Toward an Equation-Oriented Framework for Diagnosis of Complex Systems

Alexander Feldman Gregory Provan

University College Cork, Ireland

a.feldman@ucc.ie, g.provan@cs.ucc.ie

Abstract

Diagnosis of complex systems is a critical area for most real-world systems. Given the wide range of system types, including physical systems, logic circuits, state-machines, control systems, and software, there is no commonly-accepted modeling language or inference algorithms for model-Based Diagnosis (MBD) of such systems. Designing a language that can be used for modeling such a wide class of systems, while being able to efficiently solve the model, is a formidable task. The computational efficiency with which a given model can be solved, although often neglected by designers of modeling languages, is a key to parameter identification and answering MBD challenges. We address this freedom-of-modeling versus model-solving efficiency trade-off challenge by evolving a language for MBD of physical system, called LYDIA. In this paper we report on the abilities of LYDIA to model a class of physical systems, the algorithms that we use for solving MBD problems and the results that we have obtained for several challenging systems.

Keywords model-based diagnosis, model-based testing, automated reasoning, modeling language

1. Introduction

Diagnosing complex systems using a model-based diagnosis (MBD) approach has led to the development of several languages, most of which are extensions of simulation languages. Examples include logic-based diagnosis languages (which extend multi-valued logics), e.g., [20]; bond-graph diagnosis languages (which extend bond-graphs), e.g., [21]; and MODELICA-based diagnosis languages (which extend MODELICA), e.g., [2].

Each of these languages has strong and weak points. The logic-based diagnosis languages have a well-defined diagnosis semantics and cleverly-designed inference algorithms, but are limited in the types of behaviours that they can be described by logic-based constraints.

The bond-graph and MODELICA-based diagnosis languages can describe a wider class of systems, but do not have a well-defined diagnosis semantics or efficient diagnostics inference algorithms. For example, MODELICA focuses on a single nominal mode of a system[1], whereas a diagnosis language must be able to specify behaviours for all the pertinent nominal and faulty modes of the system.

In this article we propose a diagnosis framework that aims to provide the expressive power of a MODELICA-based diagnosis language together with a clear diagnosis semantics and efficient diagnostics inference algorithms. The language, LYDIA, is a component-based, hierarchical language that supports dynamical systems (based on ODEs) as well as logical constraint-based representations. In addition, the framework provides a range of simulation and fault-isolation algorithms. Hence, LYDIA provides many of the simulation-oriented capabilities of a Modelica implementation together with the associated mode-identification and fault-isolation algorithms. Further, LYDIA allows modes to be identified by their likelihood of explaining the observed data, using a variety of likelihood metrics.

Our contributions are as follows:

1. We describe a modeling language, LYDIA, that supports the specification of mode-based behaviours for nominal and faulty models;
2. We describe a framework that can simulate, identify modes and isolate faults for models described in the LYDIA language;

2. Related Work

This section compares our approach to that of some key existing equation-oriented systems/languages, e.g., MODELICA (`http://modelica.org/`), bond graphs (`http://bondgraph.org/`), and Rodelica [5].

2.1 Diagnostic Approaches

MODELICA is a mixed declarative/procedural language; see, e.g., [12]. The declarative aspect involves the dynamical systems equations; the procedural aspect involves the specification of code

[1] Of course, it is possible to have multiple modes in MODELICA, however, the price is often increased modeling and simulation complexity.

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.

 Proceedings available at: `http://www.ep.liu.se/ecp_home/index.en.aspx?issue=084`

EOOLT 2013 website: `http://www.eoolt.org/2013/`

fragments within the model itself. It is noted in [12] that the semantics of MODELICA is defined not just by the declarative aspects of the model, but also by the compilation of the model that aims at optimizing simulation efficiency. In other words, semantics is also provided by the process of translation of a hierarchical model, which consists of a hierarchically-nested set of classes, instances and connections, into a flat model, consisting of a set of constants, variables and equations. In the flat model, an optimization/compilation step performs several operations, including sorting and conversion of equations to assignment statements. Next, strongly-connected sets of equations are solved using symbolic and/or numeric solvers.

In contrast to MODELICA, the LYDIA language is fully declarative. For example, whereas MODELICA allows users to define procedural entities (code fragments) within the model itself, LYDIA only allows declarative statements developed from syntactically correct language statements. Allowing in-model algorithm specification is problematic for model-based reasoning, since this interferes with the symbolic manipulation of equations that is crucial to simulation and diagnostics inference.

Bond graphs are an equation-oriented language which bear a close relation to MODELICA; in fact, a bond graph library exists in MODELICA [7]. Bond graphs constrain MODELICA to represent systems in terms of energy and power flows, thus forming a semantic framework for physical systems. Bond graphs model physical systems in terms of four entities: effort e, flow f, generalized momentum p, and generalized displacement q. A graph G enables the specification of energy flows among components, where a node corresponds to a component and an edge to a bond (i.e., a flow/effort interaction) between the joined components. The semantics of a bond graph G is specified through the assignment of differential- algebraic equations to each of the nodes and edges of the graph G, as based on mapping the graph structure (noting the connection semantics of the two basic connection types, parallel and series connection) into equations. These differential-algebraic equations describe the behavior of the four variables $p,\ q,\ e$ and f, for each of the physical components in the system (i.e., the nodes in the graph G).

Standard bond graphs have no notion of mode, or mode-based inference. Extended models, e.g., [21], have been developed, but the approach is quite different to that of LYDIA. For example, LYDIA does not impose any flow/energy restrictions to semantics, and its notion of mode is an inherent part of the language, rather than an extension.

RODELICA is a diagnosis language based on MODELICA, and used as the basis for the diagnostics system RODON [5, 4]. Rodelica is similar in structure to the LYDIA-NG language, in that it specifies component modes along with their associated behaviours. However, Rodelica is strictly more limited than the LYDIA-NG language, in that it allows not full ODSs but point- or interval-valued arithmetic constraints. In addition, a Rodelica model is restricted to atemporal equations (and hence uses data from one time instance), and cannot define the stochastic occurrence of faults in components. On top of this, the Rodon diagnostics system is limited to a single inference engine, as opposed to the ability of LYDIA-NG to use multiple inference engines and residual generators.

