Delft University of Technology
Parallel and Distributed Systems Report Series

The LYDIA Approach to Diagnostic Systems Modeling

A.J.C. van Gemund

Parallel and Distributed Systems Group
Faculty of Information Technology and Systems
Delft University of Technology
P.O.Box 5031, NL-2600 GA Delft
The Netherlands

report number PDS-2002-004

PDS

ISSN 1387-2109
Abstract

In this report we present the results of an initial study aimed at developing a simple modeling language for systems fault diagnosis and simulation. The modeling language allows for behavioral and structural systems specification, and is designed to be domain-independent. Thus, for instance Boolean and real-valued models of the same system need only differ in their (leaf) component behavioral description, while the structural system description can be reused. To allow experimentation, the language comes with a simple compiler, currently featuring a Boolean evaluation module that either infers system fault modes (diagnosis), or evaluates the values of ordinary system variables (simulation). A number of important issues, such as state and time are discussed, providing directions for future extension.
Contents

1 Introduction 2
2 Preliminaries 4
3 Principles 5
   3.1 Inference Technique .................................................. 6
   3.2 Compositionality ...................................................... 6
4 Language 8
   4.1 Initial Syntax .......................................................... 8
   4.2 Domain Independence ............................................... 10
5 Implementation 14
6 Extensions 17
   6.1 Compositionality ..................................................... 17
   6.2 Time ........................................................................ 18
   6.3 State ................................................................. 19
   6.4 Syntactic Issues ...................................................... 23
7 Examples 25
   7.1 Clock .................................................................... 25
   7.2 Pulse Width Monitor .................................................. 25
   7.3 Queuing System ....................................................... 25
   7.4 Automatic Door Controller ......................................... 26
   7.5 Bouncing Ball .......................................................... 27
   7.6 Paper Feed ............................................................ 27
8 Conclusion 30
Chapter 1

Introduction

Systems diagnosis is concerned with identifying specific failure modes of the system under study, based on a set of external observations, such as system input and output signals. In this report we present our first ideas on how to develop a systems modeling formalism that enables capturing the behavioral aspects necessary for fault diagnosis. Apart from diagnostic analysis, an explicit aim of our research is to develop a modeling formalism that also enables the analysis of related system properties that arise under system execution (through simulation), and system performability (through analysis). In most of this report, however, we will consider our primary application, i.e., fault diagnosis.

Systems analysis proceeds according to the following process (see Figure 1.1). In the first step, the system is *modeled* in terms of a systems *modeling formalism*. In the second step, the system model is *analyzed* with respect to a particular property, such as fault identification (i.e., fault diagnosis). Note, that the analysis process can also be simulation, where the underlying (simulation) algorithm produces execution statistics, useful for, e.g., performance evaluation. Given the above process, our research focuses on three aspects:

- Modeling formalism: the formalism should be expressive enough to capture essential aspects of system behavior, yet be abstract enough to allow analysis of the properties of interest.

- Systems Modeling: the modeling paradigm should be conducive to efficient mapping of system structure and behavior in terms of the formalism.

- Model Analysis: the analysis algorithms should provide an optimal trade-off between complexity and fault coverage.

In this report we propose a new modeling formalism, which we coin LYDIA (Language for sYstems DIAgnosis). We develop the language progressively, starting with the basic operators required to model...
functional system behavior. Next we introduce structural operators, allowing to specify systems in terms of reusable components. Next, we describe the tools currently developed. Finally, we address the more complicated issues of time and state, proposing a number of language approaches, illustrated by a number of case studies.
Chapter 2

Preliminaries

In this report we will use the Boolean operators and, or, and not in their usual meaning. For notation convenience, however, we will typically use the following syntactic constructs:

\[
\begin{align*}
\text{a }\ast\text{ b} & \iff\text{ a and b} \\
\text{a }+\text{ b} & \iff\text{ a or b} \\
!\text{ a} & \iff\text{ not a} \\
a =\text{ b} & \iff\text{ a }\ast\text{ b }+ (\text{! a} )\ast (\text{! b}) \\
a \neq\text{ b} & \iff\text{ ! (a = b)} \\
f\text{if (a) b} & \iff\text{ (! a) + b} \\
f\text{if (a) b else c} & \iff\text{ a }\ast\text{ b }+ (\text{! a} )\ast\text{ c}
\end{align*}
\]
Chapter 3

Principles

In this chapter we discuss the principles on which our approach is based. Consider a simple electrical system comprising a lamp and a push-button switch, as depicted in Figure 3.1. In the following we will adopt a simple binary value domain, i.e., voltages (e.g., \( z \)), mechanic pressures (\( x \)), and light emission (\( y \)) are either 0 or 1. The above system can be modeled in terms of the following Boolean predicates (% delimits comment):

\[
\begin{align*}
\text{if (}x\, (z = 1) \text{ else } (z = 0) & \text{ ) } \quad \% \text{ switch} \\
y = (z = 1) & \quad \% \text{ lamp}
\end{align*}
\]

It is easily verified that the above system reduces to

\[y = x\]

expressing the fact that iff the switch is pressed the lamp emits light.

Apart from nominal (correct) behavior, the above system can be easily extended to model fault behavior. Assume that the lamp has an associated health state, represented by the variable \( h \) which denotes whether the lamp functions correctly (\( h = 1 \)) or whether the lamp is broken (\( h = 0 \)). The system model becomes

\[
\begin{align*}
\text{if (}x\, (z = 1) \text{ else } (z = 0) & \text{ ) } \quad \% \text{ switch} \\
y = h \times (z = 1) & \quad \% \text{ lamp (h is health state)}
\end{align*}
\]

When \( x = 0 \) the system evaluates to \( y = 0 \), independent of the value of \( h \). However, when \( x = 1 \) the system evaluates to \( y = h \). This result can be interpreted (applied) in two ways:

- fault simulation: in normal simulation mode typically the health values are given, e.g., \( h = 0 \), and \( y \) is inferred to be \( y = 0 \) (no light).
• fault diagnosis: in fault diagnostic mode typically an observation such as \( y = 0 \) is given, and \( h \) is inferred to be \( h = 0 \) (faulty lamp).

