NOTRE TITRE

Subtitle Text, if any

Vincent Lecrubier Bruno d'Ausbourg
ONERA DTIM/LAPS
Toulouse, France
{lecrubier,ausbourg}@onera.fr

Yamine Aït-Ameur
ENSEEIHT
Toulouse, France
yamine@enseeiht.fr

Abstract

This is the text of the abstract.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

In the last 30 years, the aerospace domain has successfully devised rigorous languages, methods and tools for the development of safe functionally-correct software. In the same time, interactive software received a very lower amount of attention. However, highly interactive Human Machine Interfaces (HMI) are now appearing in critical embedded systems and particularly in aeronautics: new generations of aircraft cockpits make use of sophisticated electronic and digital devices that may be driven by more and more complex software applications endowed with a huge number of lines of code. These applications must behave as intended with a high degree of assurance because of their criticality. An error in these software components may have catastrophic consequences.

So there is a real stake to master the development and the implementation of these critical interactive applications. The heart of the problem is that the standard processes for the development of safety critical software in aeronautics are not really suitable for interactive software design for which more iterative methods that involve end-users and that mix design and tests are still required. Moreover, the stakeholders that participate in the design and implementation of these applications do not have common means to express properly and rigorously the intended behaviour of these interfaces.

In this paper we advocate devising a well-defined domain specific language for representing the behavior of the designed interactive software in a way that allows, on the one hand, system designers to iterate on their designs before injecting them in a development process and, on the other hand, system developers to check their software against the chosen design.

Section 2 details some motivations to devise such a new DSL.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. Motivation for a language for ....

A lot of research work have focused on how to design, program and verify functional concerns for critical systems and more particularly aeronautical systems. HMI systems did not benefit from the same attention and efforts.

2.1 Context

A significant amount of work has focused on devising models for the development process of software systems in the field of software engineering.

The system development process in critical domains as, for instance, in aeronautics inherited these models. This process is now widely based on the use of standards that take into account the safety and security requirements of the systems under construction. In particular the DO178C standard [2], in aeronautics, defines very strict rules and instructions that must be followed to produce software products, embedded systems and their equipments. The objective is to ensure that the software performs its function with a safety level in accordance with the safety requirements.

The HMI development does not follow the same processes. Nevertheless, in aeronautics, HMI systems are now made up by multiple hardware and software components embedded in aircraft cockpits. These systems are large and complex artifacts that also face tough constraints in terms of usability, security and safety. They support interactive applications that must behave as intended with a high degree of assurance because of their criticity. An error in the software components that implement interactions in these applications may lead to a human or system fault that may have catastrophic effects.

For example, the BEA report [**?**] about the crash of Rio-Paris AF 447 A330 Airbus establishes that, during the flight, interface system displayed some actions to be performed by the pilot in order to change the pitch of the aircraft and to nose it up while it was stalling. These indications should clearly not have been displayed. Indeed, by following those erroneous displayed instructions the pilot increased the stalling of the aircraft.

In fact, in the industrial context, the development process of critical interactive embedded applications stays very primitive. The usual notations are essentially textual and coding is generally performed from scratch or by reusing previous developments based themselves on textual specifications. In aeronautics, the produced code must be in conformance with the ARINC 661 standard [3]. It may be noticed that some tools recently appeared to enhance the design and coding stages of these systems. But these tools, as for instance Scade Display [4], deal mainly with presentation layers of the systems and do not deal with their complex functional behaviour. In this context, the validation process of the interactive applications is very restricted and poor because it resides practically only in a massive test effort and in expensive evaluation phases at

the end of the development process. Moreover there is no actual formal reference to check the implementation is in conformance with. So new approaches and new paradigms are today needed to help in the development process of critical interactive embedded applications.

2.2 Requirements

We believe that part of these issues are due to the lack of a well-defined language for representing interactive software design in a way that allows, on the one hand, system designers to iterate on their designs before injecting them in a development process and, on the other hand, system developers to check their software against the chosen design. Such a hub language (similar to VHDL for hardware description and Scade for safety-critical control and command software development) would bring increased flexibility in the development process leading not only to easier iterations within and between its different phases but also to the automation of parts of the process. Some requirements can be expressed concerning such a language.

- **REQ1.** The language is a domain specific language for HMI systems of embedded systems. It permits to use and manipulate the HMI design concepts and so to describe properly the interactive behaviors the software will implement. In particular the language must permit to describe and to handle both continuous and discrete interactions.
- **REQ2.**The language is a pivot language that must be read, understood and written by different stakeholders coming from different scientific disciplines
- **REQ3.** The language is formal. Its semantics is clear and unambiguous.
- **REQ4.** Descriptions in this language may be used to generate a safe code for an interactive application and this code may be compatible with the ARINC 661 Standard [3].
- **REQ5.** The language permits to design in the same way any interactive component of an interactive system through the desciption of its input, output, and internal states.
- **REQ6.** The language permits to devise and to describe complex interactive systems by giving some syntactical constructions to assemble and compose interactive components.

We devised such a language (the LIDL Interaction Definition Language) to help the definition of critical interactive applications. The following section presents the LIDL language.

3. The LIDL language

3.1 Informal description

An interactive entity is an entity that can interact with other entities by exchanging information in the form of data flows. A human being is an interactive entity, a controllable embedded system is an interactive entity, a human-machine interface is an interactive entity. They are all able to exchange information in different directions, using different means (eyes, hands, bus, network, screen, keyboard...)

