation, and managing the vehicle operations in the case of faults [1]. In general, this includes a large number of sub-disciplines (e.g., sensor technology, signal processing algorithms, and robust control techniques), but our focus in this paper is on the software aspects of designing and implementing these systems.

The variety of tasks covered by the IVHM framework (monitoring, detection, feature extraction, isolation, identification, and consequence analysis) implies the need for significant diversity in the IVHM software components. A recent industry initiative, “Open System Architecture - Condition Based Maintenance” (OSA-CBM) [2] was targeted toward defining a framework for constructing and organizing software

and systems for condition-based maintenance – a close relative of IVHM. The OSA-CBM development architecture standard serves as a reference for constructing layered software products that solve problems similar to the ones arising in IVHM systems, including data acquisition, data manipulation, condition monitoring, health assessment, prognostics, decision support, and presentation. Note that OSA-CBM defines the interfaces between these layers but leaves the implementation of the functions open to the use of different algorithms and procedures.

An important component of the IVHM function is related to fault detection and fault isolation. Here we can only hint at the vast literature and the diversity of techniques in these fields that come from well-established disciplines like signal processing, dynamic system theory, control theory, and automated symbolic reasoning and search (subfields within "AI"). Note that while a large number of algorithms and techniques are available, very little has been published on the software implementation, integration, and run-time environments for these systems.

Our work develops a model-based software development toolsuite for constructing embedded IVHM applications. We have used a specific sort of model-based development approach called Model-Integrated Computing (MIC) [3]. MIC is a variant of the Model-Driven Architecture (MDA) variant of the Object Management Group (OMG) that focuses on the use of domain-specific modeling languages (DSML-s), model analysis tools, model transformations (including code generation), and model-integrated toolchains. *Domain-specific modeling* implies that the models used to analyze, (possibly) generate, and integrate the system are expressed in languages that are based on the concepts of the engineering discipline that the application is created for. Specifically, in the case of IVHM, the DSML(s) should support the modeling of dynamic systems from the viewpoint of the health management. Note that a number of system-level modeling languages exist, but they are not well-suited for modeling systems for IVHM applications. *Model analysis* uses the domain-specific models for design-time analysis, facilitated by simulators, symbolic and mathematical analysis techniques (e.g., model checking and theorem proving). *Model-based code generation* is the process of transforming models into executable code. While it is doubtful that automated code generation from models will completely replace manual code development in algorithmic languages, it seems useful for specific, well-structured domains (e.g., controllers specified using Statechart-like notations), and for interface (a.k.a. "glue") code that is not hard but tedious to write by hand. Code generation can be looked upon as a specific version of model transformations that translate and transform models while preserving model semantics. Model transformations also play an essential role in forming *tool chains* for a set of tools that are each generated using model-based development tools, but each tool may employ a different modeling language. For instance, a design tool may be based on a dedicated design language (e.g., Statecharts) but model analysis tools could use their own notation and language (e.g., Promela in SPIN) — which necessitates a model transformation when theses tools are used in an engineering process.

MIC has been defined, developed, and used over the past fifteen years, and it is facilitated by a suite of meta-programmable tools that support the construction of DSML-s, model transformation tools, and integrated toolchains. The meta-tools and example toolchains are described elsewhere [3], and are available through the Escher Institute[1]; a non-profit organization that hardens and disseminates research products from government programs.

3. The FACT toolsuite

The FACT toolsuite [4], developed using MIC meta-programmable tools, is a model-based software development environment for constructing IVHM application. Not all aspects of the IVHM software development are addressed in the current version of the FACT paradigm. In this paper, we focus mainly on the functionality for (a) model-based observation and tracking, (b) fault-detection using statistical techniques, (c) fault-source isolation using symbolic reasoning, (d) fault-magnitude estimation, and (e) reconfigurable control via switching among pre-determined alternative control laws. The toolsuite is constructed such that in the final IVHM application a toolchain can be constructed that includes one or more of all of the above capabilities.

FACT has two main components: (1) a design-time environment for modeling and model transformation, and (2) a run-time environment consisting of the model-based reasoner components. A Windows desktop version f the runtime environment can be used for simulation-based experiments. Synthesized code for embedded boards is used for online processing on the actual system. The notional architecture of FACT is shown on Figure 1.