Consequently, simulation and diagnosis are a complementary, but quite similar way of analyzing (evaluating) a system model. This motivates our aim of system models being amenable to both types of analysis. Apart from the elegance of capturing different applications within one paradigm, the ability of generating fault simulators and fault analyzers from the same model opens up the opportunity to evaluate diagnostic performance using fault simulation with guaranteed model correctness.

### 3.1 Inference Technique

In the above example, the inference technique has not been discussed explicitly. In the following we illustrate how a simple evaluator is capable of deriving the conclusions mentioned above.

To allow simple evaluation, the above system is converted to a canonical form comprising a system of Boolean equations using the definitions mentioned in Chapter 2:

\[
\begin{align*}
x \cdot z + (! x) \cdot (! z) &= 1 & \text{% switch} \\
y \cdot (h \cdot z) + (! y) \cdot (! (h \cdot z)) &= 1 & \text{% lamp}
\end{align*}
\]

The fact that each expression is equated to 1 is motivated by our desire to be compatible with the form that real-valued systems of equations are written (discussed later on), and is technically needed in order to conveniently perform evaluation. Assuming our observations \( x = 1 \), and \( y = 0 \), by substitution \((x, y)\) and propagation \((z)\) the entire system of equations evaluates to

\[
\begin{align*}
z &= 1 \\
(! h) &= 1
\end{align*}
\]

which implies \( h = 0 \). Note, that a simple evaluation scheme, that repeatedly evaluates those equations whose variables have been touched, yields the solution. Provided the above equations do not contain a recursion, the evaluation process is always guaranteed to terminate.

### 3.2 Compositionality

In the above system we have already accounted for the fact that there are two components, the switch and the lamp, both of which are represented by a separate predicate. In general, system specifications must be compositional such that the behavior can be automatically derived from a structural composition of individual components. To this end, we must generalize the above component expressions such that they do not carry context information regarding their role within some system. For instance, in the above system model, the switch equation already takes into account that it is connected to a voltage source with value 1 at its upper terminal, and that it directly controls \( z \) at its lower terminal. Thus we already decided that the component has an input and an output, based on the knowledge how the switch is connected to the voltage source and the lamp. In order to be specified as a component, the switch description should be entirely portable, such that the associated equation directly fits in with the rest of the system of equations.

In the following we develop a compositional approach to system and component specification, using the switch-lamp systems as an example. In order to allow for structural composition we first define a number of variables that represent the connections between the components. The system must now be decomposed in terms of the following components:

• switch
• lamp
• voltage source value 1
• voltage source value 0

as shown in Figure 3.2. The connection variables are a, b, and c.

Figure 3.2: Switch-lamp system (compositional view)

The behavioral description of the switch might be given by

\[
\text{if} \ (x) \ (a = b)
\]

which specifies that both switch terminals have the same value if the switch is closed. Note, that nothing is specified when the switch is open, it may still be that both terminals hold the same value (although in the case of the switch-lamp circuit both values will typically differ). In the same spirit, the behavioral description of the lamp would be given by

\[
y = h \ast (b \neq c)
\]

The switch-lamp system is then specified by

\[
\begin{align*}
a &= 1 & \text{% v source 1 at a} \\
íf \ (x) \ (a = b) & \text{% switch between a and b} \\
y &= h \ast (b \neq c) & \text{% lamp between b and c} \\
c &= 0 & \text{% v source 0 at c}
\end{align*}
\]

Based on the earlier observations \(x = 1\), and \(y = 0\), this specification yields the same result \((h = 0)\).

In the above, compositional approach, we can reuse the switch and lamp (sub)system specifications (typically defined in terms of a component library), and only need to specify how the components are connected, i.e., we only specify system \textit{structure}, system \textit{behavior} being inferred \textit{automatically}. A language, supporting behavioral and structural system specification is outlined in the next chapter.
Chapter 4

Language

In this chapter we develop a simple system specification language that supports behavioral and structural specification. The core idea of the language is that it should be as domain independent, and as simple as possible. While the form of behavioral specifications are essentially domain-specific, with respect to structural specification, it should not matter in which domain all the components have been defined.

4.1 Initial Syntax

Again, we take the switch-lamp system as running example. In order to separately specify components, at minimum we need a construct that delimits component specifications (system specifications in general), while allowing to pass variables needed to implement component interconnections. In the lamp-switch example, a, b, and c are variables that interconnect the switch, lamp, and voltage sources. The language construct used to specify systems is given by

```plaintext
system <name> [(<parameters>)] = {
    <specification>
}
```

where the keyword system is introduced on order to distinguish the definition from other definitions that might be needed in the future. The internal specification can either be behavioral (Boolean equations in the case of the switch-lamp system), or structural, as will be shown below.