A first important thing to notice is that, in order to interact, two interactive entities must establish a data flow, by matching an entity's input to another entity's output.

LIDL is a language that allows to specify interactive entities, called interactors. A LIDL interactor is described through two different aspect: interfaces and interactions. Interfaces are the description of the data flows between the interactor and other interactive entities. Interactions are the description of the link between input data flows and output data flows.

⟨*interaction*⟩ ::= '(' ⟨*element*⟩* ')'

⟨*element*⟩ ::= ⟨*interaction*⟩
| ⟨*word*⟩

⟨*word*⟩ ::= ? anything except parentheses ?

Figure 1. Expression grammar of LIDL

As a first short example, Listing 1 is a piece of LIDL code that defines the interface associated with a button interactor. This definition specifies that a button has a title, which is a text output to the user, and it can receive clicks, which are activation signals coming from the user.

```
interface Button is
  {
    title: text out,
    click: activation in
  }
```

Listing 1. Interface of the speed controller

Listing 2 is a piece of LIDL code that defines the interaction associated with a button interactor. It specifies a pattern of use `(Button(...)with title(...)triggerring(...))` , along with the interfaces associated with arguments, the interface of the interaction itself `Button` , and an interaction expression which is the definition of this interaction.

```
interaction
  (Button (status: activation in)
  with title (theTitle: text in)
  triggering (onClick: activation out))
implementing
  Button
is
  (bind(this) : (when (status) : (all
      ((this.title)=(theTitle))
      ((onClick)=(this.click))
  )))
```

Listing 2. Interaction of a button

Listing 3 is an instantiation of the interaction defined in Listing 2. It creates a Button whose status is always `active` , which has the title `"Ok"` , and which, when clicked, activates another interaction called `Beep` .

```
(myButton) = (Button(active) with title ("Ok")
    triggering (beep))
```

Listing 3. Usage of a button

3.2 Syntax

LIDL is very general, and so is its syntax. The syntax use lots of parentheses, which is a solution to two contradictory requirements: **REQ2** and **REQ3**. Interaction expressions are used to compose interactions in order to form more complex interaction patterns. This grammar is very simple, here is the extended Backus Naur form (EBNF) of the interaction grammar of LIDL:

Said simply, an interaction is expressed as a sentence between parentheses. The interactions it refers to are also between parentheses. LIDL syntax is made in such a way that every parentheses pair `(...)` is an interaction, and reciprocally.

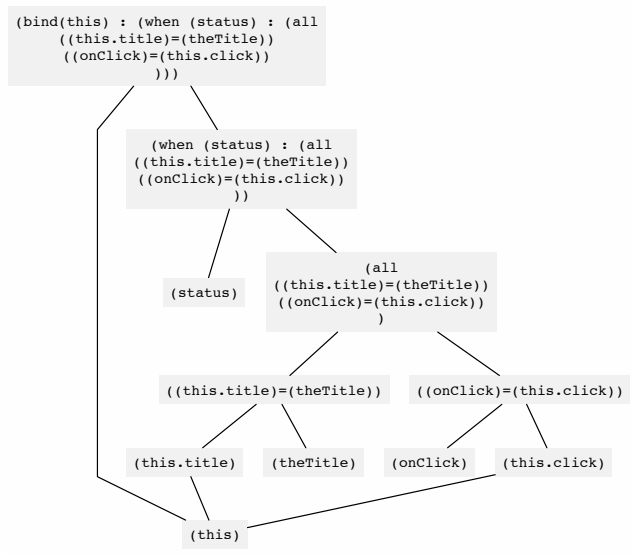


Figure 2. The expression tree associated with Listing 2

The interaction expression in Listing 2 is composed of several interactions, each between parentheses. Figure 2 shows the expression tree associated with this expression.

3.3 Synchronous execution

LIDL systems are synchronous. This means that interactions are evaluated at discrete points in time, all at once. The interaction expression defining a system is evaluated at discrete points in time, called steps. Every expression of this composed expression will be evaluated at every step.

Synchronous execution avoids spaghetti behaviours. The state of the system is explicitly defined for each execution step. This makes reasoning about code much simpler, verification is much easier than with other asynchronous approaches.

Figure 3 shows the synchronous execution of a LIDL system. TODO : talk about the transition function here

3.4 Activation

One of the contribution of the LIDL language is the built-in notion of activation. Each piece of data in LIDL is attached to an activation. This notion is some kind of abstraction over other similar features like `null` values.

An interaction's activation represents the fact that the interaction exists and is active at a point in time, or not. For example, if an interaction represents an event, then it will only be activated when the event happens. As another example, the assignment interaction `$=$` is only effective when it is active. If an interaction's value is not defined anywhere, then its activation is false.

In previous approaches, and in some other languages, a difference is made between events and flows, which are considered as two different first-class entities. Some languages only allow one of those. In these approaches, events represent data defined at discrete points in time, while flows represent data defined on continuous time intervals. But when we think about it, the only difference between an event and a flow lies in the domain. For events this set is discrete. For flows this set is continuous.

The logical conclusion of this remark is that the merger of these two concepts needs to include the indicator function of the domain. This indicator function is the activation.