MATLAB/SIMULINK models (`http://mathworks.com/`) have a highly procedural semantics associated with simulation of a block-oriented model. Procedural tools for execution of a Simulink block during a given simulation step are governed by a number of factors; these include, among others, whether or not the block (or a subsystem containing the block) has a sample time, or whether or not the block resides in a conditionally executed subsystem. Block execution can also be disabled by *conditional input branch* statements. Matlab/Simulink has no inherent language framework for modes, nor well-established algorithms for diagnosis. As mentioned earlier, the LYDIA language is fully declarative and has associated algorithms for simulation and diagnostics inference.

2.2 LYDIA versus MODELICA

LYDIA is an equation-based language, i.e., a LYDIA model is translated to a system of equations. These equations can be Boolean, linear, or systems of ODEs. One of the major differences between LYDIA and MODELICA is the approach to solving systems of equations. MODELICA can solve DAEs. LYDIA tries to identify the type of the system of equations and invoke appropriate solver (simulation engine). For example, if LYDIA-NG detects a model that contains Boolean variables only, it will not use an ODE solver but a SAT algorithm which is better optimized for this class of systems. The idea is to use specialized solvers for various tasks, for examples trigonometric systems, etc.

LYDIA is in the same category of equation/simulation-based declarative languages but is targeted toward the diagnostic user-group. These are the the major differences between the two languages:

Syntax: LYDIA evolved from several diagnostic projects. The design of the language syntax was probably dictated by the experience of the language designers. LYDIA has syntactical resemblance to C, VERILOG, and ADA.

Type system: Both LYDIA and MODELICA are strongly typed languages. LYDIA optimizes heavily the use of Boolean variables.

Object orientation: MODELICA is an object-oriented language, while LYDIA is not. From all features of object-oriented languages [14], the most important one for equation-oriented approaches is inheritance which may lead to more compact models. LYDIA uses external pre-processors to achieve the same goals, however, in future extensions of the language the authors of LYDIA may bring inheritance into the language. Information hiding is supported by MODELICA and not supported by LYDIA. Information hiding may help modelers avoid mistakes.

Explicit procedures: LYDIA does not support MODELICA-type algorithm sections. The reason for that is that while MODELICA models are typically used from simulation only, LYDIA models are used for multiple simultaneous simulations. This imposes strict requirements on (1) the computational efficiency of the simulation and (2) the side-effects of the simulation. An example of a side-effect would be a MODELICA algorithm using a file on disk. In LYDIA this would create a problem as this file would be overwritten by the multiple simultaneous simulations for the various fault-modes. Further it is very difficult to have automated performance analysis on procedural code.

Units: LYDIA does not support directly units. Units can be specified as string attributes, but there is no unit algebra. Units may be supported as first-class citizens in future versions of LYDIA.

Modes: The most important difference between LYDIA and MODELICA is the use of modes in LYDIA. This is not strictly a language difference but an issue of interpreting the models. LYDIA detects health and user-input variables and identifies components based on these variables. This can be easily achieved in MODELICA by using special variable types,

e.g., according to a naming convention. LYDIA also identifies which equations belong to which component mode.

3. LYDIA Modeling Examples

This section describes the LYDIA language through examples, rather than use a formal approach. Viewed simply, LYDIA enables users to define models in terms of constraints, where constraints may range from logic to differential equations. Further, LYDIA supports component-based definitions of systems, such that for each component we can associate a *mode* that represents the distinct functional modes that drive the component's behaviors. When a system is composed, the system-level modes consist of the cross-product of the component modes, and the system equations consist of the union of the mode-based component equations. We use standard methods for component composition, e.g., [13].

A LYDIA model has four sections: prologue, domains, structure, and components. The prologue describes the main characteristics of the model, i.e., the types of the constraints, fault-modeling, etc. The structure displays the model hierarchy that is essential for many of the MBD algorithms existing today. The domain description specifies symbolic values for all the Finite Domain Integer (FDI) variables (Booleans are treated as a special case of many-valued logic). Finally, for each component a set of constraints and transitions are specified. In particular, LYDIA supports constraints ranging from logic to differential equations.

The models discussed in this section come from three different domains. The first one is an analogue electrical circuit. The example shown in section 3.1 is a logic circuit similar to the ones that are used for benchmarking of Automated Test Patter Generation (ATPG) [3]. Finally, the third example illustrates the use of Ordinary Differential Equations (ODEs) in LYDIA.

Another difference between LYDIA and simulation languages like SIMULINK and MODELICA is that LYDIA splits the model in two: system model and diagnostic scenario. The system model is similar to what we have in other modeling languages. The diagnostic scenario contains sensor data, initial values and other information that is unknown at modeling time. This allows the use of compilation methods—the system model does not change and can be compiled (sometimes in the strict sense of the term [8]) to facilitate reasoning. while the sensor data is supplied at run-time and even includes noise.

3.1 An Analogue Electrical Circuit

Figure 1 shows a small electrical circuit. It consists of a single voltage source (V_1), two switches (SW_1 and SW_2), and a current sensor (I_1). If we disregard the switches and the sensor (take the voltage source and resistors only) we can easily calculate all node voltages and branch currents. This can be done with an electronic calculator or with a simulation program like SPICE [16]. LYDIA-NG implements SPICE simulation.