The switch-lamp system may be specified as follows:

```plaintext
system switch(push,term1,term2) = {
    if (push) (term1 = term2)
}
system lamp(term1,term2,light) = {
    light = h * (term1 != term2)
}
system v_source(term1,value) = {
    term1 = value
}
system switch_lamp = {
    v_source(a,1)
    switch(x,a,b)
    lamp(b,c,y)
```
v_source(c, 0)
}

The specification comprises three behavioral component specifications and one structural specification that defines the switch-lamp systems in terms of the components specified earlier, using variables a, b, and c to connect (equate) the local component variables. In this specification approach, the structural specifications are simply macro-expanded to behavioral specifications. This macro substitution process can be seen by evaluating the switch-lamp specification,

```plaintext
system switch_lamp = {
    v_source(a, 1)
    switch(x, a, b)
    lamp(b, c, y)
    v_source(c, 0)
}
```

which evaluates to

```plaintext
system switch_lamp = {
    a = 1
    if (x) (a = b)
    y = h * (b != c)
    c = 0
}
```

As can be seen from the voltage source components, this approach also allows numeric values to be entered as parameter. For example, a specification without explicit voltage source components would also apply, i.e.,

```plaintext
system switch(push, term1, term2) = {
    if (push) (term1 = term2)
}
system lamp(term1, term2, light) = {
    light = h * (term1 != term2)
}
system switch_lamp = {
    switch(x, 1, b)
    lamp(b, 0, y)
}
```

Note that in the examples h is a global parameter. This implies that for multiple component instantiations the macro substitution scheme must apply a renaming scheme, based on, e.g., a global component indexing scheme. For example, consider a system where the switch component switches two lamps connected in parallel, i.e.,

```plaintext
system switch(push, term1, term2) = {
    if (push) (term1 = term2)
}
system lamp(term1, term2, light) = {
    light = h * (term1 != term2)
}
system v_source(term1, value) = {
```
term1 = value
}

system switch_lamp = {
    v_source(a, 1)
    switch(x, a, b)
    lamp(h, b, c, y1)
    lamp(h, b, c, y2)
    v_source(c, 0)
}

Let the components switch(x, 1, b), lamp(b, 0, y1), lamp(b, 0, y2) be assigned component indices 0, 1, 2, respectively. Then the health variable h is replaced by h_1, and h_2, respectively, resulting in

system switch_lamp = {
    a = 1
    if (x) (a = b)
    y1 = h_1 * (b != c)
    y2 = h_2 * (b != c)
    c = 0
}

Note, that if, for some reason, the same health variable was intended, this would be implemented by extending the component parameter list with a h parameter, connecting the parameters to a single, higher-level variable h, such as in

system switch(push, term1, term2) = {
    if (push) (term1 = term2)
}

system lamp(h, term1, term2, light) = {
    light = h * (term1 != term2)
}

system v_source(term1, value) = {
    term1 = value
}

system switch_lamp = {
    v_source(a, 1)
    switch(x, a, b)
    lamp(h, b, c, y1)
    lamp(h, b, c, y2)
    v_source(c, 0)
}

Of course, this “shared health” specification makes little sense as it would not allow both lamps to behave differently.

4.2 Domain Independence

As mentioned earlier, the substitution approach towards structural specification is based on the philosophy to keep away as much as possible from domain-specific modeling issues. An example that has motivated this approach is the desire to also be able to specify the switch-lamp system in terms
of a real-valued, voltage/current domain. Note that the initial motivation for this specification is simulation, rather than diagnosis, as the latter analysis essentially requires the domain to be Boolean (or finite, as in the Finite Domain approach [2]).

In the real-valued domain, the switch-lamp system is modeled in terms of a system of voltage and current equations, taking into account the fact that a switch acts as a mechanically controlled resistor, and the lamp acts as a passive resistor. Just like in the Boolean case, we start the discussion with the final behavioral system model, working our way back to see how the eventual component descriptions should look like. For each of the connections a, b, and c (see Figure 4.1) we specify the Kirchhoff

![Kirchhoff diagram]

Figure 4.1: Switch-lamp system (real-valued domain)

voltage and current equations (using \(x.v\) and \(x.i\) respectively) according to:

```plaintext
system switch_lamp = {
    a.v = 1                      % voltage at a
    a.v - b.v = a.i * (if (x) 1 else 10000000) % switch: Ohms law
    a.i = b.i                   % switch: current in=out
    b.v - c.v = 10 * b.i        % lamp: Ohms law
    b.i = c.i                   % lamp: current in=out
    y = if (h * c.i > 0.05) 1 else 0  % transfer current to y
    c.v = 0                     % voltage at c
}
```

The system is subsequently simulated ("evaluated", "solved") for \(x = 1\) and \(h = 1\), which yields (after substitution of \(x, a.v, \) and \(c.v\)):

```plaintext
system switch_lamp = {
    1 - b.v = 0.01 * a.i    % voltage at a
    a.i = b.i
    b.v = 10 * b.i
    b.i = c.i
    y = if (c.i > 0.05) 1 else 0
}
```

Similar to the Boolean case, the system is represented as a system of equations. Unlike the Boolean case, value substitution does not reduce the entire system to explicit form as the above system contains a system of linear equations, given by
system switch_lamp = {
  0.01 * a.i + b.v = 1
  a.i - b.i = 0
  b.v - 10 * b.i = 0
  b.i - c.i = 0
}

Ordered in terms of a.i, b.v, b.i, c.i, the associated system matrix is given by

\[
\begin{bmatrix}
0.01 & 1 & 0 & 0 & 1 \\
1 & 0 & -1 & 0 & 0 \\
0 & 1 & -10 & 0 & 0 \\
0 & 0 & 1 & -1 & 0
\end{bmatrix}
\]

where the last column is the b vector of the system \(Ax = b\). As the solution vector \(x\) is given by \((0.0999, 0.999, 0.0999, 0.0999)\), the system reduces to

system switch_lamp = {
  a.i = 0.0999
  b.v = 0.999
  b.i = 0.0999
  c.i = 0.0999
  y = 1
}

With respect to the structural language, the above goal specification suggests the following component descriptions:

system switch(push,term1,term2) = {
  term1.v - term2.v = term1.i * (if (push) 1 else 10000000)
  term1.i = term2.i
}

system lamp(term1,term2,light) = {
  term1.v - term2.v = 10 * term1.i
  term1.i = term2.i
  light = if (h * term2.i > 0.05) 1 else 0
}

system v_source(term1,value) = {
  term1.v = value
}

system switch_lamp = {
  v_source(a,1)
  switch(x,a,b)
  lamp(b,c,y)
  v_source(c,0)
}

In the underlying macro substitution mechanism it is tacitly assumed that signal members such as .v and .i are recognized (and maintained) in the substitution.