3.5 Flow direction

Reciprocity (in matches out)

3.6 Formal description

3.6.1 Identifier

Definition 1 (Identifier)**.** $\mathbb{I}$ *is the set of all possible identifiers. An identifier* $i \in \mathbb{I}$ *is noted* $i =$ `foo`

Each interaction instance has an identifier associated with it. The identifier associated with an interaction is the LIDL code of this interaction, with white spaces removed. For example the interaction noted `(when (click) : (beep) )` in LIDL is associated to the identifier `(when(click):(beep))`.

3.6.2 LIDL system

The basic notion is the interactive component named `interaction` which produces output and new current internal state flows from input and previous internal state flows. Each instance of an interaction is associated with an identifier.

Definition 2 (LIDL System)**.** $\mathcal{P}(\mathbb{I})$ *is the set of all possible LIDL systems.*

A LIDL system is a composition of interactions that defines a more complex interaction.

Homoiconicity le fait qu'un programme puisse se représenter dans les structures de données du langage lui-même

Definition 3 (Value)**.** $\mathbb{V} = \{\bot\} \cup \{\top\} \cup \mathbb{B} \cup \mathbb{R} \cup \mathbb{T}$ *is the set of all atomic values of the LIDL language, where* $\mathbb{B}$ *the booleans,* $\mathbb{R}$ *the set of real numbers,* $\mathbb{T}$ *the set of all possible texts. The set of all possible values of the LIDL language, including composed values is* $\mathbb{V}^*$.

There are 4 base data types in LIDL:

- `activation`, whose value set is $\{\top\}$, the active value.
- `boolean`, whose value set is $\mathbb{B} = \{true, false\}$
- `number`, whose value set is $\mathbb{R}$, the real numbers
- `text`, whose value set is $\mathbb{T}$, the set of all possible texts

Each

Definition 4 (Memory)**.** $\mathbb{M} = \mathbb{V}^{\mathbb{I}}$ *is the set of all possible memories of a LIDL system. A memory* $m \in \mathbb{M}$ *is a function noted* $m : \mathbb{I} \to \mathbb{V}$.

Definition 5 (Execution)**.** $\mathbb{E} = \mathbb{M}^{\mathbb{N}}$ *is the set of all possible executions of a LIDL system, where* $\mathbb{N}$ *is the set of natural numbers. An execution is a function noted* $e : \mathbb{N} \to \mathbb{I} \to \mathbb{V}$

Definition 6 (Valuation)**.** $\texttt{foo}\,{}^{e}_{n}$ *denotes the value associated to the identifier* `foo`*, in the* n^{th} *memory of an execution* e.

$$\forall n < 0 \quad \forall\, \texttt{x} \in \mathbb{I} \quad \texttt{x}\,{}^{e}_{n} = \bot$$

$$\forall n \geq 0 \quad \forall\, \texttt{x} \in \mathbb{I} \quad \exists f \quad \texttt{x}\,{}^{e}_{n} = f()$$

In the following tables, values on the left side of the vertical bar are the ones which have to be computed beforehand, so that values on the right side of the vertical bar can be computed. Values noted as letters such as x are supposed to be different from $\bot$.

Here is an example, where input1 and input2 are input values, and ouput1 and output2 are output values. The example reads like this : on execution step n when input1=a and input2=b, then output1=x and output2=y...

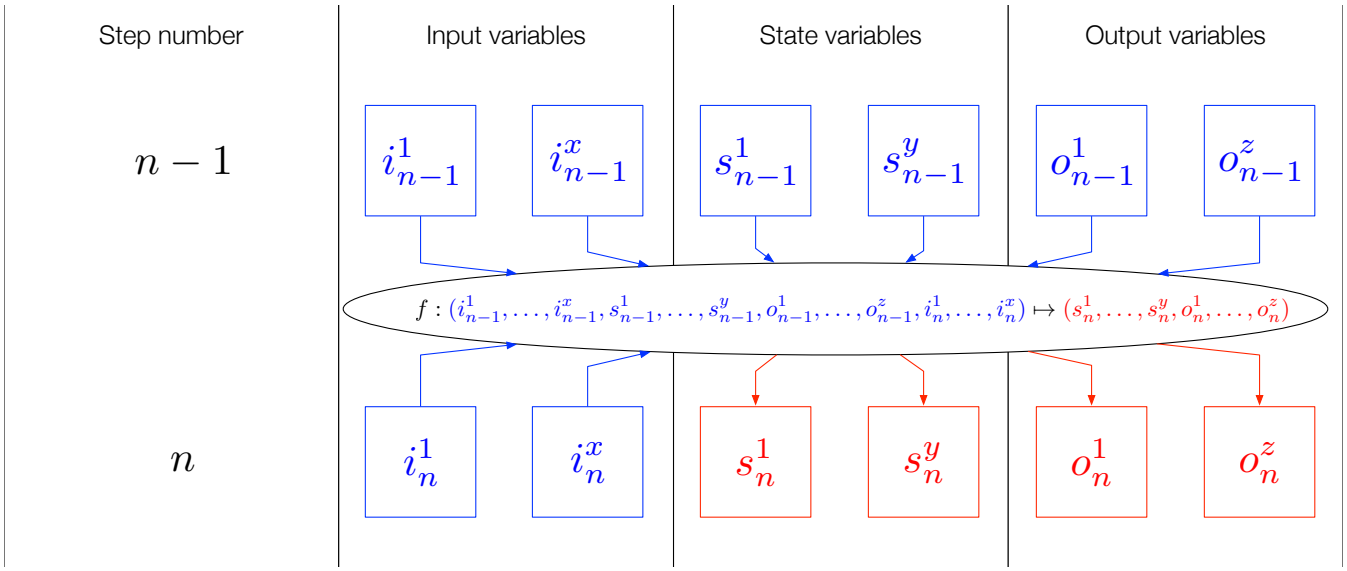


Figure 3. Execution of a LIDL System with x input variables $\{i^1 \ldots i^x\}$, y state variables $\{s^1 \ldots s^y\}$, and z output variables $\{o^1 \ldots o^z\}$. The transition function is named f, its domain is in blue, its codomain is in red.

