The LIDL Interaction Description Language

Vincent Lecrubier
ONERA DTIM/LAPS
Toulouse, France
lecrubier@onera.fr

Bruno d'Ausbourg
ONERA DTIM/LAPS
Toulouse, France
ausbourg@onera.fr

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

ABSTRACT

This paper describes LIDL, a language dedicated to the specification of interactive systems. LIDL is based on the idea that most programming languages are useful to specify computations, but are not adequate when it comes to specifying interactions. We first introduce the context and the need for new paradigms for interactive systems specification. Then we describe the basic concepts of LIDL, such as Interfaces, Data activation, Interactions, and LIDL program structure. Some uses of LIDL programs such as verification and code generation are then explained. Finally, a boiling water nuclear reactor user interface is partially developed using LIDL, as an example use case.

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

Author Keywords

Human-Machine Interfaces; Domain-Specific Language; Interactive Systems; Formal Language; Critical Systems

INTRODUCTION

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.

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 [1], 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.

Because of the problem stated in Figure 1, 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 [6] 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 [4]. 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 [23], 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.

THE LIDL LANGUAGE

It seems to us that the current state of the art provides no complete solution to the need described in the previous section. The aim of LIDL is to provide a language and tools to deal with this set of problems.

Informal presentation

While most programming languages focus on the description of *computations*, the main idea behind LIDL is to describe

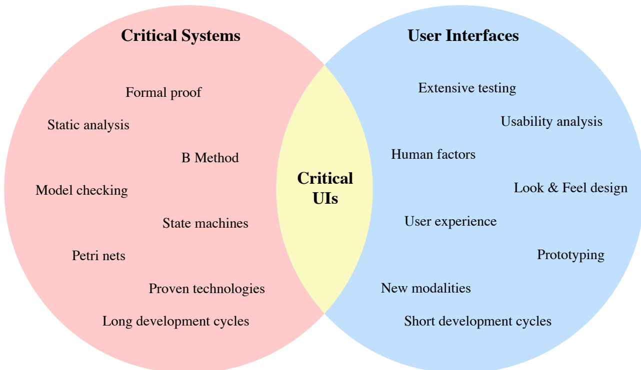

Figure 1. Critial UIs development is as the intersection of two clashing domains.

interactions. This is quite a paradigm shift in the sense that many experienced programmers will at first be surprised by the language semantics. However, we argue that LIDL provides an easier way to specify interactive systems, since its main concepts (interfaces and interactions) are more relevant to the field of interactive systems than other programming languages concepts (objects, functions, algorithms...)

LIDL programs are defined in a declarative manner, and represent interactive systems whose execution is synchronous.

Interfaces

Typed programming languages rely on data types to check the composability of functions and operations. This is convenient when the goal is to describe *computations*. But this is not enough when we try to describe *interactions*. When composing interactions, another very important aspect which is rarely stated is the *direction* data goes in.

As an answer to that matter, an important feature of LIDL is the notion of *interface*. An interface is the combination of two orthogonal aspects: the data type and the data direction.

The notion of data type is well known to most programmers. The notion of data direction is also quite easy to understand: the data can either go *in* or go *out*. The notion of interface is hence quite easy to catch, here are a few example of basic interfaces: `Number in`, `Boolean out`, `Text in`...

The same way compound data types exist, one can express compound interfaces. The syntax to specify compound interfaces is inspired by the Javascript Object Notation (JSON) [14]. Listing 1 shows an example compound interface defined in LIDL.

```
interface Example is
  {
    redSquares       : Square in,
    greenPentagons   : Pentagon in,
    yellowTriangles  : Triangle out,
    blueCylinders    : Cylinder out
  }
```

Listing 1. LIDL definition of the example interface

Metaphorically, interfaces can be seen as the specification of pipes of specific shapes that allow objects to go in specific directions. Figure 2 shows a way to visualise the example interface of Listing 1.

