Delft University of Technology
Embedded Software Report Series

Lydia User Guide

Alexander Feldman a.b.feldman@tudelft.nl
Arjan van Gemund a.j.c.vangemund@tudelft.nl

Report Number: ES-2009-05
ISSN: 1877-7805

Published and produced by:
Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Information about the Embedded Software Report Series:
info-es-ewi@tudelft.nl
Information about the Embedded Software Section:
http://www.es.ewi.tudelft.nl/

Lydia User Guide

Alexander Feldman a.b.feldman@tudelft.nl
Delft University of Technology,
Faculty of Electrical Engineering, Mathematics and Computer Science,
Mekelweg 4, 2628 CD, Delft, The Netherlands

Arjan van Gemund a.j.c.vangemund@tudelft.nl
Delft University of Technology,
Faculty of Electrical Engineering, Mathematics and Computer Science,
Mekelweg 4, 2628 CD, Delft, The Netherlands

Abstract

In this report we present the syntax and semantics of the Lydia (Language for sYstem DIAgnosis) modeling language. Lydia is a low-level language for modeling of combinational systems. Some of its features are tailored to Model-Based Diagnosis (MBD). Lydia is also the name of the framework in which we have developed a large number of reasoning and diagnostic algorithms. The Lydia tool-kit also contains model translators (e.g., to CNF, DNF, OBDDs, etc), utilities, and reference implementation of algorithms such as CDA*, HA*, and others.

The initial design and implementation of Lydia can be found in van Gemund (2002, 2003). All the material in this thesis corresponds to version 2.0 of Lydia. This report presents an up-to-date overview of the Lydia language syntax and semantics.

Lydia models are collections of systems. Each system represents a component or a subsystem. Lydia systems are introduced in Sec. 1 where we also show the model of a small combinational circuit (a full-adder). In this early example we use variables and constraints intuitively. Section 2 explains the basic Lydia expressions. Lydia variables and data types are discussed in detail in Sec. 3. Lydia expressions are discussed in Sec. 4. Finally, Sec. 5 discusses Lydia predicates.

1. Systems and Subsystems

Synthetic and real-world systems have hierarchical structure which is represented in a Lydia model by using systems, subsystems and system instantiations. The structure of a Lydia model can be represented as multidigraph in which each system is a nodes and each instantiation is an edge. Lydia preserves this structural information until, explicitly "flattened-out" in the model translation pipeline as hierarchy exploitation has the potential of speeding-up the reasoning process.

A system declaration includes the keyword **system**, the system name and a list of formal parameters. System declarations conform to the following syntax:

system <*name*> (<*type*$_1$> <*formal*$_1$>, [<*type*$_2$>] <*formal*$_2$>, . . .)
{
⋮

```
}
```

Any valid Lydia identifier[1] can be used for the system name. A system may have zero, one or more formals. Similar to when declaring local variables, subsequent identical formal types may be omitted.

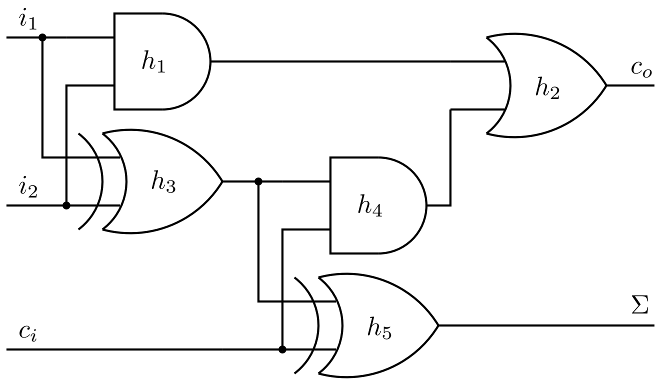

Figure 1: A full adder

We show the use of systems and subsystems and some basic Lydia constructs by modeling the Boolean full-adder shown in Fig. 1. A full-adder consists of five logic-gates: an or-gate, two and-gates, and two xor-gates. Each of these three component types will be modeled as a Lydia system. Note that Lydia does not have an explicit notion of a component. Lydia uses assumable (or health) variables to denote components. What follows are weak-fault[2] models of the three basic gate types.

```
system xor2(bool o, i1, i2)
{
    bool h;
    attribute probability(h) = (h ? 0.9999 : 0.0001);
    attribute health(h) = true;

    h => (o = (i1 != i2));
}

system and2(bool o, i1, i2)
{
    bool h;
    attribute probability(h) = (h ? 0.9999 : 0.0001);
    attribute health(h) = true;

    h => (o = (i1 and i2));
}

system or2(bool o, i1, i2)
{
```

1. Valid Lydia identifiers are sequences of alphanumeric characters or the underscore, starting with a letter.
2. For a discussion on fault-modeling strength, cf. Chapter **??**.

```
    bool h;
    attribute probability(h) = (h ? 0.9999 : 0.0001);
    attribute health(h) = true;

    h => (o = (i1 or i2));
}
```

Each of the above systems has three formal parameters[3]: an output o and two inputs (*i1* and *i2*). In each system there is a Boolean variable h that has the extra "health" and "probability" attributes. These two attributes make all h variables assumables. Finally, a single Lydia predicate models the nominal behavior of the gate. We will discuss variables and constraints in the sections that follow.