(input1) ^e_n	(input2) ^e_n	(output1) ^e_n	(output2) ^e_n
a	b	x	y

Affectation

((a) = (b)) ^e_n	(b) ^e_n	(a) ^e_n
$\bot$	$\bot$	$\bot$
$\bot$	b	$\bot$
$\top$	$\bot$	$\bot$
$\top$	b	b

All

(all(a)(b)) ^e_n	(a) ^e_n	(b) ^e_n
$\bot$	$\bot$	$\bot$
$\top$	$\top$	$\top$

Either

(either(a)(b)) ^e_n	(a) ^e_n	(b) ^e_n
$\bot$	$\bot$	$\bot$
$\top$	$\top/\bot$	$\bot/\top$

Always

(always(a)) ^e_n	(a) ^e_n
$\bot$	$\top$
$\top$	$\top$

Previous

(previous(a)) ^e_n
(a) $^e_{n-1}$

Arithmetic operators

(a) ^e_n	(b) ^e_n	((a) + (b)) ^e_n
$\bot$	$\bot$	$\bot$
$\bot$	b	$\bot$
a	$\bot$	$\bot$
a	b	$a+b$

If then else

(a) ^e_n	(b) ^e_n	(c) ^e_n	(if(a)then(b)else(c)) ^e_n
$\bot$	$\bot$	$\bot$	$\bot$
$\bot$	$\bot$	c	$\bot$
$\bot$	b	$\bot$	$\bot$
$\bot$	b	c	$\bot$
$true$	$\bot$	$\bot$	$\bot$
$true$	$\bot$	c	$\bot$
$true$	b	$\bot$	b
$true$	b	c	b
$false$	$\bot$	$\bot$	$\bot$
$false$	$\bot$	c	c
$false$	b	$\bot$	$\bot$
$false$	b	c	c

Init

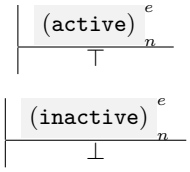
(init) ^e_n
$\begin{cases} \top & \text{if } n = 0 \\ \bot & \text{else} \end{cases}$

Constant litterals

(”atext”) ^e_n
“a text”

(1234) ^e_n
1234

(true) ^e_n
true



3.6.3 Data types and value sets

Basic data types

$$\begin{array}{lcl} \texttt{activation} & = & \{\bot\} \cup \{\top\} \\ \texttt{boolean} & = & \{\bot\} \cup \{true, false\} \\ \texttt{number} & = & \{\bot\} \cup \mathbb{R} \\ \texttt{text} & = & \{\bot\} \cup all\_possible\_texts \end{array}$$

Compound types (examples)

$$\begin{array}{lcl} [\texttt{number}] & = & \{\bot\} \cup \bigcup_{n\in\mathbb{N}} \texttt{number}_n^e \\ (\texttt{number}, \texttt{number}) & = & \{\bot\} \cup (\texttt{number} \times \texttt{number}) \\ \{\texttt{x} : \texttt{number}, \texttt{y} : \texttt{number}\} & = & \{\bot\} \cup (\texttt{number} \times \texttt{number}) \\ |\texttt{number}, \texttt{boolean}| & = & \texttt{number} \cup \texttt{boolean} \end{array}$$

3.6.4 Lustre examples of some base interactions

```
node Affect
  (actX:bool, valX:int, actB:bool, valB:int)
returns
  (actA:bool, valA:int);
let
  actA = actX and actB;
  valA = valB;
tel
```

```
node All
  (actX:bool, valX:int)
returns
  (actA:bool, valA:int, actB:bool, valB:int);
let
  actA = actX;
  valA = valX;
  actB = actX;
  valB = valX;
tel
```

```
node Always
  (actX:bool, valX:int)
returns
  (actA:bool, valA:int);
let
  actA = true;

tel
```

3.6.5 Composition

LIDL offers **referential transparency**, which means that composition of interactions is very simple and works by simple substitution.

3.7 Formal semantics

We define the actual behaviors of a LIDL interaction system by using a semantics of traces that permits to characterize the behaviors of a LIDL description in terms of execution traces. Intuitively every behavior of a LIDL interaction systems corresponds to the sequences of the values each LIDL variable has. These values define a *state* of the system and a sequence of states defines a behavior of this system.

Definition 7 (Memory)**.** *Let I a set of identifiers of LIDL. V is a set of values and $\bot$ denotes an undefined value. We define $A = \{false, true\}$. Every function $\sigma : I \to A \times (V \cup \{\bot\})$ defines a memory of the LIDL interaction system.*

Every state of a LIDL interaction system is characterized by a memory that gives a value $\sigma(id)$ to each $id \in I$. This value is a pair (a, v) with $a \in A$ and $v \in (V \cup \bot)$: a indicates if id is *active* or not. If not, v is not significant and stays undefined ($v = \bot$).