When modeling a system in LYDIA we start with each component separately. With some luck, these are standard components and are already modeled in a component library. Otherwise the component models have to be created. We next show a resistor model in LYDIA. This is already part of the electrical component library `electrical.lcl`.

```
1  type ResistorHealth = enum { nominal, open, short };
2
3  system Resistor ( float resistance , current , pn, nn)
4  {
5      ResistorHealth h;
6
7      attribute health (h) = (h == ResistorHealth .nominal);
8
9      switch (h) {
10         ResistorHealth .nominal ->
11         {
12             resistor ( resistance , current , pn, nn);
13         }
14         ResistorHealth .open ->
15         {
16             // no constraint
17         }
18         ResistorHealth . short ->
19         {
20             resistor (0, current , pn, nn);
21         }
22     }
23 }
```

Listing 1. Resistor model

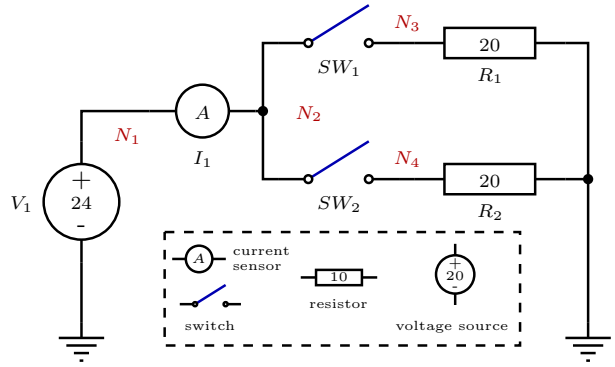

Figure 1. A small power distribution network

Line 1 defines a new discrete LYDIA type that will be used for the health of the resistor. This type defines three modes for the resistor: nominal, open-circuited, and short-circuited. The first mode is a nominal and the other two are fault-modes. The health variable itself is declared in line 5. We specify in the model which mode is the nominal and which modes are the fault modes by adding a variable attribute. This is done in line 7 of the resistor model.

Component models in LYDIA typically follow the same structure. LYDIA simulates for a set of nominal/fault modes. The choice which simulation goes for which fault mode is made in line 9 of the resistor model. When diagnosing, disambiguating, or otherwise reasoning, LYDIA-NG, will pick the relevant equations (constraints) depending on the hypothesized (assumed) value of the health variables (in the case of the resistor model above, the health variable is h).

The actual resistor equations (constraints) are specified in lines 12, 16, and 20. Notice that in the case of an open-circuit, there is no constraint, i.e., an open-circuited resistor is modeled with an empty set of constraints. This is equivalent to specifying a resistor with infinite resistance but eliminates the need of this resistor to be pruned (LYDIA-NG supports infinite resistance and conductance through symbolic preprocessing). For the nominal mode we specify the built-in constraint resistor , parametrized with its nominal resistance. LYDIA-NG will take this and fill-in the proper values in a nodal equation matrix so it can compute the unknown voltages and currents. In the case of a short-circuit

(line 20), we make the resistance parameter zero and LYDIA-NG knows how to deal with this case during simulation.

When diagnosing, LYDIA-NG will choose constraints based on hypothesized fault modes and construct a simulation model. This simulation model will be simulated with a domain-specific solver (in the above case with SPICE). This process will be repeated multiple times until the proper fault mode is identified.

We next describe a component that cannot be fully-simulated with SPICE . This is the current sensor. The current sensor consists of a small-resistor (just like the majority of the electronic current sensors do) and some equation that allows the reading of the sensor to be "stuck-at" some value in the presence of a fault. Of course, the last equation has nothing to do with SPICE and is a very simple algebraic equation. This algebraic equation can be solved by value propagation *after* the SPICE simulation finishes. LYDIA-NG partitions the constraints (equations) and invokes the appropriate solvers (up until now we have mentioned the SPICE solver and a simple algebraic propagation-based solver) automatically. Here is a model of a current sensor:

```
1  type SensorHealth = enum { nominal, failed };
2
3  system CurrentSensor( float pn, nn)
4  {
5      float r;
6
7      attribute observable(r);
8      attribute name(r) = "current␣[A]";
9
10     float current;
11
12     SensorHealth h;
13
14     attribute health(h) = (h == SensorHealth.nominal);
15
16     switch (h) {
17         SensorHealth.nominal ->
18         {
19             resistor (0.01, current, pn, nn);
20             r = current;
21         }
22         SensorHealth.failed ->
23         {
24             resistor (0.01, current, pn, nn);
25             r != current;
26         }
27     }
28 }
```

Listing 2. Current sensor model

Last, we show how to model a switch in LYDIA:

```
1  type SwitchHealth = enum { nominal, stuck };
2  type SwitchCommand = enum { open, closed };
3
4  system Switch( float current, pn, nn)
5  {
6      SwitchHealth h;
7      SwitchCommand cmd;
8
9      attribute health(h) = (h == SwitchHealth.nominal);
10     attribute control(cmd) =
11         (cmd == SwitchCommand.closed);
12
13     switch (cmd) {
14         SwitchCommand.open ->
15         {
16             switch (h) {
17                 SwitchHealth.nominal ->
18                 {
19                     // no constraint
20                 }
21                 SwitchHealth.stuck ->
22                 {
23                     resistor (0, current, pn, nn);
24                 }
25             }
26         }
27         SwitchCommand.closed ->
28         {
29             switch (h) {
30                 SwitchHealth.nominal ->
31                 {
32                     resistor (0, current, pn, nn);
33                 }
34                 SwitchHealth.stuck ->
35                 {
36                     // no constraint
37                 }
38             }
39         }
40     }
41 }
```

Listing 3. Single throw switch model

The new feature in the switch model is that we have a user-command variable—the commanded position of the switch. So, this is an example in which we have two parameters: the commanded position and the health. Remember that, when building component models we have to simulate for a subset of the Cartesian product of each user-command/health variable. In the above example we have two possible switch positions (open and close) and two possible modes (nominal and stuck), hence there is a total of four models for each combination of values of the parameters. Remember that the user-commands have to be specified in the outer switch statement and the health models in the inner switch statements. It is also possible to specify each simulation with a sequence of if-statements but using switches is more elegant.

Of course, there are also models of the voltage sensor and the voltage source but we will not discuss those. Fortunately, there are component libraries that come with LYDIA and the user is not required to model standard components. We have shown the three component models above only to explain some basic LYDIA modeling and to provide information for users to design their own component libraries for non-standard components.