A full-adder is composed of two half-adder, each of which consists of an xor-gate and an and-gate. A model of a half-adder is shown below.

```
system halfadder2(bool i1, i2, sum, carry)
{
    system and2 A(carry, i1, i2);
    system xor2 X(sum, i1, i2);
}
```

Finally, two half-adders and an or-gate are connected (instantiated) in the final model:

```
system fulladder2(bool i1, i2, ci, sum, carry)
{
        bool f, p, q;

        system halfadder2 HA1, HA2;

        HA1(i1, i2, f, p);
        HA2(ci, f, sum, q);

        system or2 O(carry, p, q);
}
```

2. Basic Expressions

A Lydia expression is a formula over the standard propositional operators $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication), and $\Leftrightarrow$ (equivalence). Negation is typed as "!" or, alternatively, as the keyword "**not**". Conjunction can be typed as "&&" or textually as "**and**" and disjunction is denoted both by the symbolic "||" and mnemonic "**or**". Implication is denoted as "=>" and equivalence either as "=" or "==".

To illustrate the use of the above five operators we will use simulation to compute the value of the Boolean function

$$f(x, y, z) \equiv x \wedge \neg y \vee (z \Rightarrow y) \Leftrightarrow z \tag{1}$$

3. Referred to as ports in HDL languages such as Verilog$^{\text{TM}}$.

In our example we will evaluate $f(1, 1, 0)$. The function is implemented in the following four-variable model:

```
system expr(bool f, x, y, z)
{
    attribute observable(x, y, z) = true;

    f == (x && !y || (z => y) = z);
}
```

The value $(1, 1, 0)$ at which we want to see f evaluated is specified as the following observation:

```
observation alpha_1
{
    x && y && !z;
}
```

Finally, we invoke Lydia with the simulation option. The result is shown below.

```
lydia> sim expr alpha_1
x = true, y = true, f = false, z = false
```

3. Data Types

A Lydia model is variable-centric, that is, it specifies the properties of a system as a set of constraints over one or more variables. In the reasoning process, these variables are always treated separately (i.e., early in the translation process arrays and structures are broken down into their elements). The Lydia language allows rich variable-type semantics, resulting in short and expressive models. In what follows we discuss the built-in Lydia variable data-types and the primitives for defining user-types.

3.1 Atomic Data Types

Lydia has two atomic variable types: Booleans and enumerations. The former exists for convenience and for more aggressive optimization in the tool-kit (if the Boolean type was not built-in, a user could have easily introduced an enumeration defining it). A Lydia modeler needs to declare all variables before using them. A variable declaration can be done either in the list of formal parameters of a system or in the system body.

```
system Bar(bool v_1, v_2, bool v_3, _ldots, v_n)
{
        ⋮
}
```

Good modeling style avoids unnecessary variables in the formal parameter list of a system – sharing variables makes models difficult to read and hinders the performance of some hierarchical diagnostic reasoners (Feldman & van Gemund, 2006).

The order of Lydia statements is in most cases irrelevant for the model. Lydia even does not make mandatory to declare a type or a variable before using it (declarations are mandatory, no matter if a declaration is before or after a corresponding defintion). An enumeration, for example, can be defined anywhere in the global scope of a model. An enumeration specifies a set of symbols that are allowed for a variable of a given type. Consider an example where a variable of type *color* can be either red or green or blue:

```
type color = enum { red, green, blue };
```

Once we have declared the *color* type, it can be used anywhere in the model as illustrated in the following model excerpt:

```
system ColorMixer(color in1, in2, out1, out2)
{
        color common;
                ⋮
}
```

Constant types should be always explicitly specified as in the right hand side of the Lydia expression shown below. Having the *color* type declaration from above, we can assert the value of the internal variable *common* in the following manner:

```
        common = color.red;
```

Note, that due to the semantics of the enumerations, a variable of type *color* can be either red or green or blue but it can assume exactly one value. Hence, any system containing an expression like:

```
        (common = color.red) && (common = color.blue);
```

is trivially inconsistent, while the following constraint simplifies to true:

```
        (common = color.red) || (common = color.green) || (common = color.blue);
```

3.2 Composite Data Types

Lydia modelers can use arrays and structures to group related variables. In addition to that, the Lydia language allows type aliasing.