The important conclusion is that the structural specification of the global system is identical to the Boolean model, the behavioral aspects being hidden within the individual components. In the switch-lamp example, voltage and current can be handled (substituted) in the same way. In more complex
component networks, flow and effort variables must be treated differently given the two Kirchhoff laws that must be applied. This necessitates a syntactic extension of the above substitution technique which is deferred until a later time.

The philosophy to use a functional style of specification and substitution is based on the desire to keep things as simple as possible. This style has been inspired by our work in performance modeling that has led to the language PAMELA [4]. In PAMELA a model is a set of process, numeric, or resource equations, each equation of the form

\[
[\text{process}|\text{numeric}|\text{resource}] \ <\text{identifier}>[\langle\text{parameters}\rangle] = \\
<\text{expression}>
\]

In the LYDIA language, the keyword is \textit{system}, each system comprising a set of equations. As each equation involves time signals (as will be discussed later on), it can be argued that LYDIA comprises process equations as well as numeric equations.
Chapter 5

Implementation

In order to acquire some experience with this modeling approach we have implemented the above, prototype modeling language in terms of the following modules

- *lydparses* parses system specification source, producing an intermediate representation (IR, implemented in Tm)
- *lydsubs* recursively macro-substitutes system components in terms of their component bodies (symbolic eval)
- *lydeval* numerically solves (reduces) set of Boolean equations by repeated value substitution (numeric eval)
- *lydprint* unparses IR into source format for reading convenience

The basic specification compiler consists of the following pipeline:

```
lydparses | lydsubs | lydeval | lydprint
```

In order to enable interactive experimentation we also implemented a simple interpreter that allows editing of variables, each variable modification followed by an evaluation run. The interpreter is based on the pipeline

```
lydparses | lydsubs | lydinter
```

The *lydinter* module internally uses the evaluator *lydeval* and unparsers *lydprint* according to the following control loop:

```
lydinter:
  loop {
    result = eval(system)
    print(result)
    done = edit(system)
    if (done) exit loop
  }
```

As an illustration, we give an execution trace of the switch-lamp example, i.e.,

```
system switch(push,term1,term2) = {
  if (push) (term1 = term2)
}
```
system lamp(term1,term2,light) = {
    light = h * (term1 != term2)
}

system v_source(term1,value) = {
    term1 = value
}

system switch_lamp = {
    v_source(a,1)
    switch(x,a,b)
    lamp(b,c,y)
    v_source(c,0)
}

where we simulate correct system operation (x = 1, h = 1 which must produce y = 1). In the trace > is the edit prompt.

> x = 1

> h = 1

> y = 0

Next, we demonstrate the inverse, diagnostic application:
(y = 0),
(b = 1),
(h = 0)
]
> q
Chapter 6

Extensions

In this chapter we address a number of shortcomings of the current approach. First, we address the trade-off between compositionality and modeling power. Next, we develop language extensions to model the important concepts of time and state.

6.1 Compositionality

The important compositional requirement that the switch description be symmetric in terms of its terminals (a and b) has been met. This requirement is essential to compositionality as it is not known a priori how the component will be connected (assuming a non-directional world). This applies to both the Boolean and the real-valued specifications. While the real-valued specification is complete, the Boolean model

```plaintext
system switch(push,term1,term2) = {
    if (push) (term1 = term2)
}
```
as used thusfar suffers from an incompleteness that manifests itself in the case when $x = 0$. Let $h = 1$. The system evaluates to

$$(y * b) + ((! y) * (! b)) = 1$$

Consequently, the value of y cannot be deduced. This is logical since when $x = 0$ nothing can be specified about the values of term1 or term2 without compromising compositionality. A solution is to distinguish between voltage values and current values, a bit similar to the real-valued domain, such as

```plaintext
system switch(push,term1,term2) = {
    if (push) (term1.v = term2.v) else (term1.i = 0)
    term1.i = term2.i
}
```

For compatibility reasons, the lamp description now becomes

```plaintext
system lamp(term1,term2,light) = {
    term1.i = (term1 != term2)
    light = h * term1.i
    term1.i = term2.i
}
```
The above modifications yield a complete specification which solves the problem mentioned.

It would seem that, with proper specification, the modeling power of Boolean models is unlimited. However, there is a limit to what extent Boolean models can model systems, and be compositional at the same time. Consider a system comprising a series connection of two lamps, i.e.,

```plaintext
system series_lamps = {
    lamp(1,z,y1)
    lamp(z,0,y2)
}
```

In a digital-valued domain one must choose between the interconnect \( z \) being 0 or 1, as can be seen when assuming a simple lamp model such as

```plaintext
system lamp(term1,term2,light) = {
    light = (term1 != term2)
}
```

Evaluation of the model terminates at

\[
\begin{align*}
((y1 \cdot (! vb)) + ((! y1) \cdot vb)) &= 1, \\
((y2 \cdot vb) + ((! y2) \cdot (! vb))) &= 1
\end{align*}
\]

which implies that always \( y1 \neq y2 \) (only one of both lamps will burn).

This limitation of modeling power is overcome by a real-valued system (which allows unlimited numbers of networked components). This higher modeling power, however, comes at the price of Gaussian network analysis.

