A formal language for critical embedded user interfaces

Vincent Lecrubier
ONERA - DTIM
2 Avenue Edouard Belin
31400, Toulouse, France
Vincent.Lecrubier@onera.fr

ABSTRACT

The LIL Interaction Language (LIL) is a domain-specific language (DSL) which aims at enhancing the design process of critical user interfaces (UIs). LIL is a human-readable textual language, and has formal semantics. The ambition of LIL is to become a *lingua franca* between stakeholders of the critical UI specification process. This paper describes LIL positioning amongst other similar approaches, and the main concepts of LIL. We will also present ongoing work around the LIL framework, which leverage the expressiveness of the language in order to offer tools for verification, validation and code generation.

Author Keywords

Human Machine Interface; User Interface; Design; Specification; Verification; Validation; Formal Language; Domain-Specific Language.

ACM Classification Keywords

H.5.2. User Interfaces: Evaluation/methodologyabstr

INTRODUCTION

Critical embedded UIs conception is a discipline which lies at the limits of two software engineering domains: embedded software development and user interface development. As a consequence, critical embedded user interfaces must comply with a set of conflicting constraints.

While embedded software makes robustness a priority, UIs must be flexible and configurable. While embedded systems have limited ressources, UIs must be complete and integrate lots of functionalities. While critical software have strong timing constraints, UIs must be user friendly and let the user interact at his own pace. While critical systems are often real-time, UIs are mostly event-driven.

This set of conflicting constraints is a strong limitation for critical UIs, and makes their development particularly costly and time-consuming. A good critical UI development process should take into account the methods and tools in use for both of these domains.

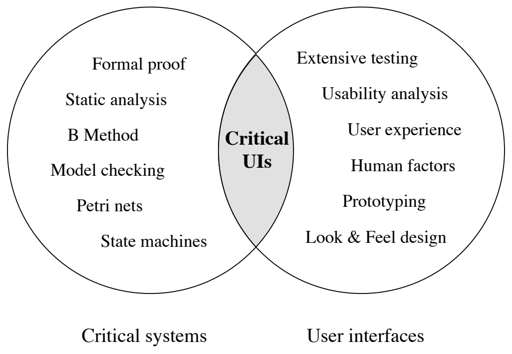

Figure 1. Two clashing domains

CONTEXT

This project tends to make a constructive use of formal methods in critical UIs development. As so, it relies on previous work on this subject. Much work was already performed on formal modeling of software aspects of UIs by suggesting two main approaches.

The first one is based on proof systems. For example, Z and VDM were used to define atomic structures of interactions ([?],[?]) and HOL (High Order Logic Theorem Prover) was used to check user interface specification ([?]). B system was also used for the proof of incremental specifications ([?],[?]).

The second one is based on the evaluation of logical properties on state transition systems. This technique was used, for example, to verify formally with SMV some properties of interactive systems ([?]). Model checking was used in ([?],[?]) where the user and the system are modeled by object Petri nets (ICOs). Model checking was also coupled with static analysis of program codes in [?] or [?] to extract and abstract a formal model of an interactive system and to check various properties. Modeling and analyzing of human-automation interaction has also been demonstrated in [?], [?] and [?].

Figure 2 describe the positioning of LIL amongst some other methods and tools currently in use or development.

GOALS

This project aims at inverting the approach in [?] or [?] by defining a language which would allow to formally describe and model the behavior of UIs. The model perimeter is centered around the *dialog controller* of the UI Arch Model from [?], but allows to progressively expand the model into both sides of the Arch Model.