3.2.1 Arrays

A Lydia model can use arrays of any type and dimension. These arrays are "broken down" into sets of variables at compile[4] time, hence only statically-defined arrays are allowed. Unlike other languages, arrays may start from any index. We will illustrate the use of arrays by solving the well-known N-Queens problem. The model, shown below, defines the chess-board as a two-dimensional array of type **bool**. Placing a queen on the chess board will be expressed as assigning **True** to the respective member of the array. Below is the full N-Queens Lydia model:

4. We mean non-strict compilation, that is a translation to some normal form, e.g., a Conjunctive Normal Form (CNF).

```
const int N = 6;

system nqueens()
{
    bool board[1:N][1:N];

    forall (i in 1 .. N) {
        exists (j in 1 .. N) { /* At Least One */
            board[i][j];
        }
        forall (k in 1 .. N − 1) { /* At Most One */
            forall (l in k + 1 .. N) {
                !board[i][k] || !board[i][l];
                !board[k][i] || !board[l][i];
            }
        }
    }

    forall (i in 1 .. N − 1) { /* At Most One */
        forall (j in 1 .. N − i) {
            forall (k in 1 .. N − i − j + 1) {
                !board[i + j − 1][j] || !board[i + j + k − 1][j + k];
                !board[j][i + j − 1] || !board[j + k][i + j + k − 1];
                !board[N − i − j + 2][j] || !board[N − i − j − k + 2][j + k];
                !board[N − j + 1][i + j − 1] ||
                !board[N − j − k + 1][i + j + k − 1];
            }
        }
    }
}
```

To solve the N-Queens problem we have to impose a number of constraints. To do this we use quantifiers (cf. Sec. 5.3) that specify that there is at least one queen in every row (or column) and at most one queen per row, column and diagonal of the board. Note, that for convenience, we started our arrays from one.

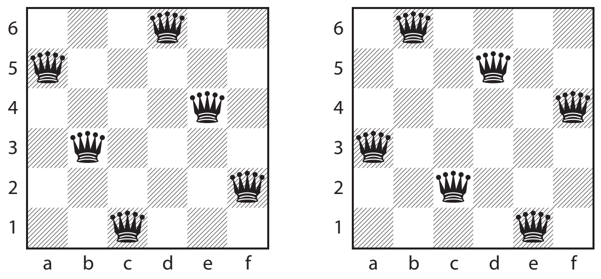

Figure 2: Two of the four solutions of the 6-queens problem

An array declaration, like a variable declaration, can appear anywhere in a system and should adhere to the following syntax:

$<\textit{type}>\ <\textit{name}>\ [s_1 : e_1][s_2 : e_2] \ldots [s_n : e_n];$

In the syntax definition above, <type> is any atomic or composite type, <name> is a valid Lydia identifier and s_i, e_i $(i = 1, 2, \ldots, n)$ are integers specifying the first and last elements of the array[5]. Alternatively, one can define arrays in a C-like manner, specifying only the size of each dimension.

$<\textit{type}>\ <\textit{name}>\ [d_1][d_2] \ldots [d_n];$

In the excerpt below, the declarations of the cubeX and cubeY arrays are of equivalent type.

```
bool cubeX[0:7][0:7][0:7];
bool cubeY[8][8][8];
```

3.2.2 Structures

Variables can be grouped in structures. The latter are simply naming conveniences, as variables in structures share common suffixes (the names of the structures). The structures are expanded by Lydia into separate variables at compile time. Consider several of the type definitions from the X-34 domain shown next.

```
type range = enum { below, inside, above };
type fluid = enum { helium, LO2, RP1 };

type line = struct
{
    range pressure,
    range temperature,
    fluid contents
};
```

The third type definition combines the *pressure*, *temperature* and *contents* variables in a structure defining a pipe. Lydia imposes no restrictions on the types of the structure members. In the example above, the three member variables are all of enumeration types, while a structure member can be also a (nested) structure, or even an array (cf. Sec. 3.2.1). This allows for a fairly complex composite Lydia data types.