### 6.2 Time

Thusfar, our modeling language is purely functional and also lacks the notion of time. In this section we develop an extension how to model and reason about time. For the ease of the exposition we will consider a Boolean domain. Consider, the following simple system

```plaintext
system s(a,b,c) = {
    c = h \cdot a \cdot b
}
```

that computes the AND of \( a \) and \( b \) if the health state \( h = 1 \) and returns 0 otherwise. Given the observations \( a = 1, b = 1, \) and \( c = 0, \) it follows \( h = 0. \) Now consider a propagation delay such that \( c \) is delayed by \( t_p. \) This implies that if \( a \) becomes 1 at time \( t_a \) and \( b \) becomes 1 at time \( t_b, \) \( c \) should only become 1 at time \( \text{max}(t_a, t_b) + t_p. \) Consequently, it would be premature to evaluate that \( h = 0 \) immediately after observing that \( c = 0. \)

Let \( s(t) \) denote the variable (“signal”) \( s \) at time \( t. \) It holds

\[
c(t) = h(t) \cdot a(t - t_p) \cdot b(t - t_p)
\]

where \( h \) is defined as the current health state\(^1\). In order to infer \( h(t) \) we need to evaluate \( a(t - t_p) \) and \( b(t - t_p). \) Consequently, the evaluation scheme developed earlier can be applied, provided that an appropriate signal history is maintained. Assuming the availability of these (delayed) signals, evaluation of \( h \) (or \( c \)) can be applied continuously, as shown in Figure 6.1. In the figure it is shown that \( h(t) \) evaluates to 0 the moment \( c \) is observed to be 0 while \( c \) should have been 1. At a later time

\(^1\)One might argue that \( h \) should reflect the health state between \( t - t_p \) and \( t, \) but this is ignored.
instance, $c$ indeed becomes 1, at which moment $h$ is evaluated to be $1$. The earlier fault situation, however, has been properly inferred (and would be probably be recorded to initiate future system repair).

While the above approach essentially allows us to reason in time, an important issue is how this scheme is to be implemented. In principle, in our domain-independent approach, the history information applies to continuous signals. However, maintaining a possibly huge number of samples can be computationally prohibitive. For a Boolean domain (or any finite domain, for that matter), however, a considerable optimization can be applied due to the finite domain (only values 0 or 1) by adopting a Discrete Event approach. In this event-driven approach, signal samples are only stored at the events the signals change their value. Thus, in practice the FIFO buffer size per delayed signal will be limited, depending on the maximum delay window and the average event frequency per signal. Indeed, the evaluation itself, need only be applied at such instants (denoted by the vertical ticks on the time axis in the figure).

To extend our specification language with the notion of time we denote propagation delay in terms of an infix construct, according to

```plaintext
system s(a, b, c) = {
    c = h * ((a * b) after t_p)
}
```

The construct is taken from VHDL [1] which also models propagation delay. Note, that while in a simulation context it implies that $c(t + t_p) = h \cdot a(t) \cdot b(t)$, it also holds $c(t) = h \cdot a(t - t_p) \cdot b(t - t_p)$ as the equation applies continuously. In fact, the equation is internally parsed in terms of the latter form. This internal form is expressed as

$$c \text{ at time } = h * ((a * b) \text{ at time } - t_p)$$

where time denotes the actual (simulated) system time. The latter form is used when reasoning about time behavior.

### 6.3 State

Thusfar, our language is functional in the sense that the eventual system model (i.e., after all structural specifications have been substituted) comprises a system of equations. As most real-world system incorporate state, we need to develop an approach to model and reason about state.
As a motivating example, consider a simple Boolean set-reset latch (SR-latch), which is modeled according to

\[
\text{system } \text{srlatch}(s, r, q) = \{
q = (q + s) * (! r)
\}
\]

where \( s \) and \( r \) denote the “set” and “reset” input, respectively. Note the recursion which is characteristic for latching behavior. If \( s = 1 \) and \( r = 0 \) then the system latches to \( q = 1 \), while if \( s = 0 \) and \( r = 1 \) the system latches to \( q = 0 \). For \( s = 0 \) and \( r = 0 \) the system remains latched in (i.e., memorizes) its current state (represented by \( q \)).

In our current evaluation approach the recursion poses a fundamental problem since the \( q \) variables on the left-hand and the right-hand are not distinguished (a functional equation is symmetric). In fact, the equation is internally rewritten to the following canonical form (see Section 3.1):

\[
((q * ((q + s) * (! r))) + ((! q) * (! ((q + s) * (! r))))) = 1
\]

For \( s = 1 \) and \( r = 0 \) the system does evaluate to \( q = 1 \). However, for \( s = 0 \) (and \( r = 0 \)) the system reduces to

\[
q + (! q) = 1
\]

which admits both solutions for \( q \). Clearly, the above specification is incorrect, since it does not express the notion of “new” versus “old” variable values, and our according desire to distinguish between present and past, respectively.

Given our earlier extension to time, we can solve this specification problem by introducing propagation delay, which is, in fact, the primary mechanism responsible for latching behavior in real-world recursive systems. The corresponding specification is

\[
\text{system } \text{srlatch}(s, r, q) = \{
q = ((q + s) * (! r)) \text{ after } t_p
\}
\]

where \( t_p \) can be arbitrary small. Now the canonical equation becomes

\[
((q * ((q + s) * (! r)) \text{ at time-} t_p) +
((! q) * (! ((q + s) * (! r)) \text{ at time-} t_p))) = 1
\]

which produces the desired behavior, as can be seen as follows. Let \( q \) initially be \( u \) (undefined). For \( s = 1 \) and \( r = 0 \), after \( t_p \) it follows

\[
((q * (u + 1) * 1) + ((! q) * (! (u + 1) * 1))) = 1
\]

which reduces to

\[
q = 1
\]

After \( s \) is released to \( s = 0 \), it follows (assuming \( s \) has maintained its value for at least \( t_p \), so that \( q \) has become 1):

\[
((q * (1 + 0) * 1) + ((! q) * (! (1 + 0) * 1))) = 1
\]

which again reduces to

\[
q = 1
\]
The latching process is depicted in Figure 6.2.