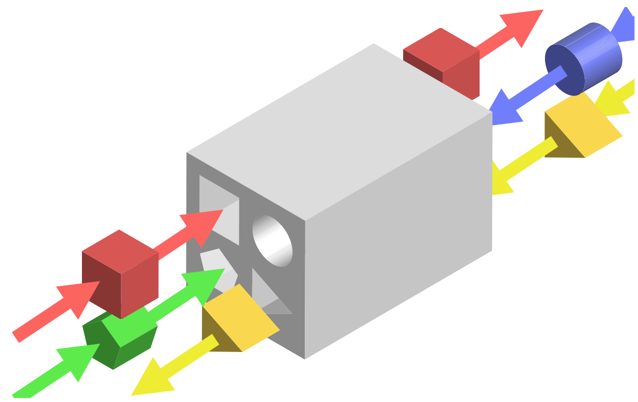

Figure 2. A metaphor of interfaces as pipes that allow specific data types to flow in specific directions

Every interface has a conjugate interface, which has the same data types, but opposite directions. Two interactive systems can only connect if their interfaces are conjugate. This is the consequence of the natural intuition that the output of an entity is the input of another one.

Interfaces are central in LIDL as they have the same role as data types in typed programming languages.

Data activation

Interactive systems rely on two different paradigms: flow-based representations and event-based representations.

Flow-based representations maps well to systems whose data is defined on *continuous* time intervals, such as the pressure inside a reactor. Examples of flow-based representations include Lustre [17], Scade...

On the other hand, event-based systems maps well to systems whose data is defined on discrete time sets, such as clicks on a button. Examples of event-based representations include most User Interface (UI) Toolkits such as Java Swing, Qt...

Several approaches tried to bridge the gap between flow and event representations [2]. However most approaches are biased toward one paradigm or the other. Interestingly, some approaches treat input and output differently, for example by only allowing discrete inputs (events) and continuous output (status). Figure 3 presents the positioning of different academic approaches regarding this aspect. Shown approaches include [17], [12], [11], [13], [18], [3], [21], [20] and LIDL.

Restriction to a paradigm or the other often prevents natural description of interactive systems, which generally are best described using a mix of both. LIDL proposes a simple way to unify and mix the two paradigms: the notion of data activation.

The notion of data activation is latent in industrial art. Most languages exhibit constructions such as the *null* value, the *maybe* monad, callback functions, listeners, observers, signal slots...

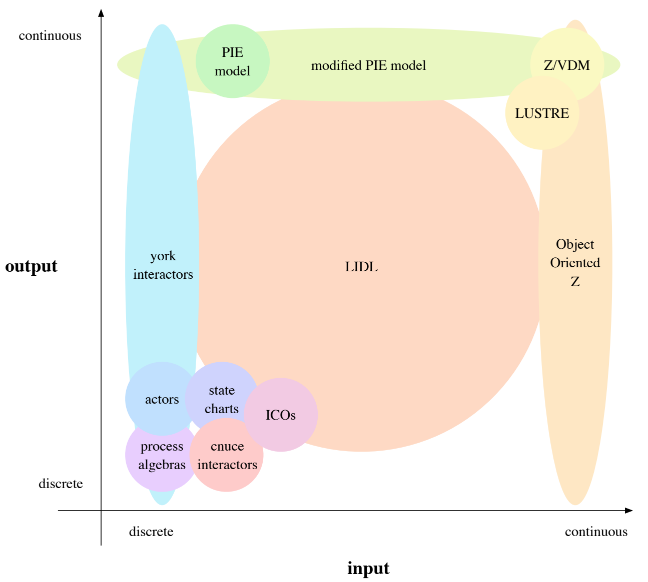

Figure 3. Positions of different academic approaches in the flow vs event space.

In the context of interactive systems, all these constructions boil down to one unique concept: identify the presence of a piece of data, most of the time a message that has to be received or sent. This is exactly what the data activation feature of LIDL does.