Before we are ready to start simulation/diagnosis in LYDIA-NG we have to connect together all components that are shown in figure 1. This is done in the top-level (or main) LYDIA system:

```
1  #include "electrical.lcl"
2
3  attribute void reference;
4
5  system main()
6  {
7      float ground;
8
9      attribute reference(ground);
10
```

```
11      float N1, N2, N3, N4;
12      float V1, R1, R2, SW1, SW2;
13
14      voltage_source (24.0, V1, N1, ground);
15
16      system CurrentSensor I1(N1, N2);
17      system SimpleSwitch SW1(SW1, N2, N3);
18      system SimpleSwitch SW2(SW2, N2, N4);
19      system Resistor R1(20.0, R1, N3, ground);
20      system Resistor R2(20.0, R2, N4, ground);
21  }
```

Listing 4. Top-level system in a model of a power distribution network

The above top-level system starts with the include directive in line 1 so LYDIA can use the electrical component library. LYDIA-NG uses a C-like preprocessor. The first three models in this section were excerpts from the electrical component library.

Notice that the top-level system in a LYDIA model comes last (i.e., first all component and subsystem models and the last system is the top-level one). The rest of the systems and sub-systems do not have to be in a particular order as far as the top-level system is the last one.

The significant part of the top-level system instantiates and connects components from the component library (see lines 16–20). A system instantiation is done by specifying the keyword **system** followed by the type of the system and then the name of the instantiation. After the name of an instance follows (a left parenthesis and) a list of variables. The number and type of variables should match the system interface, otherwise the LYDIA compiler is going to produce an error.

3.2 A System of Boolean Equations

It is straightforward to enter Boolean equations in a LYDIA model. These systems of equations can be used for modeling of digital integrated circuits, or other combinatorial computational devices. Boolean functions are often represented graphically, by using the same symbols as in a standard computer arithmetic schoolbook [19]. An example Boolean function is shown in figure 2. This function implements a full-adder, a device that computes the sum (and the carry) of two Boolean numbers (and carry). By composing multiple of these one can build, for example, a 32-bit adder.

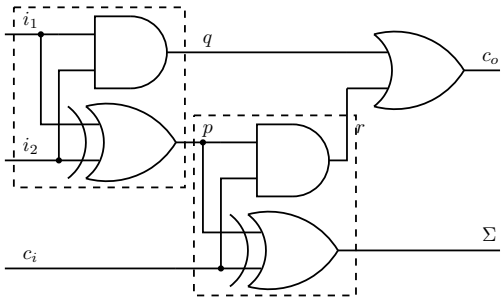

Figure 2. Boolean full adder

The full-adder shown in figure 2 has three types of components, in this case logic-gates: OR-gate, AND-gate, and XOR-gate. This is how a model of an AND-gate looks like:

```
1  system and2(bool o, i1, i2)
2  {
3      bool h;
4
5      attribute health(h) = h;
6
7      h => (o = (i1 and i2));
8      !h => (o = !(i1 and i2));
9  }
```

Listing 5. Model of an AND-gate with two inputs

Let us have a closer look at the above AND-gate model. First, all variables are Boolean. There are one output (o) and two input variables (i1 and i2). These three variables are declared as formals of the system (line 1). The health variable h is declared in line 3. The health attribute in line 5 tells LYDIA that when h equals true, the component is healthy, and otherwise. LYDIA-NG needs this so it can perform diagnosis. When diagnosing, LYDIA-NG performs multiple simulation for different values of the health variables and it needs to know which value means healthy (nominal) and which values means fault.

There are two constraints in the model of the AND-gate: one for when the gate is working nominally (line 7), and one for when the gate is at-fault (line 8). Instead of a **switch**-predicate like in the example in section 3.1, we use conditional expressions to differentiate between the nominal and the fault mode.

We call the model of the AND-gate shown in listing 5, a "stuck-at-opposite" model. This means that when the gate is faulty, the output of the gate has the opposite value of what it is supposed to be for the specified inputs. This is the same as a weak-fault model [10] but allows simulation by simple value propagation for any value of the health variable h.

A full-adder is composed of two half-adders as illustrated in figure 2. Each of the two half-adders in figure 2 is enclosed by a dashed rectangle. Modeling separately the half-adder and composing the full-adder out of the two half-adders and the OR-gate results in non-trivial hierarchy (i.e., a hierarchy where we do not only have component models and a top-level system, but also subsystems). The model of the half-adder simply combines and AND-gate and an XOR-gate as shown next.

```
1  system halfadder2(bool i1, i2, sum, carry)
2  {
3      system and2 A(carry, i1, i2);
4      system xor2 X(sum, i1, i2);
5  }
```

Listing 6. Model of a half-adder

Of course, a large number of Boolean gates, adders and various logic circuits are already modeled in the `std-logic-so.lcl` component library and can be used in any model that includes it.

To conclude the model of the full-adder, we have to compose the two half-adders and an OR-gate into the final top-level design. This is shown in the listing that comes next.

```
1  #include "std-logic-so.lcl"
2
3  attribute void input;
4  attribute void output;
5
6  system fulladder(bool ci, i1, i2, sum, carry)
7  {
8      attribute input(ci, i1, i2);
9      attribute output(sum, carry);
10     attribute observable(ci, i1, i2, sum, carry);
11
12     bool f, p, q;
13
14     system halfadder2 HA1(i1, i2, f, p);
```

```
15      system halfadder2 HA2(ci, f, sum, q);
16      system or2 O(carry, p, q);
17  }
```

Listing 7. Top-level system in a model of a Boolean adder

What is new in the top-level system above, is that in addition to all the subsystems and variables we have two new attributes—input (line 3) and output (line 4). We use these two attributes to denote the primary inputs (ci, i1, and i2), and the primary outputs (sum and carry). LYDIA-NG needs to know what is an input and what is an output so it can simulate, i.e., propagate the values of the Boolean inputs through the circuit to obtain the values of the Boolean outputs.

3.3 Ordinary Differential Equations

One of the first devices for measuring time is a water-filled vessel with a small orifice in it. Measuring time with such a device requires solving a differential equation as the rate of change of the observable quantity (the water height) depends on the amount of water in the vessel. The so called clepsydra problem is a standard problem in schoolbooks on ordinary differential equations [23, p. 108] and can be solved analytically.