As our aim is diagnostic analysis capability rather than simulation, we demonstrate how the latch is diagnosed. Consider the SR-latch

\[
\text{system sr_latch}(s, r, q) = \{
    q = ((h * q + s) * (! r)) \text{ after } t_p
\}
\]

in which we have modeled the hold circuit (that takes over the “set” operation once \( s \) releases) to have a health state \( h \). Let \( s, r, \) and \( q \) be measured as shown in Figure 6.3 where \( t_p \) is not visible (delay is not relevant for this example). When \( s \) is asserted the model evaluates to

\[
1 = (h * 0 + 1) * 1
\]

which does not offer conclusive information concerning \( h \) (i.e., there is no reason yet to suspect anything is wrong). After \( s \) is released, however, it follows

\[
0 = (h * 1 + 0) * 1
\]

which reveals that \( h = 0 \).

The elegance of the above solution using time is that state is introduced in terms of existing constructs, not having to extend the core language with a different abstract data type.

While conveniently close to the physics behind state behavior, the above approach may introduce problems when the state space of a variable is larger than 2 states. Consider an integer domain involving a counting process, such as
system counter(q) = {
    q = (q + 1) after 1
}

This works, but, as each equation, applies indefinitely (in a non-Boolean domain, q steadily increases). This touches on the fundamental issue of time control. In our functional regimen, each equation is valid for all time. State changing constructs, however, are to be executed at distinct moments in time.

There are two ways of controlling equation evaluation, using (1) level-triggered, and (2) edge-triggered mechanisms. A level-triggered mechanism only guards equations by a level condition. For instance, the above counter might be controlled by a variable c as in

```
system counter(c,q) = {
    q = (if (c) (q + 1) else q) after 1
}
```

Unfortunately, in this particular example, this level-triggered approach will still cause q to make as many increments as would fit in the (finite) time window of c. Consequently, we need a time-triggered, \textit{event} mechanism in order to specify at which event an equation needs to apply.

Given our above approach, it is only logical to proceed with the definition of an edge-triggered flip-flop (FF) in terms of the earlier latch specification, according to common digital design principles:

```
system sr_latch(s,r,q) = {
    q = ((q + s) * (! r)) after t_p
}
system gated_dLatch(c,d,q) = {
    sr_latch(c * d,c * (! d),q)
}
system ff(c,d,q) = {
    gated_latch(!c,d,p)
    gated_latch(c,p,q)
}
```

The above FF specification is positive edge-triggered, which implies that q is assigned the value of d at the rising edge of c.

While (event-triggered) assignment is commonplace in many specifications, the above FF component does not really provide a readable system specification. For instance, the assignment of \( x \) to \( y \) on the event \( z = 1 \) would have to be specified as

```
ff(z,x,y)
```

Rather than to use the FF component, we introduce the following syntactic sugar

```
on (z) y = x
```

which specifies that the equation is only valid at the event that \( z = 1 \) (positive edge-triggered by convention, \( ! z \) would mean negative edge-triggered).

Returning to the counter example, it is interesting to note that our approach applies to any domain, rather than just the Boolean domain. If we rewrite the gated latch (i.e., substitute the SR-latch and subsequently simplify the equation) to

```
system gated_dLatch(c,d,q) = {
    q = (if (c) d else q) after t_p
}
```

22
the variables \( d \) and \( q \) could also be integers or real-valued signals. Consequently, an event-driven counter can be specified as

\[
\text{system counter}(c, q) = \\
\quad \text{on (c) } q = (q + 1) \\
\]

Internally, this equation will be expanded to

\[
p = (\text{if} \ (\! c) \ (p + 1) \ \text{else} \ p) \ \text{after} \ t_p \\
q = (\text{if} \ (c) \ p \ \text{else} \ q) \ \text{after} \ t_p \\
\]

### 6.4 Syntactic Issues

In the current syntactic approach, the use of an `on` construct will always entail some predefined propagation delay \( t_p \) as a result of the internal FF implementation. As the propagation delay itself is immaterial in the desired latching behavior, it is appropriate to introduce an artificial zero delay (in VHDL called a “delta delay”), that allows distinguishing new variable versions versus old versions. Rather than using a syntax such as

\[
m = (\text{if} \ (c) \ p \ \text{else} \ q) \ \text{after} \ delta \\
q = q + 1 \\
\]

we simply omit the `after` construct, introducing the convention that the right-hand side (rhs) of a recursive equation is always evaluated in terms of the old (existing) variable version. Thus,

\[
q = q + 1 \\
\]

is internally evaluated as

\[
q = (q + 1) \text{ after} \ delta \\
\]

In this approach, the left-hand side (lhs) variables are interpreted as if they are defined. Indeed, many equations are stated in terms of a variable definition, containing only one lhs variable. In case of a conditional equation such as

\[
\text{if} \ (c = 1) \ (q = (q + 1)) \\
\]

the condition is treated as rhs. Consequently, the above equation is evaluated as

\[
\text{if} \ (c \ \text{after} \ delta = 1) \ (q = (q \ \text{after} \ delta + 1)) \\
\]

Note that the above, recursive equation implies latching, although the equation is formally not equal to the form originally introduced as a latch, i.e.,

\[
q = (\text{if} \ (c) \ (q + 1) \ \text{else} \ q) \\
\]

which equals

\[
q = \text{if} \ (c \ \text{after} \ delta) \ (q \ \text{after} \ delta + 1) \ \text{else} \ (q \ \text{after} \ delta) \\
\]

Yet both equations have the same effect\(^2\). In our discrete-event implementation, equations, although continuously valid, are only evaluated on the event one (or more) of their rhs variables are actually modified (in the example \( c \) or \( q \)). As the variable \( q \) already exists (it has already a value, which is to be used as rhs), the conditional equation

\(^2\)Although formally \( \text{if} \ (x) \ (y = a) \ \text{else} \ (y = b) \) equals \( y = \text{if} \ (x) \ a \ \text{else} \ b \).
if (c) (q = (q + 1))

need no else clause stating q = q. In fact, being recursively defined implies that for q an "old-new" history is kept. That is, internally a variable such as q is automatically treated as a state variable.