Without exception, every piece of data in a LIDL program integrates a notion of activation. The implementation is really simple: *all* LIDL data types are extended with the *inactive* value noted $\perp$. For example, the following table shows example values for the basic data types of LIDL:

Type	Example values
`Activation`	$\perp, \top$
`Boolean`	$\perp$, *true*, *false*
`Number`	$\perp$, 0, 1, 3.14159
`Text`	$\perp$, "*Foo*", "*Bar*", "*Baz*"

Very simplistically, a flow is represented in LIDL by a piece of data which is almost always active. For example, through an execution, the pressure in a reactor would have the following trace: $\{451, 453, 452, 450, 454, ...\}$. On the other hand an event is represented by a piece of data which is almost always inactive. For example, through an execution, clicks on a button would have the following trace: $\{\perp, \perp, \perp, \perp, click, \perp, ...\}$

The notion of activation does not break composability. Here is a compound data type expressed in LIDL : `{x:Number,y:Number}`. This data type is a labelled product data type, similar to a `struct` of the C language. Here are a few example of values of this type: `{x:3,y:2}`, `{x:⊥,y:3}`, $\perp$.

Interactions

LIDL is a language to describe interactions. The interaction language has a simple syntax, which uses a lot of parentheses. An interaction is a phrase between parentheses, and it composes trivially. Listing 2 shows an example interaction expression, while Figure 4 shows its structure.

```
(when(not(powered))then(turn(light)red))
```

Listing 2. An example interaction expression

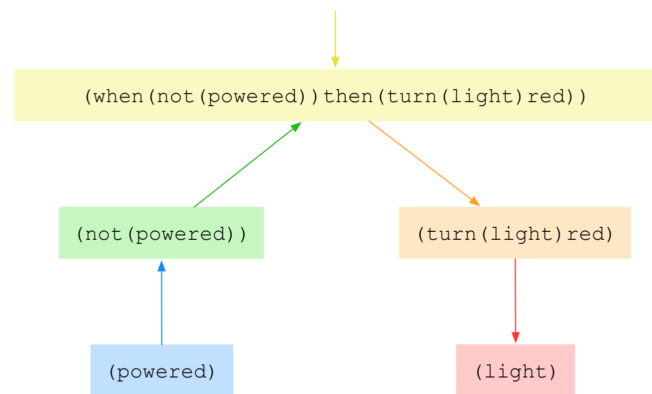

Figure 4. The structure of the example expression. Arrows represent the data flow direction

The semantics of interactions is the most challenging part of LIDL for newcomers, because it is the most disruptive part of the language, since it leverages interfaces and the notion of activation.

Each interaction (i.e. each pair of parentheses, i.e. each block in Figure 4) is attributed a value at each execution step. Depending of the data direction of the interface of the interaction, this value can be defined by the interaction itself (**`out`**) or by an external interaction (**`in`**).

Interactions that comply to the **`out`** interface behave like functions. They output a value, based on their arguments. For example `(not(powered))` receives a boolean `(powered)` and outputs a boolean that is the negation of `(powered)`. This is explained in the following table, which should be easy to understand, with parameters on the left column, and results on the right:

`(powered)`	`(not(powered))`
true	*false*
false	*true*
$\perp$	$\perp$

Interactions that comply to the **`in`** interface behave the opposite way, which is **completely foreign** to programmers. Imagine a function that does not *return* a value based on the arguments it receives, but that *receive* a value, and returns values to its arguments. For example `(turn(light)red)` receives an activation, and outputs a colour `light` which is red when the interaction is active, or $\perp$ the rest of the time. This is summarised in the following table, which will look **unfamiliar** to most programmers, with parameters on the left column, and computation result on the right:

`(turn(light)red)`	`(light)`
$\top$	$\{red : 255, green : 0, blue : 0\}$
$\perp$	$\perp$

The main advantage of LIDL interaction expressions is that they are very general. Many first-class constructions of other programming languages can be represented as LIDL interactions. As an example, the following table quickly summarises the semantics of the `()=()` interaction. Note that this interaction complies with the **in** interface, indeed, its behaviour consists in sending to the left-hand-side the value it receives on the right-hand-side, *only* when the interaction is active.

`((x)=(y))`	`(y)`	`(x)`
$\top$	5	5
$\top$	$\bot$	$\bot$
$\bot$	5	$\bot$
$\bot$	$\bot$	$\bot$

LIDL programs structure
LIDL programs structure is similar to functional programs structure. Functional programs are represented as a function. A LIDL program is nothing more than an interaction.

