EDONA/HMI – Modelling of Advanced Automotive Interfaces

S. Boisgérault[1], E. Vecchié[1], O. Meunier[2], J.-M. Temmos[3]

1: CAOR, Mathématiques et Systèmes, Mines ParisTech, 60 boulevard Saint-Michel, 75272 Paris, France.
2: Intempora, 2 pl. Jules Gevelot, 92130 Issy les Moulineaux, France.
3: Visteon Software Technologies, 1800 route des Crêtes , 06906 Sophia Antipolis, France.

Abstract: Screens are becoming the most important media for information systems in vehicles. They enabled a wide variety of new services such as navigation systems, driver assistance or entertainment. They also are increasingly replacing the analog instrument clusters used to display classic vehicle information. The design of user interfaces for such targets involves some usual requirements like rapid prototyping and interoperability. As these user interfaces present information that may directly influence the driver's behavior, they shall also be handled as safety-critical software.

In this paper, we describe a model-based process for the design of such interfaces that address these issues. We present – in the context of the EDONA project[1] – a domain-specific model for automotive interfaces that relies on graphic and functional standards. We then describe a code generation architecture and runtime platform.

Keywords: human-machine interfaces, automotive, domain-specific language, model-based design.

1 Introduction

Scope Definition. In car cockpits, screens are increasingly present and used as media for the hosting of human-machine interfaces (HMI). When they are used to display vehicle and environment data, these screens replace or complement the more traditional – analog or digital – instrument clusters. This evolution is motivated by the flexibility of such systems: they can easily support the classic features of instrument clusters but are not limited to a specific component layout. Beyond this extra configurability, they also enable the design of innovative and highly-specific interfaces that would not be otherwise possible. Such interfaces may support advanced in-vehicle applications such as driving assistance, navigation systems, vehicle communication, etc. Therefore high-end vehicles and car prototypes benefit the most from this increasing flexibility.

While we present in this paper models and tools to design human-machine interfaces for such in-vehicle screens, due to its considerable diversity, we do not address the full range of such systems. Embedded systems with little integration with the vehicle functions – DVD players for example – may be considered as traditional consumer electronics products as far as interface design is concerned.

Despite a flexible and large design space that will be detailled in the next section, we also exclude from our design scope some very specific interfaces that are categories in their own right. For example, we do not consider integral design of 3D navigation systems or video systems. Our framework however may be used to design some parts of such systems as it will be demonstrated in the last section.

What we address specifically is the direct extension of the instrument cluster concept: design of display components that represent the evolution of vehicle or environment data in the most adequate way. Flexibility in the design space is required to allow for specific design accomodating traditional data (RPM, speed, fluid levels, etc.) as well as advanced or even future services. In this respect, the last section of the paper will present an interface focused on pedestrian safety, an original source of data if any. As such information systems may have a direct influence on the driver's behavior, safety is an issue. The HMI design process and models shall therefore be amenable to safety analysis and certifications.

Finally, due to the advanced nature of applications, our solution is at ease with high-end graphic platforms – typically TFT-LCD, full-color reconfigurable display with extensive 2D graphics library support. However for simpler applications and low-end platforms (LED, VFD, etc.), our framework can still be used for modelling, simulation and specification but automatic code generation support is not available.

Issues. The design of this new generation of HMIs

[1]This work has been performed in the context of the EDONA project of the System@tic Paris Région Cluster. EDONA is an Open Development Platform to Automotive Standards. It is supported by the "Direction Générale de la Compétitivité, de l'Industrie et des Services (DGCIS)", the "Conseil Régional d'île de France", the "Conseil Général des Yvelines", the "Conseil Général du Val d'Oise" and the "Conseil Général des Hauts de Seine". www.edona.fr

Figure 1: A hybrid – mechanical and screen-based – instrument cluster. Displays may be used to reproduce the traditional features of mechanical systems (here RPM and speed dials) – as demonstrated in the top figure – but may also be easily reconfigured to display other kinds of vehicule information.