Definition 8 (Trace)**.** *An execution trace Σ is a non empty, finite or infinite, sequence of memories.*

Some identifiers play a particular role in the LIDL interaction systems. They are used to denote *activation signals* that are not valuated. Their values are in $A \times \{\bot\}$. By default $\forall id \in I, \sigma(id) = (false, \bot)$. In other words, by default, every value is not active. To become active and to get a significant data value, a behavior of the LIDL interaction system must be activated.

3.7.1 Semantics of LIDL expressions over finite traces

The value (in $A \times (V \cup \{\bot\})$ of a LIDL expression E over a finite trace $\Sigma = (\sigma_0, \ldots, \sigma_n^e)$ is defined by induction on the structure of E. We denote $\Sigma \vdash E|(a, v)$ to mean that E has the value (a, v) ($a \in A$ and $v \in A \times (V \cup \{\bot\})$) over Σ.

Constants values. If k denotes a bool, number or text constant then $\Sigma \vdash$ (k)$|(true, k)$. A particular activation constant, $active$ is defined in LIDL such that $\Sigma \vdash (active)|(true, \bot)$.

Variables values. $(\sigma_0, \ldots, \sigma_n^e) \vdash (x)|\sigma_n^e(x)$ if x denotes a variable in the language.

Boolean and arithemtic operators. If $E_1, \ldots, E_m$ denote expressions in LIDL and $\star$ is a boolean or arithmetic operator then

$$\frac{\Sigma \vdash (E_1)|(a_1, v_1), \ldots, \Sigma \vdash (E_m)|(a_m, v_m)}{\Sigma \vdash \star(E_1, \ldots, E_m)|(\bigwedge_{i=1}^m a_i, \star(v_1, \ldots, v_m))}$$

Operator previous. If E denotes an expression in LIDL,

$$\sigma_0 \vdash (\text{previous}(E))|(false, \bot)$$

$$\frac{\Sigma \vdash (E)|(a, v)}{\Sigma \cdot \sigma \vdash (\text{previous}(E))|(a, v)}$$

Operator isactive. If E denotes an expression in LIDL,

$$\frac{\Sigma \vdash (E)|(a, v)}{\Sigma \vdash (\text{isactive}(E))|(true, a)}$$

Operator if then else. If E_1 is a boolean expression in LIDl and if E_2 and E_3 are expressions in LIDL,

$$\frac{\Sigma \vdash (E_1)|(a_1, v_1), \Sigma \vdash (E_2)|(a_2, v_2), \Sigma \vdash (E_3)|(a_3, v_3)}{\Sigma \vdash \text{if}(E_1)\text{then}(E_2)\text{else}(E_3)| \begin{cases} (false, \bot) & \text{if} a_1 = false \\ (a_2, v_2) & \text{if}(a_1, v_1) = (true, true) \\ (a_3, v_3) & \text{if}(a_1, v_1) = (true, false) \end{cases}}$$

3.7.2 Compatibility of an infinite trace with a LIDL description

Behaviors that are described by LIDL interactions are infinite. The semantics of a LIDL description corresponds to the set of the infinite execution traces that are compatible with it. We note $\Sigma \vdash D$ the fact that an infinite trace Σ is compatible with the LIDL description D.

An infinite trace denotes a behavior of D if and only if every finite prefix of D is compatible with D :

$$(\sigma_0, \ldots, \sigma_n^e, \ldots) \vdash D \iff \forall n \geq 0, (\sigma_0, \ldots, \sigma_n^e) \vdash D$$

The compatibility of finite traces with D is defined by the following rules. A LIDL description is considered as (and can be un-

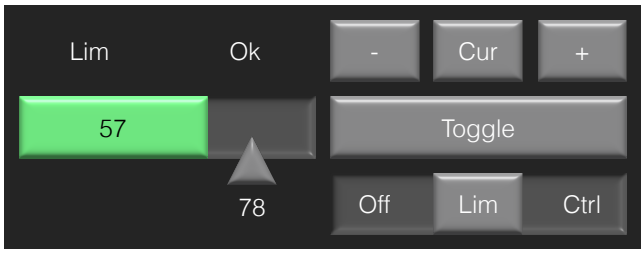


Figure 4. A mockup of the example GUI

fold in) a set of elementary interactions, depicted by equations, regardless of its syntactical structure. LIDL is a declarative language so the order of the equations is unimportant.

A new identifier $\langle(interaction)\rangle \in I$ is associated with every elementary interaction $(interaction)$. $\langle(interaction)\rangle$ denotes the activation signals that trigger $(interaction) : \sigma(\langle(interaction)\rangle) \in A \times \{\bot\}$.

Compatibility with an equation. If x is a LIDL identifier, E is a LIDL expression :

$$\frac{(\sigma_0, \ldots, \sigma_n^e) \vdash \text{(E)} | (a, v), \sigma_n^e(\text{x}) = (a, v), \sigma_n^e(\langle(\text{x=E})\rangle) = (true, \bot)}{(\sigma_0, \ldots, \sigma_n^e) \vdash \text{(x = E)}}$$

Compatibility with composition. A set of interactions defines a new interaction. If int_1 and int_2 are two interactions

$$\frac{\Sigma \vdash (int_1), \Sigma \vdash (int_2)}{\Sigma \vdash ((int_1)(int_2))}$$