5. It is not necessary that $s_i < e_i$.

Continuing our *line* structure example, we declare *line* instances and use these instances for defining system constraints. This is illustrated in the following model fragment:

```
system PneumaticValveAndMicroSwitch(line i, o, c)
{
    valveState state;
    microSwitchState microSwitch;

                    ⋮

    if ((state = valveState.open) or (state = valveState.stuckOpen)) {
        o = i;
    }                                                                        10
    if ((state = valveState.closed) or (state = valveState.stuckClosed)) {
        o.pressure = range.below;
    }
}
```

The above excerpt (from a model of a pneumatic valve) defines three formal variables of the structure type *line*. These are named *i*, *o* and *c*. The consequent of the first **if** construct states that all member variables of *i* should be equal to all member variables of *o*. Hence, o = i is a shortcut for the following set of constraints:

```
    i.pressure = o.pressure;
    i.temperature = o.temperature;
    i.contents = o.contents;
```

3.2.3 Type Aliases

Type aliasing is part of the Lydia typing system. Here is an example of how we can define a type *bar*, introduce a type *foo* that is an alias of *bar* and use the new *foo* type for instantiating a variable:

```
type bar = enum { red, green, blue };
type foo = bar;

system main()
{
    foo color = bar.green;
}
```

3.3 Variable Attributes

A Lydia variable may carry attributes. Attributes have no universal semantics in the Lydia language. Attributes can be used to carry application specific information for the end user, or, for applications using Lydia.

An attribute assigns data to each value a variable may assume. This is best illustrated with an example. Suppose we have a variable *k* of type *color* (cf. the type definition of *color* in Sec. 3.1). The variable *k*, can assume the values *red*, *green*, or *blue*. We can declare an attribute **rgb** of type **int** and assign a numeric value for each of the *k* values (*red*, *green*, or *blue*). After the diagnosis (or simulation) process computes values for k, the respective numerical attributes can be retrieved by the application that uses Lydia.

There are three special built-in variable attribute types in Lydia that do not have to be declared: *probability*, *health*, and *observable*. We have already seen these three attributes in the preceding examples and will discuss them in more detail below. These three attributes are used by all diagnostic solvers.

Lydia attribute are typed. Attributes can be of type **bool**, **int**, **float**, and **string**. The variable attributing mechanism is limited in expressiveness, and does not allow user-defined types. A variable attribute declaration has the following syntax:

attribute <*type*> <*name*>

A Lydia attribute assigns a value for each possible value a variable may assume. Any expression for which *all* values of a variable can be *precomputed* at model compilation time can be used as a variable attribute. Note that all except the three built-in attribute types should be declared at the global scope. Variable attributes are defined in the same scope where the variables are defined (the order is irrelevant). Both formal and local variables can have attributes. What follows is the syntax of a variable attribute declaration.

attribute <*name*> (<$variable_1$>, <$variable_2$>, …, <$variable_n$>) = <*expression*>

The simplest variable attributes assign constant values to variables.

We next discuss the three built-in attribute types. We have already mentioned that the **observable** attribute has a special meaning in Lydia. Variables, which have this attribute specified, may get their values from the user, i.e., they can be *observed*.

Consider a model of an inverter. It has its input and output variables i and o marked as observables and starts as follows:

```
system inverter(bool i, o)
{
    attribute observable(i, o) = true;
              ⋮
}
```

We use h for the health of the above inverter. To mark a variable as *health* or *assumable* (the latter two are synonyms) we have to assign a pair of built-in attributes: **probability** and **health**. The former is of type **float**, while the latter is of type **bool**. We continue the inverter model with assigning those two attributes to h:

```
              ⋮
    attribute probability(h) = (h ? 0.95 : 0.05);
    attribute health(h) = h;
```

The first attribute specifies that h is 0.95 for h = **true**, and 0.05 for h = **false**. Note that all a priori probabilities must sum to unity (otherwise the Lydia model compiler will produce a warning message). The health attribute specifies when a component is “healthy”. In the model continuation below the inverter will be healthy when h = **true**, otherwise, for h = **false**, it is broken (or faulty).

As an alternative to the above two attributes we can use f = **false** to denote a faulty gate (thus inverting the health logic) in the following way:

```
        ⋮
attribute probability(f) = (f ? 0.01 : 0.99);
attribute health(f) = !f;
```

The two lines above specify that the prior fault probability of f = **false** is 0.99 and that of f = **true** is 0.01. The meaning of the health attribute is inverted in comparison to *h*, i.e., *f* is faulty for f = **false**.

Having discussed the three built-in attributes, we continue our discussion with an example on the use of user-defined attributes. The model excerpt below declares three user type attributes. These can be used in a model, where, for example, we would like to attach user-supplied data to some health variables. After the computation of diagnosis, this data can be used for locating the faulty component or computing cost of repair, etc.

```
  ⋮
attribute string location;
attribute float cost;
attribute bool input;
  ⋮
```

The part from the Lydia system description, shown next, uses the three newly defined attribute types (*location*, *cost*, and *input*) to attach user-supplied values to the variable *h*.

```
  ⋮
attribute cost(h) = h ? 97.5 : 2.5;
attribute input(h) = true;
attribute location(h) = "quadrant 12";
  ⋮
```

In the above case the *input* and *location* attributes are constant for every value of *h*.