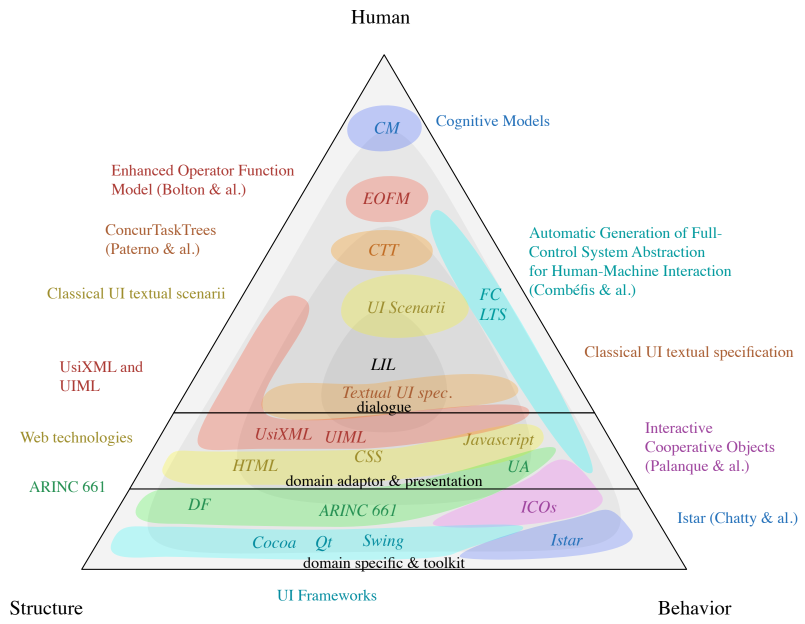
Human
CM
Cognitive Models
Enhanced Operator Function Model (Bolton & al.)
EOFM
ConcurTaskTrees (Paterno & al.)
CTT
Automatic Generation of Full-Control System Abstraction for Human-Machine Interaction (Combéfis & al.)
Classical UI textual scenarii
UI Scenarii
FC
LTS
UsiXML and UIML
LIL
Classical UI textual specification
Textual UI spec.
dialogue
Web technologies
UsiXML
UIML
Javascript
Interactive Cooperative Objects (Palanque & al.)
CSS
HTML
UA
ARINC 661
domain adaptor & presentation
DF
ARINC 661
ICOs
Istar (Chatty & al.)
Cocoa
Qt
Swing
Istar
domain specific & toolkit
Structure
UI Frameworks
Behavior

Figure 2. LIL (in light grey) positioning amongst other UI development methods and tools (in colors), analyzed around 3 main axes: structural aspects, behavioral aspects and human aspects. Methods which describe an aspect with a high level of detail are found close to the edges of the figure, while higher level, more abstract methods are found closer to the center of the figure. For reference, the Arch model [?] is represented with black frames.

This project has similarities with the I* approach in [?]. Being driven by a similar set of constraints, we share the idea that an ideal development process should rely on a unified framework which supports modularity and multiple levels of granularity.

One of the main goals of LIL is to be useful in the earliest phases of the UI specification process, which implies a strong emphasis on abstract UI modeling as described in [?]. LIL also aims at being a common language for all stakeholders related to the UI conception process, and this implies that it has a simple and clean syntax, which seems a natural conclusion to the ideas expressed in [?].

LIL promotes best practices in critical system development, such as Modularity, Reusability, Error propagation prevention, Bottleneck identification and Traceability.

MAIN CONCEPTS OF LIL

LIL tries to model in a consistent way all elements involved in UI design and execution:

- *Actors*: Elements in interaction with the UI. A classical scenario involves two actors: a human and a system. However, LIL support more complex scenarios.
- *Data*: Information that passes through the UI. Modeling of data in LIL is limited to structured data types definitions and specification of constraints.
- *Interactors*: Building blocks of the UI. A UI is made of a hierarchical composition of interactors.

Interactors

Interactors are modeled in detail in LIL, they are formally equivalent to automata, and they are are specified using:

- *Components*: Smaller interactors that compose an interactor. For example, a combo box is made of a text box, an expand button, and a list which expands when needed. The composition of interactors describes the structural aspects of UIs.
- *Signals*: The pipes that convey data inside and outside of the interactor. The notion of signal is one of the main contribution of the LIL approach.
- *Behaviors*: A set of behavior expressions express how the interactor should deal with incoming signals and send outgoing signals.

Signals

Signals bridge the gap between structural aspects and behavioral aspects of a UI. LIL signals have the following specificities:

- Signals can express communication inside of interactors, between interactors, but also between actors and interactors. For example, a label is an interactor which sends signals to the user. On the other hand, a joystick input is an interactor which receives signals from the user, and transmits them to other interactors.
- LIL accepts two types of signals: events, and flows. Flows model signals that have a continuous time domain, *e.g.* a joystick input, or a progress bar. On the other hand, events model signals that have discrete time domains, behavior a “Beep” sound or a user click.

EXPLOITATION

Verification

UIs specified in LIL are formally described as asynchronous interleaving product of automata. Model checking tools such

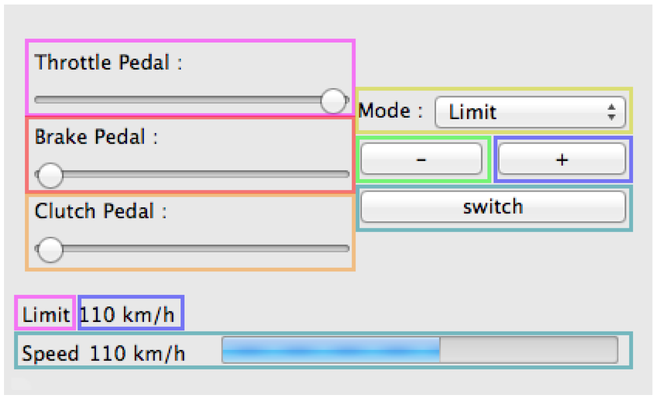

Figure 3. A WIMP concretization of the example application..