Compatibility with a guarded interaction. Let G an activation identifier in LIDL ($\sigma(\text{I}) \in A \times \{\bot\}$) and int a LIDL interaction :

$$\frac{\begin{array}{c}(\sigma_n^e(\text{G}) = (true, \bot), \sigma_n^e(\langle(\text{when(G):}(int))\rangle) = (true, \bot), \\ \sigma_n^e(\langle(int)\rangle) = (true, \bot)\end{array}}{(\sigma_0, \ldots, \sigma_n^e) \vdash (\text{when(G):}(int))}$$

4. An illustrative LIDL description

To illustrate LIDL capabilities, we will use it to describe a simple graphical user interface (GUI). The example is a typical speed controller. Three modes are available:

- `Off`: Manual control of the speed
- `Lim`: Limiter, the speed should stay under a certain value
- `Ctrl`: Controller, the speed should stay around a certain value

Three buttons are available to control the target speed:

- `+` to increment the target speed
- `-` to decrement the target speed
- `Cur` to set the target speed to the current actual speed

```
interface
  SpeedController
is
  {
    system: {
      desiredMode:  text out,
      desiredSpeed: number out,
      actualMode:   text in,
      actualSpeed:  number in
    },
    user: {
      mode: Label,
      status: Label,
      actual: Gauge,
      desired:  Slider,
      increment:  Button,
      decrement:  Button,
      current:  Button,
      toggle: Button,
      switch: SegmentedSwitch
    }
  }
```

Listing 4. Interface of the speed controller

```
interaction
  (TheSpeedController)
implementing
  SpeedController
is
  (bind ({
    system:{
      desiredMode: (theDesiredMode),
      desiredSpeed: (theDesiredSpeed),
      actualMode: (theActualMode),
      actualSpeed: (theActualSpeed)
    },
    user:{
      mode: (Label (active)
        with value (theActualMode)),
      status: (Label (active)
        with value (
          if ((theActualMode) != (theDesiredMode))
          then ("Wrong mode")
          else if (((theDesiredMode) != ("Off"))
          and((theActualSpeed)>(theDesiredSpeed)))
          then ("Over speed")
          else ("Ok")
        )),
      actual: (Gauge (active)
        with value (theActualSpeed)),
      desired: (Slider (active)
        with value (theDesiredSpeed)
        selecting (new(theDesiredSpeed))),
      increment: (Button (active)
        with text ("+")
        triggering ( (new(theDesiredSpeed)) =
        ((previous(theDesiredSpeed))+(5)))),
      decrement: (Button (active)
        with text ("-")
        triggering ( (new(theDesiredSpeed)) =
        ((previous(theDesiredSpeed))-(5)))),
      current: (Button (active)
        with text ("Cur")
        triggering ( (new(theDesiredSpeed)) =
        (round (theActualSpeed) to (5)))),
      toggle: (Button (active)
        with text ("Toggle")
        triggering ( (new(theDesiredMode))=(
          if((previous(theDesiredMode))==("Off"))
          then(theSelectedMode)
          else("Off")
        ))),
      switch: (SegmentedSwitch (active)
        with choices (["Off","Lim","Ctrl"])
        selecting (theSelectedMode))
    }
  }) : (all
    (make (theDesiredSpeed) flow)
    (make (theDesiredMode) flow)
  ))
```

Listing 5. The speed controller interaction

```
interaction
  (TheSpeedController)
implementing
  SpeedController
is
  (bind ({
    system:{
      desiredMode: (theDesiredMode),
      desiredSpeed: (theDesiredSpeed),
      actualMode: (theActualMode),
      actualSpeed: (theActualSpeed)
    },
    user:{
      mode:
        (bind(x1) :
          ((all
            ((x1.value)=(theActualMode))
          )=(active))),
      status:
        (bind(x2) :
          ((all((x2.value)=(if ((theActualMode) !=
               (theDesiredMode))
            then ("Wrong mode")
            else if (((theDesiredMode) != ("Off"))
                 and((theActualSpeed)>(
                theDesiredSpeed)))
            then ("Over speed")
            else ("Ok")))
          )=(active))),
      actual:
        (bind(x3) :
          ((all
            ((x3.value)=(theActualSpeed))
          )=(active))),
      desired:
        (bind(x4) : (
          (all
            ((x4.value)=(theDesiredSpeed))
            ((new(theDesiredSpeed))=(x4.selection)
                )
          )=(active))),
      increment:
        (bind(x5) : (
          (all((x5.value)=("+"))(((new(
              theDesiredSpeed))=((previous(
              theDesiredSpeed))+(5)))=(x5.click)))
              =(active))),
      decrement:
        (bind(x6) : (
          (all((x6.value)=("-"))(((new(
              theDesiredSpeed))=((previous(
              theDesiredSpeed))-(5)))=(x6.click)))
              =(active))),
      current:
        (bind(x7) : (
          (all((x7.value)=("Cur"))(((new(
              theDesiredSpeed))=(round (
              theActualSpeed) to (5)))=(x7.click))
              )=(active))),
      toggle:
        (bind(x8) : (
          (all((x8.value)=("Toggle"))(
          ( (new(theDesiredMode))=(
            if((previous(theDesiredMode))==("Off")
                )
              then(theSelectedMode)
              else("Off")
            ))=(x8.click)))=(active))),
        switch:
          (bind(x9) : ((all
            ((x9.choices)=(["Off","Lim","Ctrl"]))
            ((theSelectedMode)=((["Off","Lim","Ctrl"
                ])[(x9.selection)])))=(active))),
    }
  }) : (all
    ((theDesiredSpeed) =
      (if ((new(theDesiredSpeed)) is active)
      then (new(theDesiredSpeed))
      else (previous(theDesiredSpeed))))
    ((theDesiredMode) =
      (if ((new(theDesiredMode)) is active)
      then (new(theDesiredMode))
      else (previous(theDesiredMode))))
  ))
```

Listing 6. The speed controller interaction unfolded

Once totally unfolded, the SpeedController uses the following base interactions :

- `bind$:$`
- `$=$`
- `all$$`
- `if$then$else$`
- `previous$`
- `$+$`
- `$-$`
- `$==$`
- `$and$`
- `round$to$`
- `{system:{desiredMode:$,desiredSpeed:$...},user:{...}}`
- `5`
- `active`
- `"Wrong Mode"`
- Other constants...