The same way that functional programming languages use function signatures to define functions, LIDL use interaction signatures. Since LIDL uses interfaces instead of data types, interaction signatures are described in terms of interfaces.

As an example, here is the signature of the interaction `when ()then()` which is instantiated as the root of the example interaction expression of Listing 2:

```
( when (condition: Boolean in)
  then (effect: Activation out)
): Activation in
```

Listing 3. The signature of an interaction

The same way that functional programming languages allow to define functions by specifying a signature and the expression it reduces to, LIDL allow to define interactions by specifying a signature and the expression it reduces to.

As an example, here is the definition of the interaction `turn ()red` which is used in our example expression of Figure 4:

```
interaction
  (turn (thing: Color out) red): Activation in
is
  ((thing)=({red:(255),green:(0),blue:(0)}))
```

Listing 4. Complete LIDL definition of an interaction

Finally, the same way a functional programmer composes functions in order to make more complex functions, a LIDL programmer composes simple interactions in order to make more complex interactions, ending with a final complex interaction: the LIDL program itself.

USE OF LIDL PROGRAMS
LIDL is only a convenient textual way to describe Directed Acyclic Graph (DAG) structures. Indeed, the compiler first expands interactions into base interactions, using definitions. Then it assigns data flow directions using interfaces definitions. This results in a DAG which express the transition function of a state machine. As an example, Figure 5 shows the graph associated with our example expression.

It is really important to notice that the graph shown in Figure 5 is really nothing more than a graph ordering of the graph shown in Figure 4, with data dependency as the ordering relationship. Data dependency is easily inferred from the interfaces.

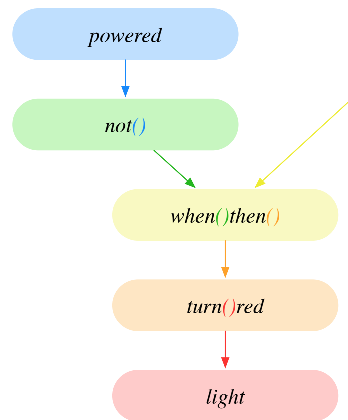

Figure 5. The example expression compiled into a directed acyclic graph

This graph representation is in fact Single Static Assignment (SSA) form [9] of the executable implementing the specified interaction. This form allows different uses such as optimisations, verification, proofs and code generation.

Optimisation
Optimisation can be performed by analysing the graph representation, and generating different execution schemes depending on the requested inputs and outputs, using techniques such as push (data driven evaluation) and pull (demand driven evaluation) as applied to functional reactive programming in [15].

Verification
Verification and proof can be performed by transforming intermediate representation into state machines. The graph representation exactly describes the transition function of such a system, while the state vector is easily derived. It is important to note that the only way for data to persist from one execution step to the next is to be part of a `previous()` interaction. Hence, the state vector is *exactly* the set of interactions which are included in `previous()` interactions. Finally, the generated system has a structure which is very similar to systems generated by other synchronous data flow programming languages such as Scade or Lustre [17]. This potentially allows to leverage the verification tools that have been developed and used for these languages.

Code generation
Code generation has two main objectives: prototype code generation, and production code generation. Both are similar in nature, and are made relatively easy thanks to the intermediate representation. The target languages only have to provide a few features: compound data types, functions, and data types corresponding to LIDL basic data types. At the moment, code generation tools are being developed for two languages: The first is Javascript, in order to enable quick

prototyping in a web app, and even some sort of Read-Eval-Print-Loop similar to the one available online for the Elm functional reactive programming language [10]. The other target language is C, as it is probably the most common language in use for critical systems.

Human models and automatic testing

We have seen that LIDL can be used to specify interactions to be performed by computers. It is also a surprisingly convenient way to model interactions to be performed by human agents.

The high abstraction capabilities of LIDL coupled with its close-to-natural-language syntax allows to specify human interactions associated with a system in a very formal way, while remaining similar to a user manual. LIDL descriptions of human interactions are interesting because they bridge the gap between *task models* and *user manuals* [8], being a generalisation of both.