as SPIN [?] can thus be applied to LIL models in order to check properties such as:

- *Correct behavior*: Sets of safety and liveness properties to check that the UI behaves as expected.
- *Performance*: If additional performance information is available, Worse Case Execution Times (WCET) of UI related tasks , bandwidth consumption between subsystems (*e.g.* between User Applications (UAs) and Cockpit Display System (CDS), in the ARINC 661 context), memory footprint of the UI could be checked.
- *Task model fitness*: Fitness of the actual UI relative to a task model, (*i.e.* check if the UI is able to support every possible scenario of a Task Model)

Code generation

Prototyping is an interesting use of the code generation capabilities, since it allows to get user feedback on aspects of a UI during the earliest stages of the conception process. Tools enabling semi-automatic generation of prototypes are under development.

Another application is code generation for the dialog controller components[?] of UAs. Controlled code generation preserves properties checked on the LIL specification. This guarantees that the final UI will exhibit the behavior specified in LIL.

EXAMPLE APPLICATION

The example UI described in Listing 1 is a car speed controller/limiter. It is modeled as a single, non-composed interactor, which enables interaction between two actors: the driver, and the car. This application is a good example because it is not a Windows, Icons, Menus, Pointer (WIMP) UI, but it can also be concretized as such. It also mixes events signals and flow signals in every possible way (signals triggering events, events modifying signals).

The completeness requirement for abstract UIs [?] states that an abstract UI may give rise to many concrete UIs. Indeed, Figure 4 shows the final, material UI, while Figure 3 represents another concretization of the abstract UI based on the WIMP paradigm.

ACKNOWLEDGMENTS

This research is funded by ONERA. I want to thank my colleagues at ONERA - DTIM, and in other labs in Toulouse for the discussions about and feedback on this research.

```
mode data:
   symbol in {"Off","Limit","Control"}

speedController interactor:
   driver : human actor
   car : system actor

   step : number constant
   minTarget : number constant
   maxTarget : number constant

   topLine : text or number flow to driver
   topLine = if modeCar == "Off" then "" else modeCar

   bottomLine : text or number flow to driver
   bottomLine = if modeCar == "Off" then "" else
        targetSpeed

   targetSpeed : number flow to car
   actualSpeed : number flow from car to driver

   increment : void event from driver
   on increment : targetSpeed = targetSpeed + step
               toggle = true

   decrement : void event from driver
   on decrement : targetSpeed = targetSpeed - step
               toggle = true

   switch : void event from driver
   on switch : toggle = !toggle

   toggle : boolean flow
   toggle = clutch < 0.01 && brake < 0.01 && toggle &&
        modeDriver!="Off"

   modeDriver : mode flow from driver

   modeCar : mode flow to car
   modeCar = if toggle then modeDriver else "Off"

   throttle : number flow from driver
   brake : number flow from driver
   clutch : number flow from driver

   targetSpeed = crop(minTarget,maxTarget,if toggle then
        targetSpeed else round(actualSpeed,step))
```

Listing 1. LIL code of the example application

REFERENCES

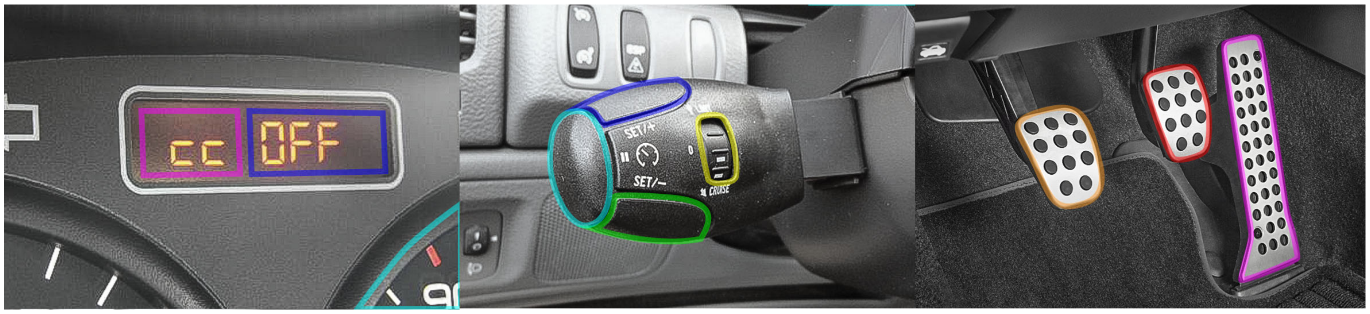

Figure 4. The final UI of the example application