Specifically, FACT includes (1) a graphical modeling environment that supports a multitude of modeling languages for capturing plant and controller models, (2) a set of program generators that "compile" the

[1] www.escherinstitute.org

models into executable code, and (3) a run-time component that includes (a) a hybrid observer, (b) configurable fault detectors, (c) two diagnostic reasoners (one that operates in continuous-time but accommodates discrete changes in system models , and a second that operates with timed discrete-event models) that could be used separately or in conjunction, (d) a controller reconfiguration manager, and (e) a small footprint dataflow kernel that coordinates the operations of the various components. We briefly describe each of these components below.

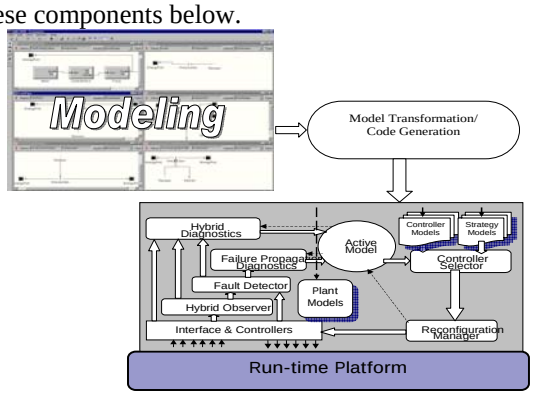

Figure 1. The FACT architecture

3.1 Modeling in FACT

FACT models contain two parts. One is geared toward physical plant representations that facilitate diagnosis and control. The second component is geared toward modeling the plant controllers. Two plant modeling forms are employed that correspond to the two independent fault isolation methods: one based on hybrid bond-graphs [5], and a second based on fault propagation graphs [6]. Both languages describe the plant in terms of hierarchically structured components. The two languages share a common component hierarchy representation for the plant model, but they represent different aspects of component behavior at each level of the hierarchy.

The approach based on hybrid bond graphs (HBG) represents the physical plant as an interconnected set of junctions that can be switched *on* and *off*, effort and flow sources that define mechanisms for energy transfer in and out of the system, resistive that model the dissipative behaviors in the system, energy storage elements that define the time-varying characteristics of system behavior, and transformers and gyrators that model transformations between different energy domains. An example is shown on Figure 2. The modeling approach is based on the bond-graph modeling technique [7]: a physics- and energy-based, domain-independent modeling approach for dynamic systems. There are two extensions in FACT beyond the basic bond-graph approach: we support non-linear behaviors by allowing component parameter values to be functions of the value(s) of state variables, and we support junction switching, which changes the model configuration depending on whether the junction is on or off. This latter feature is highly productive for representing systems that undergo discrete changes, such as a valve turning on and off. As mentioned above, bond graphs represent energy flows, however, in FACT information flows (controlling junctions and influencing non-linear elements) are also explicitly modeled. Note that other modeling approaches, e.g., signal flow diagrams can easily be converted into bond graphs, and other dynamic system models, e.g., state-space representations can be easily derived from the bond graph models. We make the assumption that all faults and degradations of interest are captured as component parameters in the hybrid bond-graph model, therefore, all fault hypotheses are captured as parametric faults, i.e., changes in one or more parameters of the model. For example, increase resistance, decreased capacitance, changes in the transformer coefficient define fault hypotheses, and the objective of the fault diagnosis is to isolate the faulty parameter and identify its specific value.

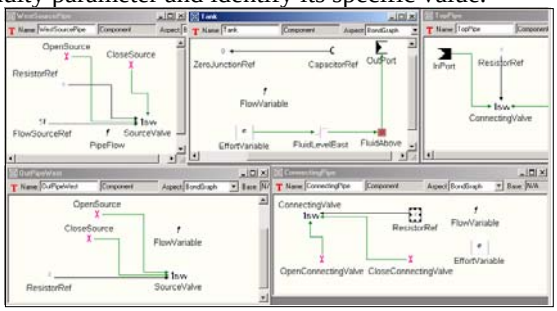

Figure 2. Example HBG model

The approach based on timed failure propagation graphs (TFPG) [6] uses a model where component failure modes and their effects, called discrepancies, are enumerated. Failure modes belong to components and cause discrepancies which may lead to other discrepancies, thus, produce the effects of the failure mode cascading through the system. The assumption here is that (the majority of) discrepancies are detectable by one or more observations. The causal links between failure modes and discrepancies, and between pairs of discrepancies, may be mode-dependent, and could have a simple dynamics associated with them. The "simple dynamics" means that the modeler can specify minimum and maximum propagation times for each link. The fault assumption here is that components fail with known failure mode(s), and the effect of these faults are detectable through (hardware and software) monitors that detect specific discrepancies.