Returning to the level-triggered counter example, the original code

```plaintext
system counter(c,q) = {
    q = (if (c) (q + 1) else q) after 1
}
```

can now be written as

```plaintext
system counter(c,q) = {
    if (c) q = (q + 1) after 1
}
```

It is important to note, that conditional equations are always interpreted to be functional, unless there is an explicit recursion. For instance, the following equations

```plaintext
if (q = 0) (q = 1)
if (q = a) (q = b)
```

are not interpreted as latches, but simply reduce to

```plaintext
q = 1
(((! ((q * a) + ((! q) * (! a)))) + ((q * b) + ((! q) * (! b)))) = 1
```

That is, it may even be q itself that is evaluated as a result. If, on the other hand, the equations are meant to specify a state progression, this should be specified as

```plaintext
on (q = 0) (q = 1)
on (q = a) (q = b)
```

respectively.

Finally, note that the above, syntactic interpretation of a conditional equation, in fact, also applies to an event equation. For instance,

```plaintext
on (c) (q = (q + 1))
```

also implies that q is defined to its existing value on absence of an event on c. However, in contrast to a conditional equation, the equation

```plaintext
on (c) (a = (b + 1))
```

will define a to be a state variable, unlike the conditional case.
Chapter 7

Examples

In this chapter we discuss a number of case studies in order to evaluate our approach towards introducing state and time.

7.1 Clock

The following code models a simple clock:

```plaintext
system clock(t) = {
    t = (p) after t_c
}
```

7.2 Pulse Width Monitor

Consider a system that asserts y true if an input pulse x has minimum width t_w.

```plaintext
system pulse_monitor(x,y) = {
    on (x) t = time
    on (! x) y = (time - t > t_w)
}
```

In this example we introduce a built-in, real-valued variable time which returns system time.

7.3 Queuing System

Consider a queuing system comprising a customer generating process and a server process that serves a queue of customers. Let t_a denote the inter-arrival time, and let t_s denote the service-time (both can be declared as stochastic variable). The system can be modeled as

```plaintext
system customers(queued) = {
    queued = (queued + 1) after t_a
}

system server(queued) = {
    if (queued > 0) {
        queued = (queued - 1) after t_s
    }
}
```
system single_server = {
    customers(queued)
    server(queued)
}

As the arrival process is not synchronized with the (infinite) queue, this equation applies indefinitely. In contrast, the server process blocks on the queue, which is modeled by the if clause.

Again, note the fundamental difference between an if condition and an on condition. The if condition also causes the if clause equation to selectively apply. However, this clause does not apply only once per event (event-triggered), but is gated by the condition (level-triggered). The one-only event queued > 0 causes the expression to be evaluated, which, in turn triggers re-evaluation after time \( t_p \) as long as queued > 0.

### 7.4 Automatic Door Controller

Consider an automatic door that opens on the detected presence of a person, stays open for a while after detection, to be closed on timeout (assuming that no person has been detected within the timeout window). Let \( x \) denote the presence of a person. The controller has three states: (1) closed (! \( x \)), (2) open while \( x \), and (3) open while ! \( x \), waiting for timeout. Let \( y \) denote the door open signal. Let \( t_d \) denote the timeout duration. The (Moore-type) controller is specified according to

\[
\text{system controller}(x,y) = \{
\]

\[
\% Initialization
on (time = 0) {
    state = 1
    timeout = 0
}
\%
\]

\[
\% Finite state machine
on ((state = 1) * (x = 1)) state = 2
on ((state = 2) * (x = 0)) state = 3
on ((state = 3) * (x = 1)) state = 2
on ((state = 3) * (timeout = 1)) state = 1
\]

\[
\% Timeout control
on (state = 3) timeout = 1 after \( t_d \)
\%
\]

\[
\% Door control
\quad y = (state = 2) + (state = 3)
\}
\]

Unfortunately, the above specification has a flaw regarding the timeout mechanism. Let \( \text{state} = 3 \) pending a timeout. If yet another person were detected, the system returns to \( \text{state} = 2 \), after which a new timeout period should commence. However, at present there is no mechanism to cancel a pending event, caused by the earlier \( \text{timeout} = 1 \ after \ t_d \) equation.

We consider an alternative solution based on a separate timer with an input \( r \) that resets the timer, and an output \( \text{timeout} \) that becomes high at \( t_d \) after the last time \( r \) has become low. The controller specification remains the same, except for the timeout control statement which is replaced by
% Timeout control
r = ! (state = 3)

The timer specification is given by

    system timer(r,timeout) = {
        on (! r) target = time + t_d
        if (r) timeout = 0 else timeout = (time > target)
    }

Although the specification produces the desired behavior, another implementation problem surfaces. The second equation is evaluated *continuously* as a result of the continuously changing time signal (note, that, e.g., a VHDL implementation based on a counter suffers from the same problem as there must be a clock that continuously triggers the counting and comparison process).