5. Assessment

We provide an assessment of the description capabilities of the defined language.

- Flows : continuous
- Events : discrete
- composition of flows and events to build behaviors

These notions are embedded in a reusable component implementing the notion of interactor.

Give a positionning with respect to the requirements

6. Harnessing LIDL

- Verification
- Code generation
- Test and automatic test generation

Give a positionning with respect to the requirements

7. Related work

- Notion of interactor: York,
- Extended LUSTRE et lien avec le langage Lustre

None of the previous approaches handle

8. Conclusion and future work

- A language for concurent interface design among different stakeholders, programming, verifying, testing
- Code generation
- prototype

Future work

- Tool support
- Handling the description and the verification of a whole system including the human. We believe that the integration of the human is eased by the abstraction level in which LIDL components are described [CITER RUSHBY]
- Verification of other properties, Honesty, freshness, device overload
- temporal properties related to delays and their impact on the architecture of the interface

8.1 TO DO list

- yamine
  - Choisir une figure pour l'étude de cas parmi les 10 choix donnés
- bruno
- vincent
  - ~~renommage pour éviter homonymie~~
  - ~~supprimer les bind, this, all~~
  - exprimer les basic constructs
  - éviter les éléments inachevés
  - ~~flow pourrait devenir flow_mk~~
  - ~~changer figure étude de cas.~~

A. Appendix

A.1 Generic

Some generic interactions are necessary to make the language useful. Listing 7 shows the most important ones.

```
// (if(cond)then(a)else(b)) is similar
// to the ternary operator cond?a:b in C
interaction
  (if (condition:boolean in)
  then (a:<type>)
  else (b:<type>))
implementing
  -<type>
is
  'native'

// ((a) is active) projects (a) into the
// activation type. It removes the value of
// (a), only keeping its activation
interaction
  ((a:<type> in) is active)
implementing
  activation out
is
  'native'

// ((a) is true) takes a boolean and returns
// an activation which is active if and only
// if (a) is active and true
interaction
  ((a:boolean in) is true)
implementing
  activation out
is
  'native'

// (bind(a):(b)) is equivalent to (a),
// but it also execute the behavior (b)
// in parallel. (b) can have an effect
// on (a) by using variables
interaction
  (bind (a:<type> data) : (b:activation out))
implementing
  <type> data
is
  'native'

interaction
  ((a: <type> in) default (b: <type> in))
implementing
  <type> out
is
  (if ((a) is active) then (a) else (b))
```

Listing 7. Section of the standard LIDL library regarding generic interactions

A.2 Lists

```
// Get list element
interaction
  ( (a:[<type>]in) [ (index:number in) ] )
implementing
  <type> out
is
```

```
  'native'

// Set list element
interaction
  ( (a:[<type>]out) [ (index:number in) ] )
implementing
  <type> in
is
  'native'

// Compose list
interaction
  ([(first:<type>in),(second:<type>in),...,(last:<
      type>in)])
implementing
  <type> out
is
  'native'


// Decompose list
interaction
  ([(first:<type>out),(second:<type>out),...,(last
      :<type>out)])
implementing
  <type> in
is
  'native'
```

Listing 8. Section of the standard LIDL library regarding lists interactions

A.3 Behavior

What is a behavior ? Very generally, a behavior is some kind of action that is executed at some points in time. This is why we define the `Behavior` interface to be equivalent to the `activation in` interface. This means that interactions implementing the `Behavior` interface receive a flow of `activation`. When they receive the `active` value, they are effective, when they receive the `inactive` value, they are not effective.

The LIDL standard library defines some useful basic behaviors, as shown on Listing 9.

```
interface
  Behavior
is
  activation in

interaction
  ((a:<type>out) = (b:<type>in))
implementing
  Behavior
is
  'native'

interaction
  (all (a:activation out) (b:activation out))
implementing
  Behavior
is
  'native'

interaction
  (either (a:activation out) (b:activation out))
implementing
  Behavior
is
  'native'

interaction
  (always (a:activation out))
implementing
  Behavior
is
  'native'

interaction
  (when (condition:activation in) : (effect:
      activation out) )
implementing
  Behavior
is
  ((effect)=(condition))

interaction
  (make (a: <type> s) flow)
implementing
  Behavior
is
  ((a) = ((new(a)) default (previous(a))))
```

Listing 9. Section of the standard LIDL library regarding Behaviors

A.4 Activation

```
interaction
  (active)
implementing
  activation out
is
  'native'
```

A.5 Boolean operators

What is a boolean operator ? Very generally, it is something that takes some parameters and returns a boolean. This is why we define de `Boolean` interface to be equivalent to the `boolean out` interface. This means that interactions implementing the `Boolean` interface return a flow of `boolean` which depends on their parameters.