In this paper we solve a diagnostic version of the water clock problem. We start with the clepsydra example in *Ordinary Differential Equations: An Elementary Textbook for Students in Mathematics, Engineering, and the Sciences* [22, p. 183] and modify it so it has two holes as illustrated in figure 3. Either of these two holes (or both) can get instantaneously and fully blocked. The goal is, given a measurement of the water height, to determine which of the holes is open and which is stuck. This is how the problem is formulated:

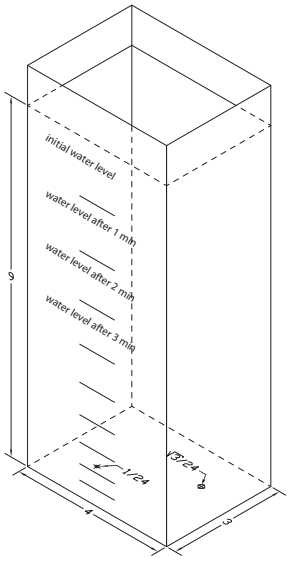

Figure 3. Water clock with two holes that can get blocked

Problem 1. Water flows freely throgh two orificies of a water vessel. It has been established that the rate of flow of water is proportional to the area of the orifices and the square root of the height of the water:

$$Adh = ka\sqrt{h}dt, \tag{1}$$

where

A is the area of the water surface,

h is the height of the water,

r_1 and r_2 are the radii of the orifices.

It has been determined experimentally that $k = -4.8$. In this problem, $A = 12$, $r_1 = \sqrt{3}/24$, $r_2 = 1/24$, and the initial level of the water is $h_0 = 9$.

In addition to all that, there is a discrete parameter $m \in \{0, 1, 2, 3\}$ where $m = 0$ means that both holes are unblocked, $m = 1$ means that the hole with radius r_1 is blocked, $m = 2$ means that the hole with radius r_2 is blocked, and $m = 3$ means that both holes are stuck. Given a measurement of the water α_t at time t and (predicted) levels of the water h_0, h_1, h_2, and h_3 for $m = 0, m = 1, m = 2$, and $m = 3$, respectively, we can compute the value of m from the formula:

$$m = \underset{i \in \{0,1,2,3\}}{\arg\min} |\alpha_t - h_i| \tag{2}$$

Given $h_t = 6.5$ at time $t = 135$, compute the value of the discrete parameter m.

Let us model in LYDIA the clepsydra shown in figure 3.

```
1   type ClepsydraHealth = enum { nominal, s1, s2, sb };
2
3   system Clepsydra()
4   {
5       float A = 12;
6       float r1 = ( sqrt (3) / 2) / 12;
7       float r2 = (1 / 2) / 12;
8
9       float height;
10
11      attribute observable(height);
12
13      ClepsydraHealth h;
14
15      attribute health(h) = (h == ClepsydraHealth.nominal);
16
17      switch (h) {
18          ClepsydraHealth.nominal ->
19          {
20              float area = pi * r1 ^ 2 + pi * r2 ^ 2;
21              height' = (-4.8 * area * sqrt(height)) / A;
22          }
23          ClepsydraHealth.s1 ->
24          {
25              float area = pi * r2 ^ 2;
26              height' = (-4.8 * area * sqrt(height)) / A;
27          }
28          ClepsydraHealth.s2 ->
29          {
30              float area = pi * r1 ^ 2;
31              height' = (-4.8 * area * sqrt(height)) / A;
32          }
33          ClepsydraHealth.sb ->
34          {
35              height' = 0;
36          }
37      }
38  }
```

Listing 8. Clepsydra model

The only new feature in the clepsydra model above is the use of derivatives in lines 21, 26, 31, and 35. In this case the dependent variable is height and the independent variable is (implicitly) t.

Notice that t (not used in the clepsydra model) is a keyword in LYDIA.

What remains is to specify the initial value (water height) and the measurement 10 min, after the beginning of the scenario. This is done in a scenario file similar to the one that follows.

```
1  scenario start  @ + 0      {}
2   initial values @ + 0      { height = 9; }
3  observation     @ + 135 s  { height = 6.5; }
4  scenario end    @ + 10 min {}
```

Listing 9. Clepsydra scenario

There are four events in the scenario above. The first and last ones in lines 1 and 4 mark the beginning and the end of the scenario (timestamps are in milliseconds). The initial value for the ODE is given in line 2. In line 3, the scenario supplies the measured water level.

If we now invoke LYDIA-NG to diagnose the above system with the above scenario, we should get that the clepsydra is most likely in state $m = 2$, i.e., the hole with radius r_2 is blocked.

4. LYDIA Concepts and Definitions

We represent a generic linear analogue system in terms of a relation between effort $\vec{x}$ and flow $\vec{z}$ vectors of variables, using $T\vec{x} = \vec{z}$, where T is an $n \times m$ matrix. For example, for circuits $\vec{x} \in \mathbb{R}^n$ is an (unknown) nodal voltage vector, and $\vec{z} \in \mathbb{R}^m$ is a measurable current-source vector.

We will adopt a mode-based representation, i.e., we assume that the system can operate in a set Ω of modes, which can consist of nominal or faulty operating conditions. Given a system that consists of a discrete set of components with a corresponding set of health parameters COMPS, a mode $\omega \in \Omega$ is an assignment to all variables in COMPS.

Further, for each mode we assume that we can specify a distinct set of equations. Hence, for each $\omega \in \Omega$ we specify an equation set SD_ω given by $T_\omega \vec{x_\omega} = \vec{z}_\omega$.

4.1 Models and Residuals

Definition 1 (Diagnostic Model)**.** Given a system that consists of a discrete set of components with a corresponding set of health parameters COMPS, a diagnostic model $M = \langle \mathrm{SD}, \mathrm{COMPS} \rangle$ is specified using a function $\mathrm{SD} = \bigcup_{\omega \in \Omega} \mathrm{SD}_\omega$.

In this article, we are typically given the flow vector $\vec{z}$ and must compute the effort vector via $\vec{x_\omega} = T_\omega^{-1} \vec{z}_\omega$, a process we call *simulation* of $\vec{x_\omega}$. Since SD is linear, we can simulate efficiently.