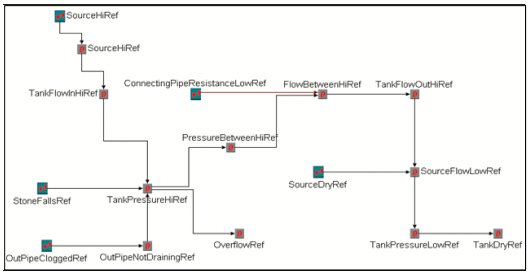

Figure 3. Example TFPG model

The modeler can use either or both modeling approaches to construct plant models. Note that the HBG-approach requires a high-fidelity model of the plant, while the TFPG-approach uses a simpler and coarser causal model that can usually be generated by design or plant engineers. An interesting issue is to link failure modes defined by the HBG model based on physics models, and TFPG failure modes that are specified by human experts, and study the integrated results from the two reasoners.

Beyond modeling the physical plant, the FACT modeling language also support modeling of reconfigurable controllers. Such controllers are modeled using a signal-flow, block-diagram oriented paradigm, where the configuration of block diagram is determined via a controlling finite state machine (CFSM). Blocks in the diagram represent individual controller algorithms whose behavior is specified using either procedural code or using a FSM-like modeling language. The reconfiguration implies a topological change in the block diagram, and this is captured by assigning various sub-graphs (blocks and connecting 'wires') to each state of the CFSM. A state transition in the CFSM can be linked to failure-modes identified in the HBG or in the TFPG, with the following semantics: ***if*** *failure mode* $<F_i>$ *is identified as the fault source* ***then*** *trigger transition* $<T_j>$ ***and*** *potentially reconfigure the controller structure*. Figure 4 shows an example controller model.

The modeler creates models for the plant and the controllers (as needed) using the DSML of FACT, which is supported by MIC's meta-programmable visual modeling environment called Generic Modeling Environment (GME) [8]. These models are then compiled into either a compact loadable file, or executable code. The purpose of these is to reconstruct the model data structures in the run-time environment. The loadable file is used whenever file storage is available for the run-time system (e.g., in a desktop experiment), while the executable code is used whenever the "model construction" code needs to be compiled into the run-time environment because of the lack of a file system (e.g., an embedded board).

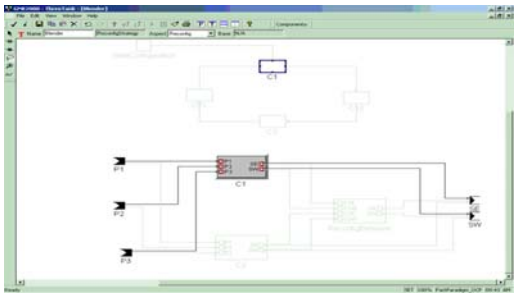

Figure 4. Example controller model

3.2 Run-time environment

The FACT run-time environment consists of a number of generic software components that are instantiated and configured according to the models. These components could be used as standalone components (e.g., a TFPG-based reasoner), or under the control of a small, component integration platform: a dataflow kernel that schedules component execution. Figure 5 shows the details of the FACT run-time system. We describe the main components and how they are implemented below. Figure 5 shows the details of the FACT run-time system.

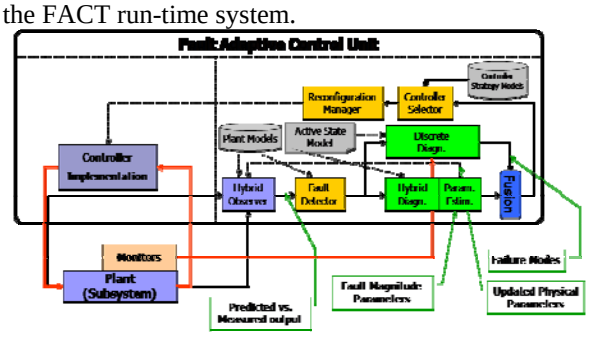

Figure 5. FACT run-time system

The FACT run-time system includes a hybrid observer (HO) that tracks the continuous plant behavior, and also across discrete mode changes, which correspond to changes in the junction state of the HBG. Changes in junction configuration modify the model topology, and corresponding to each bond graph topology, a new set of state space equations is derived. The HO essentially implements a multi-model Extended Kalman filter (the system models are nonlinear), with a different continuous model for each mode (junction states) of operation. When a mode change occurs, the filter is switched to a different set of equations. The HO is automatically and symbolically derived at the time when the run-time system is initialized with the models. Plant inputs and plant outputs are both fed to HO, and some of these signals are used to trigger mode changes in the filter. Obviously, the HO requires the presence of HBG models. The HO main-

tains an "active state model," which is the best current estimate of the model parameters, state variables and mode of the plant.