Typically, LIDL developers would code two things: computer-side interactions (e.g a widget behaviour), and their human-side counterparts (e.g. how to use a widget). This approach is similar to test driven development, applied to interactive systems. This formal specification of human interactions enables automatic testing, by executing a system composed of the computer-side interactions on one side, and the human-side interaction on the other, in an approach similar to [5].

Furthermore, LIDL makes it easy to take into account and model human errors and non deterministic behaviour such as those detailed in [7] and [19]. This allows to test interactive systems even more completely, by simulating the consequences of human errors. Listing 5 shows an example human-side interaction that details how to click on a button, taking into account one error type: omission. The `either()()` interaction represents a non-deterministic choice.

```
interaction
  (click on (theButton: Button)): Activation in
is
  ( either
    ((theButton.click)=(active))    // Nominal
    (nothing)                       // Omission !
  )
```

Listing 5. LIDL definition of a potentially faulty human interaction

USE CASE

In this section, we will use LIDL to describe the Boiling Water Reactor (BWR) use case. For the sake of simplicity, we will limit ourselves to an abstract interface as described in [22]. However, LIDL is not restricted to the specification of abstract user interfaces.

LIDL puts an emphasis on reusability. In the use case, this means that we will take advantage of the similarities between components in order to limit the bulk of code. Figure 6 shows common elements in coloured frames, these common elements will be coded as reusable components.

LIDL implementation of a basic component

Figure 6. A screenshot of the BWR simulator with some common elements outlined in common colors

To get started, let's look at the implementation of a simple abstract slider, which could be part of a standard abstract widget library for LIDL.

Listing 6 shows the interface that this abstract slider complies to. The abstract slider outputs two things to the user: The value of the slider, concretely implemented by the position of the cursor, and the slider range, concretely implemented by the labels at each end of the slider. The abstract slider has one input from the user: The position that the user wants the slider to be at.

```
interface Slider is
  {
    value: Number out,
    range: {min: Number, max: Number} out,
    selection: Number in
  }
```

Listing 6. The interface of an abstract slider

We could define many interactions that implement this interface. Listing 7 presents one of them. This implementation follows these arbitrary design choices:

- It takes an `enabled` argument that specifies if the slider is enabled or not.
- It takes two arguments to specify the range of the slider.
- In case no value is provided for the slider position, it will initialise as the lower bound of the range.
- It sets the value of the argument `theSelection` when changed by the user, or when the range is changed so that it becomes incompatible with the previous value of the slider.
- It take an argument `constrainedPosition` that allows to programatically set the value of the slider, overriding user input.

Several interactions are used in order to define the slider interaction. For example, note the use of the `()fallbackto() fallbackto...` interaction (lines 16-19). This interaction uses the activation of its arguments, and picks the first argument which is active.

Another important point to notice is the argument named `theSelection` (line 5). Since it is an **out**, it will not be read in order to compute a result. In fact, it will be written to, i.e. a value will be sent to it. This is unlike other arguments that are **in**, which have roles similar to arguments programmers are used to.

By looking at this implementation of the slider, it is really easy to notice that it is a stateful component. Indeed we can see a `previous()` interaction (line 18). The interaction inside the `previous()` is `currentValue`, so `currentValue` is the state variable.

```
1  interaction
2    ( slider (enabled: Activation in)
3      between (min:Number in) and (max:Number in)
4      constrained to (constrainedPosition: Number in)
5      selecting (theSelection: Number out)
6    ): Slider
7  is
8    ((when (enabled)
9      then ({
10       value:(currentValue),
11       range:({min:(min),max:(max)}),
12       selection:(userInput)
13     }))
14    behaviour
15     ((current value)=
16       (((constrainedPosition)
17         fallback to (userInput)
18         fallback to (previous(currentValue))
19         fallback to (min))
20       kept between (min) and (max))
21     ))
22 with
23   interaction (currentValue):Number ref
24   interaction (userInput):Number ref
```