Given an observation $\vec{\alpha}$, we estimate the mode (i.e., solve a diagnostic problem) by computing an optimal solution of a parameter estimation problem where the parameters are discrete and the problem is split in two parts: simulation and residual analysis.

In real-world applications, straightforward simulation function (from definition 1) is not sufficient to adequately solve the diagnostic problem. This is because models are imprecise, there is sensor noise, health parameters are discrete, etc. Instead, we compute a difference between $\vec{\alpha}$ and a simulation $\hat{\vec{z}}$ (using the residual function of Definition 2), and then identify the mode that minimises this function.

Definition 2 (Residual Function)**.** Given two m-dimensional real vectors $\hat{\vec{z}}, \vec{\alpha} \in \mathbb{R}^m$, a residual function $R : \{\hat{\vec{z}}, \vec{\alpha}\} \mapsto R(\hat{\vec{z}}, \vec{\alpha})$ maps $\hat{\vec{z}}$ and $\vec{\alpha}$ into the real interval $[0; \infty)$.

Definition 3 (Health Estimation Problem)**.** Given a diagnostic model SD and a residual function R, the health estimation problem is to compute an assignment $\omega_{\min}$ to all variables in COMPS such that:

$$\omega_{\min} = \arg\min_{\omega} R(\mathrm{SD}_\omega, \vec{\alpha})$$

Solving the above health estimation problem, while maintaining computational efficiency is the main goal of our framework.

4.2 Control

LYDIA-NG supports control specifications in a straightforward manner. In particular, LYDIA-NG can specify mode-based controls (e.g., as occurs in Finite State Automata [6] and Hybrid Systems [15] models) using the notion of *mode* inherent in the LYDIA-NG language. This is possible since LYDIA-NG associates a set of dynamical equations with a mode (together with constraints on mode transitions), just as in hybrid systems. As such, a LYDIA-NG can specify a declarative control model and use the simulation infrastructure to run closed-loop feedback-control simulations.

Furthermore, the LYDIA-NG framework can support the diagnosis of hybrid systems, e.g., [1]. It is beyond the scope of this article to describe the control and diagnosis functionality of which LYDIA-NG is capable. It is important to note that LYDIA-NG currently has a simple temporal representation, and hence is limited to discrete-event models. In future work we intend to significantly extend the temporal modeling capabilities, and hence be able to analyse a wider range of hybrid models, e.g., as done in [18].

5. Framework for Model-Based Diagnosis

The basic idea of the LYDIA-NG diagnostic library (shown in Fig. 4) is to perform multiple simulations for various hypothesized health states of the plant. The output of these multiple simulations is then processed and combined into single diagnostic output.

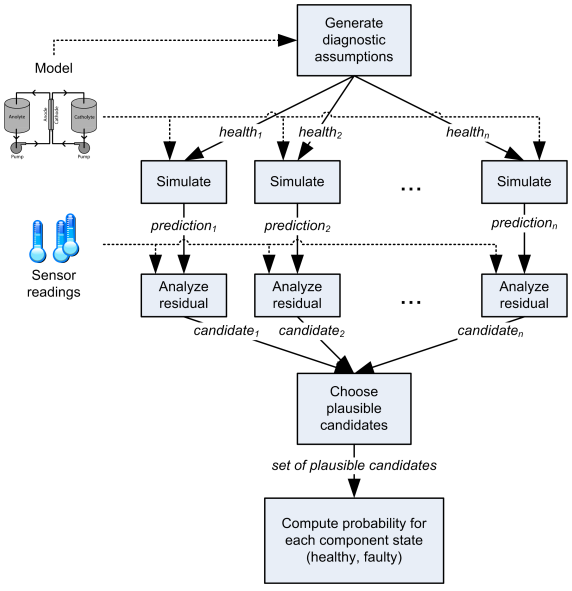

Figure 4. Overview of the LYDIA-NG diagnostic method

The LYDIA-NG diagnostic library consists of the following building blocks:

Generator of Diagnostic Assumptions: A diagnostic assumption is a set of hypothetical assignments for the health or fault state of each component in the system. The "all nominal"

diagnostic assumption assigns healthy status to each component. LYDIA-NG allows one nominal and one or more faulty states per component.

Simulation Engine: Given a diagnostic assumption, LYDIA-NG can construct a simulation model of the system. This simulation model consists of equations. By solving this system of equations LYDIA-NG computes values for one or more *observable* variables. The values of these observable variables is also referred to as a *prediction*.

Residual Analysis Engine: A prediction is compared to the sensor data by a residual analysis engine. This engine combines the individual discrepancies in each sensor data/predicted variable pair to produce a single real value that indicates how close is the prediction of the simulation engine to the sensor data obtained from the plant. A simulation that results in all predicted values coincide with the measured ones will result in the residual being zero. The data structure containing predictions, their corresponding sensor data and the computed residual is called a *diagnostic candidate* or simply *candidate*.

Candidate Selection Algorithm: Not all candidates generated by the residual analysis engine are used for computing the final system health. The candidate selection algorithm discards each candidate whose residual is larger than the residual of the "all nominal" candidate.

System State Estimation Algorithm: LYDIA-NG uses the set of candidates that is computed by the candidate selection algorithm to compute an estimate for the health of each component. This is done by the system state estimation algorithm. Finally, LYDIA-NG computes RCoF by choosing the components with highest probability of failure.

5.1 Search Algorithm

Algorithm 1 shows the top-level diagnostic process. The inputs to algorithm 1 are a model and a scenario, and the result is a diagnosis.

At the heart of algorithm 1 is the use of simulation. Algorithm 1 supports a large variety of simulation methods that may or may not use time as an independent variable. In the setup described in this paper we have used SPICE in combination with a constraint propagation solver. The latter we have used for sensor values, complex components such as mixed analog-digital electronics and other parts of the model where it is difficult or inappropriate to model with SPICE. The only requirement toward the simulation engine is to predict a number of variables whose types can be mapped to LYDIA-NG and to be relatively fast (the computational performance of LYDIA-NG will not be thoroughly discussed in this paper which emphasizes the application of LYDIA-NG to a space model).