The HO produces predictions for the plant's state, and these are continuously compared against observations. If there is a significant difference detected between the predicted and the observed behavior, then the fault isolation process is initiated. The fault detectors act as the decision making units for triggering fault isolation. To ensure robustness, and to reduce false alarms, a statistical hypothesis testing algorithm based on the Z-test is employed for the detection of deviations for each measured signal. The algorithm is highly parameterized, but can also be substituted by user-supplied fault detection code. Alternatively, if HBG models for the plant are not available, the TFPG fault isolation scheme by itself can be triggered by fault detection events that are generated from the "monitors," which are custom hardware and/or software elements attached directly to plant equipment, and generate "alarm" signals when discrepancies are observed.

As mentioned above, FACT includes two fault isolation algorithms. One of the algorithms is based on the HBG-s, and it uses a qualitative/symbolic reasoning technique for initial fault isolation. The reasoning tracks the qualitative dependencies among plant state variables, and uses the deviations (and their derivatives) detected between the expected and observed plant output. The reasoning process requires typically a few characteristic measurement deviations (derived from the HBG model as a fault signature) to isolate the parameter of the physical system whose deviation caused the fault. In many cases, due to the ambiguity in the qualitative scheme, all physical faults (i.e., parameter changes) may not be uniquely isolable because they generate similar symbolic signatures for all of the available measurements. As a result, the qualitative reasoner produces a set of candidate failure modes. This candidate set is further pruned down using a numerical system identification (SI) process. For every fault candidate, we start a separate SI thread that uses collected data, and computes the value of each of the hypothesized deviated plant parameters from the data. The deviated parameter value that produces the least error in tracking the measurements is considered to be the real candidate, and the estimated parameter value is considered the new, "faulty" parameter value for the plant. This faulty value is written back into the "active state" model in the HO, and once this update is performed, we have shown that the HO is able to track the faulty plant with small error.

The second algorithm executed in parallel uses the TFPG algorithm, which receives time-stamped data from the fault detectors (or monitors) that indicate the discrepancies that have been detected. Using a propagation algorithm on the TFPG model, the most plausible explanation for the discrepancy sequence observed is derived in terms of specific failure modes in components. The algorithm is "*robust*," i.e., it can tolerate missing (undetected) or false (erroneously indicated) discrepancies. Edges in the TFPG could be dependent on operational modes of the system: the algorithm tracks and analyses these changes as well. Note that the TFPG has structural and temporal constraints that discrepancy sequence must satisfy, and this information can be used to eliminate unlikely failure modes. The TFPG reasoner also generates a set of failure mode candidates, but these are ranked according to plausibility metrics defined in terms of the TFPG model.

If both the HBG and TFPG reasoners are present, their result is combined using a straightforward fusion algorithm: failure modes isolated by both reasoners get the highest likelihood, while failure modes generated by only one reasoner are discarded. If only one reasoner is available, its results are used as the true fault candidate. In either case, the result of the fault isolation process is made available through an API for other software components to use.

The isolation fault can be used to trigger a reconfiguration in the control system, if control models are provided. Initially, the controller models are compiled and instantiated, such that one of the alternative controllers is active. This controller operates with the assistance of the run-time kernel that acts as a dataflow scheduler for the controller blocks. The CFSM is compiled into executable code that implements the state transition and reconfiguration logic. The reconfiguration happens when such a fault is isolated that has a corresponding transition from the current, active state in the CFSM. When such a transition is found, the new state is determined and the wiring of the controller blocks is changed, under the control of the dataflow kernel. For mitigating the reconfiguration transients, we have experimented with a number of strategies for initializing and transitioning the state of the controller blocks, some of which have been modeled as reconfiguration strategies.

3.3 Using FACT

The FACT toolsuite is a model-based development environment for constructing IVHM applications. It provides tools for debugging and experimenting with an IVHM application, and it also has tools for generating runtime code for deployment on an embedded plat-

form. Figure 6 shows how the FACT tools could be used.