raises several issues. Interoperability – the ability to easily exchange components specifications and designs – is a major industrial issue. Also, as they present information that may directly influence the driver behavior, these HMIs shall also be treated as safety-critical software components. We propose here a model-based approach that relies on a scene graph graphics layer and a synchronous functional layer. Synchronous languages [9, 13, 3] are standard representations for safety critical software, they enjoy a clear and simple semantics that allows various program analysis like automatic test generation [15] or model-checking [10]. An efficient design process and the ability to do some rapid prototyping are crucial as well. Our model comes together with dedicated tools for the assisted design and the compilation of components into executable graphics interfaces.

2 HMI Models

We formalize in this section the meta-model that underlies the whole EDONA/HMI design process. The HMI graphics content is described as a data-driven scene graph, a powerful yet simple data model described in section 2.1. The document structure is fixed and no graphics node is ever created at runtime ; the model is therefore amenable to extensive analysis. This graphics layer is complemented with a functional layer that provides a component model for HMI to be handled at the proper abstraction level. The functional model is a data-flow language described in the section 2.2, an execution model widely used in the design safety-critical systems. The details of their integration that forms the HMI meta-model is discussed in section 2.3.

2.1 Data-driven Scene Graph

The graphics layer of the EDONA/HMI model is a 2D data-driven scene graph. A scene graph – or hierarchical display list – is a structured collection of graphic nodes, commonly used in 2D and 3D vector graphics. It is ultimately composed of primitive nodes such as vector graphic shapes (lines, circles, etc.), images and text. On the other hand, compound nodes are node containers that structure the scene graph.

All types of graphic nodes may be configured. Consider for example the text node definition

$$\begin{aligned}\mathbf{text}(&\text{text} \leftarrow \text{“hello”},\\ &\text{transform} \leftarrow \text{rotate}(90),\\ &\text{fill} \leftarrow \text{rgb}(255,0,0),\\ &\text{family} \leftarrow \text{“Helvetica”})\end{aligned} \tag{1}$$

Such an attribute assignment is used as a single and uniform mechanism to configure node core data (‘text’), and the geometrical transformations (‘transform’) as well as style options (‘fill’ and ’family’) that are applied to the node.

Some attributes used in this assignment are type-specific: the ‘text’ data and font ‘family’ for example are meaningless for all but text nodes. Other attributes are shared between several node types: the ‘fill’ color is applicable to all vectorial constructs and the geometrical ‘transform’ to all graphic nodes. The list of all attributes applicable to a given node type is its *context*. The union of all such contexts is called the *graphics context*.

Finally, node definitions may be partial: although the ‘opacity’ attribute belongs to the text node context, it does not appear in the node definition (1). Such undefined values are to be interpreted as a convenience mecanism for default values – 1.0 would be the opacity natural default for example. They may also be used to support context inheritance.

We formalize briefly the structure of this scene graph grammar. More information about the set of node types and corresponding context attributes that we actually consider may be found in section 2.4. Let $\mathfrak{P}$ be the set of primitive node types, let $\mathbf{g}$ – the group – be the single compound node, and let $\mathfrak{G}$ be the graphics context. Our graphics model is given by the system

$$\begin{aligned}\text{node} \quad &\coloneqq \quad p(\mathfrak{G} \leftarrow \text{val}_1, \text{val}_2, \cdots), \;\; p \in \mathfrak{P} && (2)\\ &\mid \quad \mathbf{g}(\mathfrak{G} \leftarrow \text{val}_1, \text{val}_2, \cdots)\langle\text{node}\rangle && (3)\\ &\mid \quad \text{node}_1,\ \text{node}_2 && (4)\end{aligned}$$