As we have seen from the **observable** attribute, it is possible to attribute multiple variables with a single statement. In our example, however, the attribute had a constant value. The probability, for example, is different depending on the value of *h*, hence the expression h ? 0.95 : 0.05. The above example would not work if we try to attribute multiple variables at the same time due to the fact that *h* is mentioned in the right-hand side of the attribute expression. For example, trying to compile:

```
system main()
{
    bool h1, h2;

    attribute probability(h1, h2) = h1 ? 0.95 : 0.05;
}
```

would lead to the following error:

```
err.sys:5: error: can't evaluate probability(h2) for h2 = false
err.sys:5: error: (Each non-evaluated value is reported only once
err.sys:5: error: for each attribute it appears in.)
```

As the alternative of attributing each variable separately is cumbersome, Lydia allows the use of attribute aliases in the attribute expression as illustrated next:

```
    attribute probability(h1, h2) = \x x ? 0.95 : 0.05;
```

In this case x will alias first *h1* and then *h2* and both *h1* and *h2* will receive the same a priori probability. Only variable of the same type can be attributed in this way. The same technique may be used for attributing variable arrays, as is shown in the example below:

```
    bool g[4][4];

    attribute probability(g) = \x x ? 0.99 : 0.01;
    attribute health(g) = true;
```

When attributing arrays, it is also possible to attribute all the elements of the arrays separately, or in groups:

```
     bool z[3];

     attribute observable z[0:1] = true;
     attribute health z[2] = true;
```

Variables grouped in structures (cf. Sec. 3.2.2) can be attributed by using attribute aliases if all variables are of the same type. The same applies for complex data types consisting of structures and arrays. What follows is an example of attributing all the variables in a structure:

```
type flow = struct { bool fsign, bool fder };

attribute float d;

system main()
{
    flow f;

    attribute d(f) = \x x ? 10 : 100;
}
```
10

The above mechanism would not work if the variables in a structure are of different types. In this case we can use typed aliases as illustrated next:

```
type content = enum { t1, t2, t3 };
type flow = struct { bool fsign, bool fder, content c1, content c2 };

attribute float d;

system main()
{
    flow f;

    attribute d(f) = \x::bool x ? 10 : 100;
    attribute d(f) = \x::content cond(x) (content.t1 -> 1; content.t2 -> 2)
}
```

As we have seen, variable attributes provide a powerful macro-mechanism for attaching user-defined data to variables. In the examples we have used some Lydia expressions to compute values for each value a variable may assume. Note, that the aliasing mechanism which we have seen and the **int**, **float**, and **string** attribute types present in the attribute processing mechanism only and they can not be used outside of attribute expressions.

4. Expressions

Lydia expressions are used for specifying variable constraints.

4.1 Conditional Expressions

The syntax of the arithmetic if in Lydia is similar to the one in C:

$<\textit{expression}_1>$? $<\textit{expression}_2>$: $<\textit{expression}_3>$

Arithmetic if expressions can be nested. Each arithmetic if is internally translated to $(\textit{expression}_1 \Rightarrow \textit{expression}_2) \wedge (\neg\textit{expression}_1 \Rightarrow \textit{constraints}_3)$. The conditional if in the inverter model below stipulates that inverter should be healthy if the value of its input i is equivalent to the value of its output o.

```
system inverter(bool h, i, o)
{
    attribute observable(i, o) = true;
    attribute health(h) = h;
    attribute probability(h) = (h ? 0.95 : 0.05);

    h ? (i == o) : (i != o);
}
```

An expression without equivalent in other languages is the conditional switch. A conditional switch expression has the following syntax:

cond ($<\textit{expression}_1>$) (

```
    <expression_2> –> <expression_3>
                  ⋮
    <expression_2n> –> <expression_2n+1>
    default –> <expression_2n+2>
)
```