Listing 7. The definition of an abstract slider interaction

Listing 8 shows an example use of the slider defined in Listing 7. This instance will always be enabled, because the `enabled` argument is set to the constant `active`. Since the constrained value is set to `inactive`, this instance will allow the user to select a number in the constant range $[0, 2000]$, and the value selected by the user will be sent to a variable named `myValue`.

```
(slider (active) between (0) and (2000)
  constrained to (inactive) selecting (myValue))
```

Listing 8. An instance of the abstract slider

LIDL implementation of a compound component

We will describe the components framed in green on Figure 6. These components, that we will call "complex sliders", are composed of:

- A label indicating the purpose of the slider to the user
- A slider allowing the user to select a value. The slider is the one defined in the previous section
- A toggle button to switch between manual and auto modes
- A label indicating the value and units of the selection

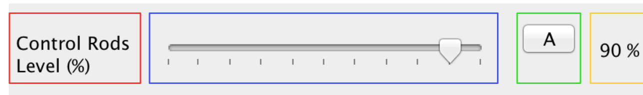

Figure 7. A screenshot of a complex slider component

Figure 7 shows the concrete implementation of this complex slider, and Listing 9 shows its LIDL interface. Note that it reuses the `Slider` interface defined in the previous section, as well as other interfaces.

```
interface ComplexSlider is
  {
    title: Label,
    slider: Slider,
    toggle: ToggleButton,
    value: Label
  }
```

Listing 9. The interface of the complex slider

Listing 10 shows an implementation of this complex slider.

```
interaction (
  complex slider
  named (title: Text in)
  between (min:Number in) and (max:Number in)
   (units:Text in)
  constrained to (constrainedPosition: Number in)
  selecting (theSelection: Number out)
  requesting (mode: Activation out) automation
  ):ComplexSlider
is
  ({
    title: (
      label (active)
      displaying (title)),
    slider: (
      slider (active)
      between (min) and (max)
      constrained to (constrainedPosition)
      selecting (theSelection)),
    toggle: (
      toggle (active)
      pushed (when(constrainedPosition))
      displaying ("A")
      toggling (mode)),
    value: (
      label (active)
      displaying ((theSelection) " " (units)) )
  })
```

Listing 10. Definition of a complex slider interaction

Listing 11 shows an example instance of this complex slider, corresponding to the concrete implementation depicted in Figure 7.

```
( complex slider
  named ("Control Rods Level")
  between (0) and (100) ("%")
  constrained to (controlRodsAutoValue)
  selecting (controlRodsLevel)
  requesting (controlRodsAutoMode) automation
)
```

Listing 11. An instance of the complex slider interaction

CONCLUSION

This paper presented a quick overview of LIDL, a language dedicated to the description of interactions, and a use case. The use case showed that LIDL allows to specify safe complex behaviour. In particular, it is noteworthy that, as compared to other approaches, the LIDL way of thinking as two consequences:

- Removing duplicate or boilerplate code as seen in other languages, such as getter/setters and observer pattern functions. This is noticeable by the relatively small size of LIDL programs.
- Forcing designers into thinking about the actual interaction, enforcing to explicitly define aspects that are usually implicit or merged into objects whose semantics are not clear. This is noticeable in the slider example (Listing 7), where an explicit distinction is made between the user input and the slider current value. This explicit distinction allows to have a sane behaviour, even when the slider range is dynamically changed, while keeping the code simple.

LIDL is only a language. Architectural concepts that fit with LIDL are being developed, but not detailled in this paper. The architectural ideas behind LIDL converge with those recently presented in [16] and similar approaches around unidirectional data flow. A general framework for the specification of abstraction levels of interactive systems inspired by [24] is being developed in parallel with LIDL.