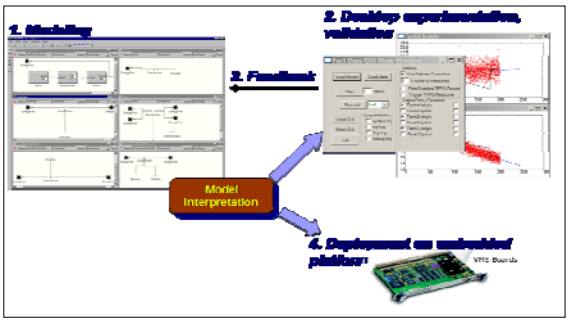

Figure 6. Using the FACT tools

Once the models are constructed, the designer can compile them into a form suitable for use in a desktop experimentation environment. The desktop environment allows the processing of input/output data files and running the HO, and the reasoners. Testing the reconfigurable controllers is also possible, but it requires some coding that integrates the library containing the FACT run-time components with real-time data streams.

Lessons learned from the desktop experimentation typically used in improving the models. Once this process converges, the executable code for the embedded platform is generated and the application tested on the real hardware.

4. Application experience with FACT

FACT has been tested on a number of example systems. Below we highlight the main results and discuss the experiences.

4.1. Academic example: The 3-Tank system

One example system FACT has been tested on is the classic 3-tank system. The notional schematic of the system is shown on Figure 7.

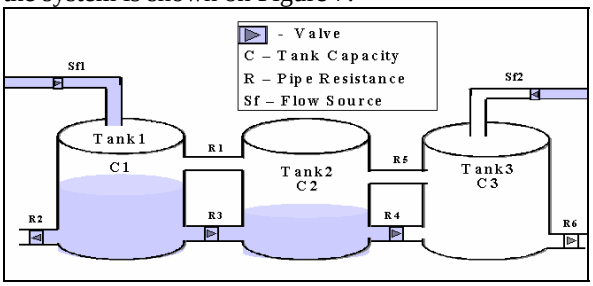

Figure 7. 3-tank system

The 3-tank system consists of 3 interconnected tanks that hold some fluid. The anticipated failure modes of the system include: tank capacity change (something is dropped in to the tank), pipe blockage, and fault in the input feed.

The construction of the model was straightforward, although the system has been modeled in the past exceedingly. For model parameter estimation we have collected data from the real physical implementation (available in the lab at Vanderbilt), and used a standard system identification techniques for model parameter estimation [9]. Once the HBG model was constructed and identified, we have shown that the HO can indeed track the plant, both in the continuous and discrete modes. The tracking error was typically less than 5% in the continuous modes and a little higher when mode transitions occurred. Next, we collected data from the real system that included faulty behavior, by inducing faults in the physical system. The "faulty" data was used to fine tune the fault detectors and ensuring that they trigger on the right event(s).

We ran a number of experiments with faults in either tanks 1 and 2 or blocks in the connecting and drain pipes. The measured values were the three tank heights. For capacity faults (C_1^-, C_2^-) the qualitative fault isolation scheme was able to isolate the fault uniquely in less than 10 time steps (sampling rate = 1 sec), whereas for the resistive fault, a block in the outlet pipe from tank 1 (R_1^+), qualitative fault isolation reduced the number of candidates to two (R_1^+, R_{12}^-), i.e., block in Tank 1 outlet pipe or leak in the transfer pipe between tank 1 and 2. After 50 time steps the SI module was initiated, and it estimated the true fault with the fault parameter value of (*2x*), i.e., the block fault doubled the drain resistance of the pipe.

4.2. Simple example: Simple aircraft fuel system

A second, more realistic example is a generic fuel transfer system for fighter aircraft, illustrated in Figure 8. The system is designed to provide uninterrupted supply of fuel at a constant rate to the aircraft engines while maintaining the center of gravity of the aircraft.