Denote the propositional (possibly multi-valued (Feldman, Pietersma, & van Gemund, 2006)) **Wff** equivalent to <*expression*$_i$> as e_i ($1 \leq i \leq 2n + 2$). The Lydia compiler rewrites each arithmetic switch expression to the following system of equations:

$$\left| \begin{array}{l} (e_1 \Leftrightarrow e_2) \Rightarrow e_3 \\ (e_1 \Leftrightarrow e_4) \wedge \neg(e_1 \Leftrightarrow e_2) \Rightarrow e_5 \\ (e_1 \Leftrightarrow e_6) \wedge \neg(e_1 \Leftrightarrow e_4) \wedge \neg(e_1 \Leftrightarrow e_2) \Rightarrow e_7 \\ \qquad\qquad \vdots \\ (e_1 \Leftrightarrow e_{2n}) \wedge \neg(e_1 \Leftrightarrow e_2) \wedge \neg(e_1 \Leftrightarrow e_4) \wedge \cdots \wedge \neg(e_1 \Leftrightarrow e_{n-1}) \Rightarrow e_{2n+1} \\ \neg(e_1 \Leftrightarrow e_{2n}) \wedge \neg(e_1 \Leftrightarrow e_2) \wedge \neg(e_1 \Leftrightarrow e_3) \wedge \cdots \wedge \neg(e_1 \Leftrightarrow e_{n-1}) \Rightarrow c_{2n+2} \end{array} \right. \tag{2}$$

The use of the arithmetic switch is exemplified by the following model excerpt (part of a model of an airplane fuel switch valve):

```
system selectorValve(selectorPosition selector, bool enginePumpOn, _mdots)
{
                    ⋮
    if (h == hSelector.nominal) {
        flowOut == cond (selector) (
            selectorPosition.off –> mass.zero;
            selectorPosition.left –> flowLeft;
            selectorPosition.right –> flowRight;
            default –> mass.zero);
    }
}
```

The above arithmetic switch specifies that if the switch selector is in "off" position, then the value of the *flowOut* variable should be equivalent to the constant *mass.zero*. Similarly, if the switch is in position "left" the value of the *flowOut* variable should be equivalent to the value of the variable *flowLeft*, etc. Note that the default case can be omitted if there are corresponding expressions for all the values *expression*$_1$ can assume. If this is not the case and a default expression is not specified, Lydia will produce an error.

4.2 Qualitative Inequalities

Lydia enumeration variables can be compared. Consider the following model:

```
type letter = enum { a, b, c, d };

system qi()
{
    letter x, y;

    bool h;

    attribute observable(x, y) = true;
    attribute health(h) = h;
    attribute probability(h) = (h ? 0.95 : 0.05);

    if (h) {
        x < y;
    }
}
```

and the two observation vectors alpha_2 and alpha_3:

```
observation alpha_2
{
    (x == letter.a) && (y == letter.c);
}

observation alpha_3
{
    (x == letter.c) && (y == letter.a);
}
```

Observation alpha_2 results in the empty set of diagnoses while swapping x and y in alpha_3 results in a negative value for h as demonstrated by the transcript below:

```
lydia> diag qi alpha_1
lydia> fm
d1 = {  }
lydia> diag qi alpha_2
lydia> fm
d1 = { h = false }
```

Qualitative inequalities should be used with care as they may lead to combinatorial blow-ups. Consider an enumeration that can assume n values:

type <*name*> = enum { t_1, t_2, …, t_n };

A qualitative inequality such as $x \leq y$ would be internally expanded to

$$\begin{array}{c} [(x \Leftrightarrow t_1) \wedge (y \Leftrightarrow t_1)] \vee \\ \{(x \Leftrightarrow t_2) \wedge [(y \Leftrightarrow t_1) \vee (y \Leftrightarrow t_2)]\} \vee \\ \vdots \\ \{(x \Leftrightarrow t_n) \wedge [(y \Leftrightarrow t_1) \vee (y \Leftrightarrow t_2) \vee \cdots \vee (y \Leftrightarrow t_n)]\} \end{array} \tag{3}$$

which is difficult for translation to a normal form (CNF or DNF).

5. Predicates

Lydia constraints are specified in the system bodies as sequences of predicates separated by semicolons. After converting all predicates to a normal form, Lydia uses the normal forms conjunction as a system model.

5.1 Basic Predicates

Basic predicates contain expressions. Consider a model of a buffer:

```
system buffer(bool o, i)
{
    bool h;

    attribute probability(h) = (h ? 0.99 : 0.01);
    attribute health(h) = h;

    h => (o = i);
}
```

The last line of the above model (h => (o = i)) is a basic predicate.