The basic idea of algorithm 1 is to simulate for various health assignments and to compare the predictions with the observed sensor data (i.e., telemetry). There are several important aspects of this algorithms that ultimately affect the diagnostic accuracy as measured by various performance metrics.

The first algorithmic property that determines many of the diagnostic performances is the order in which health-assignments are generated. In algorithm 1 this is implemented by the function named NEXTHEALTHASSIGNMENT. The latter subroutine also determines when to stop the search and should be properly parametrized depending on the model and the user requirements. In the standard LYDIA-NG diagnostic library we provide the following diagnostic search policies:

Algorithm 1 Diagnosis framework

```
1: function DIAGNOSE(SCN) returns a diagnosis
       inputs: SCN, diagnostic scenario
       local variables: h, FDI vector, health assignment
                        p, real vector, prediction
                        Ω, a set of diagnostic candidates
                        DIAG, diagnosis, result
2:     while h ← NEXTHEALTHASSIGNMENT() do
3:         p ← SIMULATE(M, γ, h)
4:         r ← COMPUTERESIDUAL(p, α)
5:         Ω ← Ω ∪ ⟨h, r⟩
6:     end while
7:     DIAG ← COMBINECANDIDATES(Ω)
8:     return DIAG
9: end function
```

Breadth-First Search (BFS): This policy first generates the nominal health assignment, then single-faults, double-faults, etc.

Depth-First Search (DFS): This search policy starts with the nominal health assignment, then adds a single-fault, continues with a double fault including the first, and so on, until all components are failed. After the all-faulty assignment is generated, the algorithm backtracks one step and generates a sibling assignment and continues traversing down and backtracking in the same manner until no more backtracking is possible.

Backwards Greedy Stochastic Search (BGSS): In this mode, the search start from the all-faulty assignment. A random health variable is then flipped and the flip is retained iff the flip leads to a decrease in the residual. The order of health variables is arbitrary. As the whole search process is stochastic, it needs to be run multiple iterations in order to achieve the desired completeness. A formal description of this method for Boolean circuit models can be found in [11].

Each simulation produces what we call a *candidate*: a set of predicted values for a given health-assignment. The second important property of algorithm 1 is the comparison and ordering of the diagnostic candidates. This is done by mapping the predicted and observed variables into a single real-number, called *residual*. The residual computation is discussed in what follows.

5.2 Residual Generation

Our mode estimation task requires a method to identify the mode ω whose simulation z_ω most closely matches the observation vector α. We use a residual function $R(z_\omega, \alpha)$ to measure this difference. The specification of $R(z_\omega, \alpha)$ is intentionally generic, since we aim to enable users to specify domain-specific residual generators that are best suited to their application domain.

The area of residual analysis has received significant attention in the literature, and it is not possible to provide a comprehensive set of residual methods within LYDIA-NG.

We currently have implemented a small set of residual generators, the size of which will increase over time.

We provide below examples of two straightforward residual generation functions, together with their advantages and disadvantages. These two residual generation functions bear resemblance to loss functions in decision theory.

Squared Residuals:

$$R_{\text{sq}}(\text{OBS}, \vec{z}, \alpha) = \sum_{v \in \text{OBS}} W(v) \left[\vec{z}(v) - \alpha(v)\right]^2 \quad (3)$$

where $W(v)$ is a weight-value associated with sensor v, $\vec{z}(v)$ is the value of variable v in the prediction assignment $\vec{z}$ and $\alpha(v)$ is the value of the observable variable v.

Absolute Residuals:

$$R_{\text{abs}}(\text{OBS}, \vec{z}, \alpha) = \sum_{v \in \text{OBS}} W(v) \left|\vec{z}(v) - \alpha(v)\right| \quad (4)$$

where $W(v)$, $\vec{z}(v)$ and $\alpha(v)$ are used in the same way as in Eq. 3.

A disadvantage of the squared residuals function R_{sq} is that it adds a lot weight to outliers. In decision theory, the absolute loss function that corresponds to the R_{abs} function is discontinuous. The latter, however, is not a problem for the algorithms described in this paper and we prefer R_{abs} over R_{sq}.

The above two residual functions may lead to relatively bad diagnostic results, especially in the presence of noise. One of the properties of R_{sq} and R_{abs} is that they are memoryless. An alternative would be to use some historical predictions and sensor data. Of course, the use of history would increase the isolation time, but has the potential to also increase the diagnostic accuracy. Another approach for more advanced residual analysis function would be to use methods from machine learning, for example neural networks. Particle filters [9] or Bayesian networks [17] can be also used for residual analysis.

5.3 Computation of Component Failure Probabilities

Consider the circuit shown in Fig. 1 and a scenario $\alpha = \{I_1 = 1.19\}$. This scenario corresponds to one of the resistors being open-circuited or one of the switches being stuck-open. Table 1 shows applying Eq. 4 for the predictions simulated from the nominal and all single-fault health assignments. The rows of Table 1 are sorted in order of an increasing residual value. In this table (and below) we abbreviate a stuck switch as S and an open-circuit resistor mode as OC.

V_1	I_1	SW_1	SW_2	R_1	R_2	faults	R_{abs}
–	–	S	–	–	–	1	0.0006
–	–	–	S	–	–	1	0.0006
–	–	–	–	OC	–	1	0.0006
–	–	–	–	–	OC	1	0.0006
–	–	–	–	–	–	0	1.1758
F	–	–	–	–	–	1	1.1888
–	F	–	–	–	–	1	1.1888
–	–	–	–	SC	–	1	79.3402
–	–	–	–	–	SC	1	79.3402

Table 1. Single-fault residuals for the circuit shown in Fig. 1 and an observation simulated from a single open-circuited resistor

The COMBINECANDIDATES subroutine from algorithm 1 uses a table similar to the one shown in Table 1. It retains only the predictions with residuals smaller than the residual of the nominal prediction. The reason for that is that the nominal prediction is the only one that has a special meaning in LYDIA-NG and leads to a "landmark" residual, i.e., LYDIA-NG does not attempt to differentiate amongst the various fault-mode predictions. As a result, in our running example, only the first four rows of Table 1 are considered when calculating the final fault-probabilities.