REFERENCES

1. 2012. DO178C. Software Consideration in Airborn Systems and Equipment Certification, release C. (2012). RTCA,Inc.
2. Gregory D Abowd and Alan J Dix. 1994. Integrating status and event phenomena in formal specifications of interactive systems. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering* 22 (December 1994), 23.
3. G. A. Agha. 1986. Actors: a model of concurrent computations in distributed systems. *MIT Press* (1986).
4. ARINC 2012. *Specification 661* (supplement 5, draft 1 ed.). ARINC. http://www.aviation-ia.com/aeec/projects/cds/index.html
5. Eric Barboni, Jean-François Ladry, David Navarre, Philippe Palanque, and Marco Winckler. 2010. Beyond Modelling: An Integrated Environment Supporting Co-execution of Tasks and Systems Models. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '10)*. ACM, New York, NY, USA, 165–174. DOI: http://dx.doi.org/10.1145/1822018.1822043
6. BEA. 2012. *Rapport final sur l'accident survenu le 1er juin 2009 à l'Airbus A330-203 immatriculé F-GZCP exploité par Air France, vol AF 447 Rio de Janeiro - Paris*. Technical Report. Direction Générale de l'Aviation Civile. http://www.bea.aero/docspa/2009/f-cp090601/pdf/f-cp090601.pdf.
7. M.L. Bolton and E.J. Bass. 2011. Evaluating human-automation interaction using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking. In *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*. 1788 –1794. DOI: http://dx.doi.org/10.1109/ICSMC.2011.6083931
8. Judy Bowen and Steve Reeves. 2012. Modelling User Manuals of Modal Medical Devices and Learning from the Experience. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '12)*. ACM, New York, NY, USA, 121–130. DOI: http://dx.doi.org/10.1145/2305484.2305505
9. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct 1991), 451–490. http://doi.acm.org/10.1145/115372.115320
10. Evan Czaplicki. 2012. Elm: Concurrent FRP for Functional GUIs. (2012).
11. A. Dix and C. Runciman. 1985. Abstract models of interactive systems. In *Proceedings of the HCI'85 Conference on People and Computers: Designing the Interface*. 13–22.
12. Alan John Dix. 1991. *Formal methods for interactive systems*. Academic Press.
13. D.J. Duke and M.D. Harrison. 1993. Abstract Interaction Objects. *Computer Graphics Forum* 12, 3 (1993), 25–36. DOI: http://dx.doi.org/10.1111/1467-8659.1230025
14. ECMA. 2013. The JSON Data Interchange Format. (October 2013). http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
15. Conal M. Elliott. 2009. Push-pull Functional Reactive Programming. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, New York, NY, USA, 25–36. DOI: http://dx.doi.org/10.1145/1596638.1596643
16. Facebook. 2013. React - a JavaScript library for building user interfaces. (2013). http://facebook.github.io/react/
17. Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous dataflow programming language Lustre. In *Proceedings of IEEE (79)*. 1305–1320.
18. D. Harel. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8, 3 (1987), 231–274.
19. Célia Martinie and Philipe Palanque. 2014. Fine Grain Modeling of Task Deviations for Assessing Qualitatively

the Impact of Both System Failures and Human Error on Operator Performance. In *2014 AAAI Spring Symposium Series*.

20. David Navarre, Philippe Palanque, Jean-Francois Ladry, and Eric Barboni. 2009. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Trans. Comput.-Hum. Interact.* 16, Article 18 (November 2009), 56 pages. Issue 4. DOI: http://dx.doi.org/10.1145/1614390.1614393

21. F. Paternò and G. Faconti. 1992. On the use of LOTOS to describe graphical interaction. In *Proceedings of the HCI'92 Conference on People and Computers*. 155–173.

22. Fabio Paterno, Carmen Santoro, and Lucio Davide Spano. 2009. MARIA: A Universal, Declarative, Multiple Abstraction-level Language for Service-oriented Applications in Ubiquitous Environments. *ACM Trans. Comput.-Hum. Interact.* 16, 4, Article 19 (Nov. 2009), 30 pages. DOI: http://dx.doi.org/10.1145/1614390.1614394

23. Esterel Technologies. 2011. Scade Display. (2011). http://www.esterel-technologies.com/products/scade-display

24. Pamela Zave and Jennifer Rexford. 2012. The Geomorphic View of Networking: A Network Model and Its Uses. In *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing (MW4NG '12)*. ACM, New York, NY, USA, Article 1, 6 pages. DOI: http://dx.doi.org/10.1145/2405178.2405179