5.2 Conditional Predicates

The **if** predicate is one of the most-commonly used constructs in Lydia. The **if** syntax is shown below.

```
if (<expression_1>) {
      <constraints_1>
} else if (<expression_2>) {
      <constraints_2>
            ⋮
} else if (<expression_{n-1}>) {
      <constraints_{n-1}>
} else {
      <constraints_n>
}
```

The **if** construct is a verbose way to write conditional expressions (cf. Sec. 4.1). In the next excerpt, which comes from the plane domain, we show some typical use of the **if** predicate. The model below shows a way to build qualitative models of sensors, where one or more measurements should coincide with the sensor readings (iff the sensor is functioning normally). Furthermore, the reasoner can discern different sensor faults (the model below is of a fuel sensor), given the sign of the time-derivative of a reading.

```
system sensor(mass real, deltaReal, indicated, deltaIndicated)
{
                ⋮
    if (h = sensorState.nominal) {
        indicated == real;
        deltaIndicated == deltaReal;
    }
    if (h = sensorState.stuckLow) {
        (indicated = mass.zero) || (indicated = mass.low);
        deltaIndicated = mass.zero;
    }
    if (h = sensorState.stuckHigh) {
        indicated = mass.high;
        deltaIndicated = mass.zero;
    }
}
```

It is often more elegant to model a succession of **if** and **else if** predicates as a single **switch** construct. The reader ought not to be confused by the resemblance of the Lydia **switch** construct to the one in some procedural languages (e.g., C and Java). In Lydia the **switch** construct is used in a more powerful, higher-level, way than a sequence of **if** and **else if** predicates. The syntax of the **switch** construct is the following:

```
switch (<expression_1>) {
    <expression_2> -> {
        <constraints_1>
    }
              ⋮
    <expression_n> -> {
        <constraints_{n-1}>
    } default -> {
        <constraints_n>
    }
}
```

If we denote the propositional (possibly multi-valued (Feldman et al., 2006)) **Wff** equivalent to <*expression*$_i$> as e_i and the conjunction of **Wff** contained in <*constraints*$_j$> as c_j, then the Lydia compiler rewrites every **switch** construct in a model to the following system of equations:

$$\left|\begin{array}{l}
(e_1 \Leftrightarrow e_2) \Rightarrow c_1 \\
(e_1 \Leftrightarrow e_3) \wedge \neg(e_1 \Leftrightarrow e_2) \Rightarrow c_2 \\
(e_1 \Leftrightarrow e_4) \wedge \neg(e_1 \Leftrightarrow e_2) \wedge \neg(e_1 \Leftrightarrow e_3) \Rightarrow c_3 \\
\qquad\qquad\qquad \vdots \\
(e_1 \Leftrightarrow e_n) \wedge \neg(e_1 \Leftrightarrow e_2) \wedge \neg(e_1 \Leftrightarrow e_3) \wedge \ldots \wedge \neg(e_1 \Leftrightarrow e_{n-1}) \Rightarrow c_{n-1} \\
\neg(e_1 \Leftrightarrow e_n) \wedge \neg(e_1 \Leftrightarrow e_2) \wedge \neg(e_1 \Leftrightarrow e_3) \wedge \ldots \wedge \neg(e_1 \Leftrightarrow e_{n-1}) \Rightarrow c_n
\end{array}\right. \tag{4}$$

The practical use of the Lydia **switch** construct is illustrated in a component model excerpt from the International Berthing and Docking Mechanism (IBDM)IBDMInternational Berthing and Docking Mechanism hard-docking system.

```
if (((motorSelector = EMAMotorSelection.primary) ?
                    stateMotor1 :
                    stateMotor2) = EMAState.healthy) {
   switch (regime) {
      EMARegime.latching –>
      {
         (current = range.inside) => (gearOut = ringGear.latching);
         (current = range.above) => (gearOut = ringGear.forceLatching);
      }
      EMARegime.latched –>
      {
         (current = range.below) => (gearOut = ringGear.latched);
      }
      EMARegime.unlatching –>
      {
         (current = range.inside) => (gearOut = ringGear.unlatching);
         (current = range.above) => (gearOut = ringGear.forceUnlatching);
      }
      EMARegime.unlatched –>
      {
         (current = range.below) => (gearOut = ringGear.unlatched);
      }
   }
}
```

5.3 Quantifiers

Lydia allows quantifiers over the elements of an array. These are expanded at compile time. There are two quantifiers: existential (**exists**) and universal (**forall**). The use of both should conform to the syntax shown next.

<**forall** | **exists**> (*index* in *start* .. *end*)
{
    <*constraints*>
}

Here, *index* is a variable identifier and *start* and *end* are both integer expressions (*start* does not need to be greater than *end*). Nesting of **forall** and **exists** blocks is allowed.