The second step of COMBINECANDIDATES is to convert R_{abs} in the interval $[0; 1]$ where $R_{\text{norm}} = 0$ for the nominal prediction and $R_{\text{norm}} = 1$ for a fault prediction that gives $R_{\text{abs}} = 0$. Applying this on Table 1 gives us Table 2.

SW_1	SW_2	R_1	R_2	R_{norm}
S	–	–	–	1
–	S	–	–	1
–	–	OC	–	1
–	–	–	OC	1

Table 2. Normalized single-fault residuals from Table 1 that are smaller than the nominal residual

Finally, what remains to be done is to normalize the rightmost column of Table 2 so it sums up to one and marginalize the probability of failure in each column. For the small circuit we are analyzing this results in $\{\Pr(\text{SW}_1 = \text{S}) = 0.25, \Pr(\text{SW}_2 = \text{S}) = 0.25, \Pr(R_1 = \text{OC}) = 0.25, \Pr(R_2 = \text{OC}) = 0.25\}$. The fact that all probabilities are 0.25 means that algorithm 1 cannot determine unambiguously which component is the faulty one. In this case this is due to the fact that there is only one sensor, i.e., the unambiguity is due to sensor placement and circuit design.

One way to reduce this ambiguity is to change the position of SW_1 and/or SW_2. In the next section we devise an algorithmic framework that works for *any* circuit or model that can be diagnosed in the LYDIA-NG framework.

6. Conclusion and Future Work

This article has described a model-based framework for modeling and diagnostics of complex systems. The framework has several important characteristics, of which we have focused on the modeling language. This LYDIA language is a constraint-based system that enables modelers to specify systems according to a discrete set of system modes, such that each mode is associated with a behaviour. The language allows behaviour specifications based on a wide range of constraints, of which two important constraint representations are ODEs and first-order logic.

We have described several model types that can be developed in LYDIA. In addition, we have proposed a model benchmark for evaluating the capabilities of LYDIA and other MBD languages. By comparing our approach to that of well-known frameworks, such as MATLAB/SIMULINK and MODELICA, we have shown properties of the LYDIA framework that are specific to MBD.

Future work includes extending the range of simulation and diagnosis solvers within our framework, and extending the residual analysis engine.

Acknowledgments

The publishing of this article is supported by Enterprise Ireland grant CC-2011-4005A.

References

[1] Shai A Arogeti, Danwei Wang, and Chang Boon Low. Mode identification of hybrid systems in the presence of fault. *Industrial Electronics, IEEE Transactions on*, 57(4):1452–1467, 2010.

[2] Olof Bäck. *Modelling for diagnosis in Modelica: implementation and analysis*. PhD thesis, University of Linköping, 2008.

[3] Franc Brglez and Hideo Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. In *Proc. ISCAS'85*, pages 695–698, 1985.

[4] Peter Bunus, Olle Isaksson, Beate Frey, and Burkhard Münker. Model-based diagnostics techniques for avionics applications with rodon. In *2nd Workshop on Aviation System Technology*. Citeseer, 2009.

[5] Peter Bunus, Olle Isaksson, Beate Frey, and Burkhard Münker. Rodon-a model-based diagnosis approach for the dx diagnostic competition. *Proc. DX'09*, pages 423–430, 2009.

[6] Christos G Cassandras and Stephane Lafortune. *Introduction to discrete event systems*, volume 11. Kluwer academic publishers, 1999.

[7] François E Cellier and Àngela Nebot. The modelica bond graph library. In *4th International Modelica Conference*, 2005.

[8] Adnan Darwiche and Pierre Marquis. A knowlege compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[9] Nando de Freitas. Rao-blackwellised particle filtering for fault diagnosis. In *Proc. AEROCONF'02*, volume 4, pages 1767–1772, 2002.

[10] Johan de Kleer, Alan Mackworth, and Raymond Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56(2-3):197–222, 1992.

[11] Alexander Feldman, Gregory Provan, and Arjan van Gemund. Approximate model-based diagnosis using greedy stochastic search. *Journal of Artificial Intelligence Research*, 38:371–413, 2010.

[12] Peter Fritzson and Vadim Engelson. Modelica—a unified object-oriented language for system modeling and simulation. *ECOOP'98—Object-Oriented Programming*, pages 67–90, 1998.

[13] Gregor Gössler and Joseph Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1):161–183, 2005.

[14] I. Graham, A. O'Callaghan, and A.C. Wills. *Object-Oriented Methods: Principles & Practice*. Addison-Wesley Object Technology Series. Addison-Wesley, 2000.

[15] Thomas A Henzinger. The theory of hybrid automata. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 278–292. IEEE, 1996.

[16] Ron M. Kielkowski. *Inside SPICE*. Electronic packaging and interconnection series. McGraw-Hill, 1998.

[17] Uri Lerner, Ronald Parr, Daphne Koleer, and Gautam Biswas. Bayesian fault detection and diagnosis in dynamic systems. In *Proc. AAAI'00*, pages 531–537, 2000.

[18] Pieter J Mosterman and Gautam Biswas. A comprehensive methodology for building hybrid models of physical systems. *Artificial Intelligence*, 121(1):171–209, 2000.

[19] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Inc., New York, NY, USA, 2nd edition, 2009.

[20] Raymond Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1):57–95, 1987.

[21] AK Samantaray, K. Medjaher, B. Ould Bouamama, M. Staroswiecki, and G. Dauphin-Tanguy. Diagnostic bond graphs for online fault detection and isolation. *Simulation Modelling Practice and Theory*, 14(3):237–262, 2006.

[22] Morris Tenenbaum and Harry Pollard. *Ordinary Differential Equations: An Elementary Textbook for Students of Mathematics, Engineering, and the Sciences*. Dover Books on Mathematics. Dover Publications, 1963.

[23] D.G. Zill. *Differential Equations With Computer Lab Experiments*. Brooks/Cole, 1998.