The system is symmetrically divided into left and right parts (top and bottom in the schematic). The four supply tanks (Left Wing (LWT), Right Wing (RWT), Left Transfer (LTT), and Right Transfer (RTT)) are full initially, and so are the two receiving tanks (Left Feed (LFT) and Right Feed (RFT)) that directly feed the engine. During engine operation, fuel is transferred from the supply tanks through a common manifold to the two feed tanks in a sequence determined by the

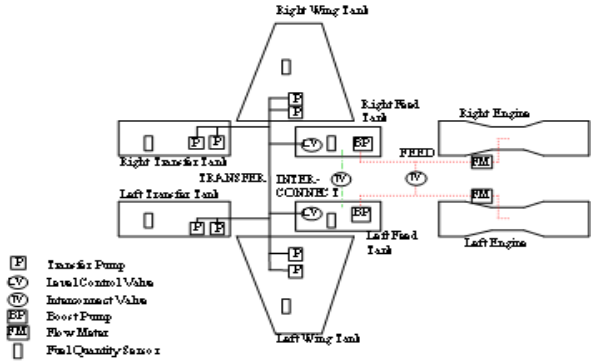

Figure 8. Simple fuel transfer system

fuel system controller. The controller generates on/off signals for the pumps in the supply tanks, and the valves in the pipes to achieve different flow configurations.

Table 1. Fuel system experiments with different fault magnitudes and noise levels

Faults		Performance Parameters							
Fault Type	Fault Magnitude	Fault Detection Time (seconds)		Fault Isolation Time (seconds)		Initial/Final Candidate Set number/number		Parameter Estimation Error (percent)	
		2%	3%	2%	3%	2%	3%	2%	3%
LTT-Pump Efficiency Drop	33%	422	555	225	398	14/3	13/4	2.19	5.43
	60%	182	183	144	240	13/4	13/4	1.28	1.79
	80%	134	134	124	197	13/4	13/5	0.88	1.49
RWT-Pump Efficiency Drop	33%	117	285	170	211	13/4	13/4	2.15	6.11
	60%	83	93	139	183	13/4	13/4	1.52	1.67
	80%	5	5	55	106	13/3	13/4	0.68	0.68
RLCV-Block (Valve)	× 1.5	63	65	97	103	25/2	25/2	0.62	0.5
	× 1.75	51	63	58	86	25/2	23/1	0.28	0.46
	× 2.0	51	52	46	79	23/1	25/2	0.2	0.2
Leg21 Pipe Block	× 1.5	99	100	136	350	14/3	14/3	1.58	1.65
	× 1.75	95	95	90	303	14/2	14/3	0.78	1.57
	× 2.0	93	93	76	202	14/2	14/2	0.19	0.34

Table 1 illustrates the results of a set of diagnosis experiments that we ran for a set of faults using the HBG scheme. In the experiments, we varied the fault size and amount of measurement noise in the signal. In designing the experiments, we had to set parameters for the Kalman filter, fault detector, and symbol generator. A high fidelity simulator was used to generate the data for the experimental runs, and measurement noise was added to the simulated data. Ten runs were conducted for each noise level and fault size, and the mean values of the detection and isolation times, the candidates generated by qualitative fault isolation, and the parameter value error after least squares estimation are reported in the table. The results indicate that as the noise levels in the measurements increase, and the fault magnitudes become smaller, the time to detection, isolation, and identification (i.e., parameter estimation) increase, and the parameter estimation error increases. Experiments conducted with the TFPG diagnoser produced similar results but for lack of space are not reported here.

4.3. Complex example: Generic fuel system

As a scaling up project, a comprehensive TFPG model of the Generic Fuel System, which is comparable in complexity to models of real-life systems, was created using the FACT modeling language. The TFPG model is illustrated in Figure 9. In addition to capturing the failure propagation across the main sub-sub-systems – Left & Right Fuselage Tanks, Left & Right Wing Tanks, Transfer Manifold, and Left & Right Feed Tanks – in great detail, this model also captures the failure propagations associated with the power and control elements in the system. This model includes 153 Components, 481 Failure Modes, 1973 Discrepancies, 270 Alarm Monitors, 9 Modes, 3409 Failure Propagation links (555 of them with Activation Functions described by Mode States). This represents a significant scale-up when compared to the TFPG model of the simple Fuel System which had 15 Components, 17 Failure Modes, 35 Discrepancy, 70 Failure Propagation Links, 14 Alarms, and 6 Modes.

The initial system was tested exhaustively on the data generated from an independent simulator for about 250 fault scenarios. In all cases the list of "Most Probable Faults" from the reasoner included the "real" fault source(s) / failure mode(s). The worst case response time on a desktop AMD-Athlon XP(2400+) 2 GHz Processor with 448 MB RAM was around 0.1 seconds while that on a PowerPC 750, 400 MHz, 256 MB RAM it was around 0.3 seconds.