In order to minimize polling in favor of interrupt-driven evaluation, it makes sense to introduce a cancellation mechanism as already hinted before. Under such a mechanism, consider the following specification

    timeout = 1 after t_d
    on (c) timeout = 0

The first equation schedules a timeout evaluation event at time + t_d. If in the mean-time c = 1, an immediate timeout evaluation event (timeout = 0) is scheduled, which cancels all pending timeout evaluations in the future. In terms of the door controller it follows

% Timeout control
on (state = 3) timeout = 1 after t_d
on (! (state = 3) timeout = 0

7.5 Bouncing Ball

The following example, taken from [3], models a bouncing ball. The example shows the use of *LYDIA* in expressing differential equations, combined with state change.

    system derivative(y,x) = {
        y = y + x * d_t after d_t
        % simple, explicit Euler implementation
    }
    system ball = {
        derivative(h,v) % h = height
        derivative(v,-g) % v = vertical speed, g = gravity
        on (h < 0) v = -c * v % c = elasticity constant
    }

7.6 Paper Feed

In this section we consider a paper feed subsystem which is part of a copier. When a control input c = 1 ("copy"), a relais pushes a rotor wheel on top of a paper sheet, which moves the sheet from the paper tray on to the copying area. The area has a sensor s, which is located at the end of the copying area. When the paper is correctly positioned, the sensor should return 1. The speed at which the paper is moved on to the copying area must be within a limited time period. Otherwise, this would signal a health problem. Let h_rotor denote the health of the paper movement hardware (between 0
and 1). Let \( v \) denote the vertical paper speed. The system is modeled by the following Boolean model (ignoring intermediate components)

\[
\text{system paper_feed}(x,s) = \{
    v = x \cdot h\_rotor
    s = v \text{ after } t
\}
\]

While the above model accounts for the nominal propagation delay between \( x \) and \( s \), the evaluator cannot infer that a timeout is caused by a low rotor speed.

To gain more understanding we consider a more detailed model. Let \( u \) denote the vertical paper displacement (state). At a more detailed level, the system is modeled by the following real-valued model:

\[
\text{system paper_feed}(x,s1,s2) = \{
    v = x \cdot h\_rotor \cdot \text{nominal\_speed}
    u = u + v \cdot \text{dt after } \text{dt}
    s = u > \text{paper\_length}
\}
\]

In this case, the propagation delay is implicitly computed as a dynamic function of \( h\_rotor \). However, this model cannot detect a timeout at all, nor is it amenable to inferring \( h\_rotor \) through measurement of \( u \) (the accumulation through state variable \( u \) prevents inferring \( v \) and thus \( h\_rotor \)).

A model that \textit{explicitly} accounts for propagation delay as a function of \( v \) is given by

\[
\text{system paper_feed}(x,s1,s2) = \{
    v = x \cdot h\_rotor \cdot \text{nominal\_speed}
    t = \text{paper\_length} / v
    s = x \text{ after } t
\}
\]

Still, the model does not permit evaluation of \( v \) given observation \( s \) as a function of time (i.e., reasoning backwards, given \( s \)).

We could attempt to instrument the model with time measurement logic as in

\[
\text{system paper_feed}(x,s1,s2) = \{
    v = x \cdot h\_rotor \cdot \text{nominal\_speed}
    t = \text{paper\_length} / v

    \% \text{measure } t
    \text{on } (x) \; t = \text{time}
    \text{on } (s) \; t = \text{time} - t
\}
\]

As \( t \) is observed, its value propagates via \( v \) to \( h\_rotor \). However, although this model \textit{is} geared towards \textit{diagnosis} (i.e., there exists a backward chain from \( s \) to \( v \)), it cannot be used in a \textit{simulation} context, as the \textit{forward} chain between \( v \) and \( s \) is now missing. Of course, we could include a forward chain as in

\[
\text{system paper_feed}(x,s1,s2) = \{
    v = x \cdot h\_rotor \cdot \text{nominal\_speed}
    t = \text{paper\_length} / v
\]
% predict s
s = x after t

% measure t
on (x) t = time
on (s) t = time - t
}

but this would make modeling less transparent, and would mean a breach from the original “bidirectional” concept that backward and forward reasoning should be accomodated through the same equation, rather than two separate ones.

Thusfar, we have introduced propagation delay as a constant, rather than as a variable. The above inference problems would be solved if we would extend our (bidirectional) reasoning to the propagation delay variable. Consider the following equation

\[ y = (x \times h) \text{ after } t \]

Let \( x = 1 \). If after propagation delay \( t \) no event on \( y \) was observed (\( y = 0 \)), it follows \( h = 0 \) as a result of the internally scheduled evaluation event after delay \( t \). However, at the event of an external observation of \( y \) at delay \( t' \), \( t \) will also be “observed”. Consequently, occurrences of \( t \) in other equations will be evaluated according to \( t = t' \). Returning to our example, this implies that

\[
\text{system paper_feed}(x,s1,s2) = \{
  v = x \times \text{h_rotor} \times \text{nominal_speed}
  t = \text{paper_length} / v
  s = x \text{ after } t
\}
\]

can be used for both simulation and diagnosis. In the diagnosis context, the inference proceeds according to the following process:

\[
\begin{align*}
\text{eval } t & \text{ from } s = x \text{ after } t \\
\text{eval } v & \text{ from } t = \text{paper_length} / v \\
\text{eval } \text{h_rotor} & \text{ from } v = x \times \text{h_rotor} \times \text{nominal_speed}
\end{align*}
\]
Chapter 8

Conclusion

In this report we present a number of initial modeling concepts aimed at the development of a systems diagnosis language called LYDIA. The basic concepts include domain-independence, and a functional approach to describing behavior in order to acquire the ability to use the same model for diagnosis and simulation. Apart from these basic concepts we propose an extension of the functional regimen to model time. Subsequently, we have used time to model state. We have illustrated our approach in terms of a few modeling case studies, which, in part, have led to suggesting a number of further extensions along the same lines. At first sight, our modeling approach seems to be appropriate in terms of striking a balance between modeling power and analytic tractability (bidirectional reasoning). Being an initial language approach, however, many more modeling case studies will be required before convergence to a final version will be reached. In particular, the planned development of a timed evaluator version will be extremely instrumental in experimentally validating a number of the concepts currently proposed.
Acknowledgements

Although most of this work has been performed at Delft University of Technology, many of the insights that have led to this report have been acquired in the course of my work at Science and Technology BV (“S & T”). In particular, the fruitful discussions with my S & T colleague Andre Bos are gratefully acknowledged.
Bibliography