The use of the universal qualifier (**forall**) can be best illustrated in solving a small 4×4 Sudoku puzzle. The actual problem we will solve and its solution are shown in Figure 3. After we define the main data type of type *tile* in the model that follows below we need a system that constraints the elements of a 4-element vector to always assume different

1		3	
2			
			3
	2		1

1	4	3	2
2	3	1	4
4	1	2	3
3	2	4	1

Figure 3: An example 4×4 Sudoku puzzle (left) and its solution (right)

values. This is done with the help of the two nested **forall** constructs.

```
type tile = enum { V1, V2, V3, V4 };

system Everywhere(tile g[4])
{
    forall (i in 0 .. 3) {
        forall (j in i + 1 .. 3) {
            g[i] != g[j];
        }
    }
}
```

Note the use of the index variables (in this example i and j). Their scope is restricted to the body of the respective **forall** block. The index variables are always integers and they can be used in integer expressions as far as static evaluation of all array subscripts can be performed at model compilation time.

Having the basic inequality constraint (which we are going to apply row-wise, column-wise and block-wise) we can define the Sudoku grid. Solving this empty grid, will in fact generate all possible Sudoku instances.

```
system Grid(tile g[4][4])
{
    system Everywhere row[4], col[4], blk[4];

    forall (i in 0 .. 3) {
        row[i]([ g[i][0], g[i][1], g[i][2], g[i][3] ]);
        col[i]([ g[0][i], g[1][i], g[2][i], g[3][i] ]);
    }
    blk[0]([ g[0][0], g[0][1], g[1][0], g[1][1] ]);
    blk[1]([ g[0][2], g[0][3], g[1][2], g[1][3] ]);
    blk[2]([ g[2][0], g[2][1], g[3][0], g[3][1] ]);
    blk[3]([ g[2][2], g[2][3], g[3][2], g[3][3] ]);
}
```

In the above system, the **forall** construct is used to repeat constraints for each row and column of the grid. What remains is to initialize the hint tiles and to feed the resulting system to a solver. The next and final excerpt from this example defines the top-level

Sudoku problem, supplying constraints for the known hints.

```
system Sudoku()
{
    tile g[4][4];
    attribute health(g) = true;
    attribute probability(g) = 0.25;

    system Grid grid(g);

    g[0][0] = tile.V1; g[0][2] = tile.V3;
    g[1][0] = tile.V2;
    g[2][3] = tile.V3;
    g[3][1] = tile.V2; g[3][3] = tile.V1;
}
```

The last system does not include any quantifiers, but we will use it to complete our example. The solution of the Sudoku puzzle specified above will be the only satisfiable term of the Disjunctive Normal Form (DNF) of the above system.

Solving the example above with, for example, the Lydia dense-encodings DNF solver indeed yields the correct solution (cf. Figure 3).

```
lydia> modes
g[0][0] = V1, g[0][1] = V4, g[0][2] = V3, g[0][3] = V2,
g[1][0] = V2, g[1][1] = V3, g[1][2] = V1, g[1][3] = V4,
g[2][0] = V4, g[2][1] = V1, g[2][2] = V2, g[2][3] = V3,
g[3][0] = V3, g[3][1] = V2, g[3][2] = V4, g[3][3] = V1
```

The next two excerpts are from a model of the IBDM domain. The first one stipulates that if some condition is satisfied, then, for two arrays, one or more elements of each array should assume a certain value[6].

```
if ((primaryRingGear = ringGear.forceLatching) or
    (primaryRingGear = ringGear.forceUnlatching)) {
    exists (i in 0 .. 11) {
        (primaryGearBoxState[i] = gearBoxState.jammed) or
        (primaryLatchState[i] = latchState.jammed);
    }
}
```

Finally, we illustrate the workings of the **forall** construct with one more example from the IBDM domain. In this excerpt we couple a dozen of gear boxes to a dozen of structural latch assemblies (these are all systems which share variables).

```
forall (i in 0 .. 11) {
    gearBox[i](primaryGearBoxState[i], _ldots, latchAssemblySecondaryCrank[i]);
    structuralLatchAssembly[i](primaryLatchState[i], _ldots, latchRoller[i]);
}
```

6. The intended meaning of the model is that an increased force when moving the primary or secondary ring gears implies the existence of a gear box (there are twelve gear-boxes driven by the two ring gears) or a latch which is mechanically jammed.

References

Feldman, A., Pietersma, J., & van Gemund, A. (2006). A multi-valued SAT-based algorithm for faster model-based diagnosis. In *Proc. DX'06*.

Feldman, A., & van Gemund, A. (2006). A two-step hierarchical algorithm for model-based diagnosis. In *Proc. AAAI'06*.

van Gemund, A. (2002). The Lydia approach to diagnostic systems modeling. Tech. rep. PDS-2002-004, Delft University of Technology.

van Gemund, A. (2003). Lydia version 1.1 tutorial. Tech. rep. PDS-2003-001, Delft University of Technology.