The LIDL standard library defines some useful basic boolean operators, as shown on Listing 10.

```
interface
  Boolean
is
  boolean out

interaction
  (not (a:boolean in))
implementing
  Boolean
is
  'native'

interaction
  ((a:boolean in) and (b:boolean in))
implementing
  Boolean
is
  'native'
```

```
interaction
  ((a:boolean in) or (b:boolean in))
implementing
  Boolean
is
  (not((not(a))and(not(b))))

interaction
  ((a:boolean in) xor (b:boolean in))
implementing
  Boolean
is
  (((a)or(b))and(not((a)and(b))))

interaction
  ((a:boolean in) nand (b:boolean in))
implementing
  Boolean
is
  (not((a)and(b))

interaction
  ((a:boolean in) nor (b:boolean in))
implementing
  Boolean
is
  (not((a)or(b))

interaction
  ((a:boolean in) implies (b:boolean in))
implementing
  Boolean
is
  (not((a)and(not(b))))

interaction
  ((a:boolean in) is equivalent to (b:boolean in))
implementing
  Boolean
is
  (not((a)xor(b)))

interaction
  ((a:boolean in) inhibited by (b:boolean in))
implementing
  Boolean
is
  ((a)and(not(b)))

interaction
  ((a:<type>in) == (b:<type>in))
implementing
  Boolean
is
  `native`

interaction
  ((a:<type>in) != (b:<type>in))
implementing
  Boolean
is
  (not((a)==(b)))
```

Listing 10. The Number expressions definitions of the standard LIDL library

A.6 Numeric operators

What is a numeric operator ? Very generally, it is something that takes some parameters and returns a number. This is why we define de `Numeric` interface to be equivalent to the `number out` interface. This means that interactions implementing the `Numeric` interface return a flow of `number` which depends on their parameters.

The LIDL standard library defines some useful basic numeric operators, as shown on Listing 11.

```
interface
  Numeric
is
  number out

// opposite
interaction
  (- (a:number in))
implementing
  Numeric
is
  `native`

// sum
interaction
  ((a:number in) + (b:number in))
implementing
  Numeric
is
  `native`

// difference
interaction
  ((a:number in) - (b:number in))
implementing
  Numeric
is
  ((a)+(-(b)))

// inverse
interaction
  (inverse (a:number in))
implementing
  Numeric
is
  `native`

// multiplication
interaction
  ((a:number in) * (b:number in))
implementing
  Numeric
is
  `native`

// division
interaction
  ((a:number in) / (b:number in))
implementing
  Numeric
is
  ((a)*(inverse(b)))
```

Listing 11. The Number expressions definitions of the standard LIDL library

A.7 Widgets

The LIDL standard library defines some widgets, as shown on Listing 12.

```
////////////////////////////////////////////
interface
  Label
is
  {
    value : text out
  }

interaction
  (Label (status: activation in)
  with title (theTitle: text in))
implementing
  Label
is
  (bind(this) : (when (status) : (all
      ((this.value)=(theTitle)))
  )))

////////////////////////////////////////////
interface
  Gauge
is
  {
    value : number out
  }

interaction
  (Gauge (status: activation in)
  with value (theValue: number in))
implementing
  Gauge
is
  (bind(this) : (when (status) : (all
    ((this.value)=(theValue))
  )))

////////////////////////////////////////////
interface
  Button
is
  {
    value: text out,
    click : activation in
  }

interaction
  (Button (status: activation in)
  with title (theTitle: text in)
  trigerring (onClick: activation out))
implementing
  Button
is
  (bind(this) : (when (status) : (all
      ((this.value)=(theTitle))
      ((onClick)=(this.click))
  )))



////////////////////////////////////////////
interface
  SegmentedSwitch
is
  {
    choices : [text] out,
    selection : number in
  }

interaction
  (SegmentedSwitch (status: activation in)
  with choices (theChoices: [text] in)
  selecting (theSelection: text out))
implementing
  SegmentedSwitch
is
  (bind(this) : (when (status) : (all
      ((this.choices)=(theChoices))
      ((theSelection)=
        ((theChoices)[(this.selection)]))
  )))

////////////////////////////////////////////
interface
  Slider
is
  {
    value : number out
    selection : number in,
  }

interaction
  (Slider (status: activation in)
  with value (theValue: number in)
  selecting (theSelection: number out))
implementing
  Slider
is
  (bind(this) : (when (status) : (all
      ((this.value)=(theValue))
      ((theSelection)=(this.selection))
  )))


////////////////////////////////////////////
```

Listing 12. A standard widget library

Here we put the LIDL source code of the case study and probably the promela code and Event-B code etc.

Acknowledgments

Acknowledgments, if needed.

References

[1] Bureau d'Enquêtes et d'Analyses pour la sécurité de l'Aviation Civile. Rapport du Vol AF 447 du 1^{er} Juin 2009. http://www.bea.aero/fr/enquetes/vol.af.447/rapport.final.fr.php.

[2] DO178C. Software Considerations in Airborn Systems and Equipment Certification, release C. 2012. RTCA,Inc.

[3] ARINC 661. Cockpit Display Systems Interfaces to User Systems. http://www.aviation-ia.com/aeec/projects/cds/index.html. Airlines Electronic Engineering Committee.

[4] http://www.esterel-technologies.com/products/scade-display.