Figure 9. TFPG model of generic fuel system

In order to validate and monitor the performance of the system, a simulator has been designed that generates data sets of alarm and mode events, given a model and a timed – sequence of faults (failure modes). The data sets are fed to the reasoner and the reasoner output

is verified against the initial fault sequence. This procedure is useful in validating the design model and improving the model to reduce the ambiguity group in the final set of hypothesis.

4.4. Real-time example: The SEC flight experiment

In a DARPA project demonstration, the FACT diagnosis engine was used to detect and diagnose faults in actuator faults in a GTMAX helicopter experiment run at Georgia Tech. The result of the diagnoser was fed to a reconfigurable flight controller. Actuator data – controller output, actuator position – obtained from GTMAX was fed to the HBG-based diagnosis engine, and run as Software in the Loop (SITL), as well as Hardware in the Loop (HITL) experiments. For HITL, the FACT diagnosis library was ported to QNX – the operating system on the GTMAX processor. Initial experiments performed with the GTMAX simulator (SITL) helped fine tune the fault detectors, and this avoided false positives in the HITL runs.

Both in SITL as well as HITL, the stuck actuator faults were detected, isolated, and estimated within 2 seconds of the inception of the fault. The quick detection and accurate estimation greatly helped the performance of the reconfigurable controller. Even in the presence of heavy wind gusts and other measurement noise, the detection and diagnosis results were robust, and activation of the reconfigurable controller was completely within 3-4 sec. The number of false positives was less than 5% (about 1 in 20).

5. Summary and conclusions

We have shown a model-based software development tool for constructing IVHM applications. The tool supports the modeling of the physical plant and its controllers, the transformation of the models into a form suitable for execution on an embedded platform, and the run-time execution of hybrid state estimation, fault detection, fault isolation and fault estimation, and reconfigurable control. The tool has been applied to a number of examples and this illustrates the power of the model-based software development paradigm.

The FACT paradigm just represents a few of the tools used by IVHM engineers. For instance, FMECA databases, testability analysis tools, fault simulators, and cost estimation tools are all useful for constructing ISHM applications. We have already started working on developing additional tools and the resultant tool integration issues. For instance, we can import component/failure mode/test function data from a testability tool. We are also able to generate Simulink/Stateflow simulations from the FACT/HBG models. Another potential growth area is to provide tools diagnosability analysis. Both the HBG and TFPG models lend themselves to analytical methods that predict the quality and performance of the fault diagnosis process.

Yet another extension could be to improve the reasoners that they use anytime algorithms, and can gracefully degrade under resource constrained situations (on the embedded platform). We plan to investigate these and other issues with FACT, in order to grow the toolsuite that supports a number of engineering activities needed in IVHM.

6. References

[1] L. Melvin, et al., "Integrated vehicle health monitoring (IVHM) for aerospace vehicles," *International Workshop on Structural Health Monitoring* Stanford, CA, pp. 705-714, Sept. 1997.

[2] M. Lebold and M. Thurston, "Open standards for Condition-based Maintenance and Prognostic Systems," Maintenance and Reliability Conference (MARCON), May, 2001.

[3] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proc. of the IEEE*, **91**(1), pp. 145–164, Jan. 2003.

[4] G. Karsai, et al., "Towards fault-adaptive control of complex dynamical systems," in *Software-Enabled Control – Information Technology for Dynamical Systems*, eds., T. Samad and G. Balas, pp. 347–368, Wiley-IEEE press, NJ, 2003.

[5] P.J. Mosterman and G. Biswas, "A theory of discontinuities in physical system models', *Journal of the Franklin Institute*, vol. **335B**(3), pp. 401–439, 1998.

[6] JR Carnes, A Misra, and J Sztipanovits, "Model-integrated toolset for fault detection, isolation and recovery (FDIR)," *IEEE Symposium and Workshop on Engineering of Computer Based Systems*, 1996.

[7] D. C. Karnopp, D. L. Margolis, and R. C. Rosenberg, *Systems Dynamics: Modeling and Simulation of Mechatronic Systems*, 3rd ed. John Wiley, NY, 2000.

[8] A. Ledeczi, et al., "Composing Domain Specific Design Environments," *IEEE Computer*, vol. 34(11), pp. 44-51, 2001.

[9] J. Wu, et al., "A Hybrid Control System Design and Implementation for a Three-tank Testbed,'' *IEEE Conf. on Control Applications*, Toronto, CA, pp. 645-650, Aug. 2005.