The notation $(\mathfrak{G} \leftarrow \text{val}_1, \text{val}_2, \cdots)$ is only a shortcut for the graphics context assignment $(a_i \leftarrow \text{val}_i \,|\, a_i \in \mathfrak{G})$

as it appears in the example (1). We denote $\mathfrak{G}(t)$ the context of the node type t and $\perp$ the undefined value. The graphics context assignment $\mathfrak{G} \leftarrow \mathrm{val}_1, \mathrm{val}_2, \cdots$ in (2) is subject to the consistency constraint

$$\{a_i \in \mathfrak{G} \mid \mathrm{val}_i \neq \perp\} \subset \mathfrak{G}(p)$$

and accordingly the same assignment in (3) to

$$\{a_i \in \mathfrak{G} \mid \mathrm{val}_i \neq \perp\} \subset \mathfrak{G}(\mathrm{g})$$

For the sake of simplicity, we may assume here that all context attribute values belong to the same value space. A concrete instance of value space would gather heterogeneous attribute types such as:

$$\begin{array}{rcl} \mathrm{val} & \coloneqq & \perp \\ & \mid \mathrm{int} \mid \mathrm{float} \mid \mathrm{matrix}(3,3) \\ & \mid \mathrm{text} \mid \mathrm{color} \mid \mathrm{fontname} \end{array} \quad (5)$$

A scene graph is in our terminology *data-driven* if the value space used in (2) and (3) is composed of undefined values and data flows.

$$\mathrm{val} \coloneqq \perp \mid \mathrm{flow} \quad (6)$$

During the HMI execution, the functional layer – a data-flow program whose structure is described in the section 2.2 – computes the *graphics state*: the set of concrete values that are substituted to the data flows. In this data model, the structure and nodes of the graphics document are given but the graphics state is mutable.

2.2 Functional Model

In the EDONA/HMI model, the graphics state is driven by a synchronous data-flow program, formally defined by the following grammar:

$$\begin{array}{rcll} \mathrm{stmt} & \coloneqq & \mathrm{flow} = \mathrm{expr} & (7) \\ & \mid & \mathbf{next}(\mathrm{flow}_1) = \mathrm{flow}_2 & (8) \\ & \mid & \mathrm{stmt}_1,\ \mathrm{stmt}_2 & (9) \\ & \mid & \mathrm{stmt}\ \mathbf{when}\ \mathrm{flow} & (10) \end{array}$$

The ',' operator denotes concurrent execution of statements, '**next**' the delay operator and the '**when**' construct represents conditional execution. Our actual functional model also has default values for constructs (8) and (10) so that data flow values are always well defined. This feature is not presented for the sake of simplicity. For the same reason we do not discuss typing issues: expression expr are either constants, function calls or the "if-then-else" construct:

$$\begin{array}{rcll} \mathrm{expr} & \coloneqq & \mathrm{constant} & (11) \\ & \mid & \mathrm{fct}(\mathrm{flow}_1, \mathrm{flow}_1, \ldots) & (12) \\ & \mid & \mathrm{flow}_1\ ?\ \mathrm{flow}_2\ :\ \mathrm{flow}_3 & (13) \end{array}$$

We do not detail the well-known semantics of such data-flow programs but refer the reader to [4]. Instead we focus on issues that directly impact the HMI component model and the integration of its functional and graphics layer.

Scoping. Every statement defines implicitely input and output data flows determined by the functions $\mathfrak{I}$ and $\mathfrak{O}$:

$$\begin{array}{rcl} \mathfrak{I}(\mathrm{flow} = \mathrm{expr}) & = & \mathfrak{I}(\mathrm{expr}) \\ \mathfrak{I}(\mathbf{next}(\mathrm{flow}_1) = \mathrm{flow}_2) & = & \{\mathrm{flow}_2\} - \{\mathrm{flow}_1\} \\ \mathfrak{I}(\mathrm{stmt}_1,\ \mathrm{stmt}_2) & = & (\mathfrak{I}(\mathrm{stmt}_1) - \mathfrak{O}(\mathrm{stmt}_2)) \cup \\ & & (\mathfrak{I}(\mathrm{stmt}_2) - \mathfrak{O}(\mathrm{stmt}_1)) \\ \mathfrak{I}(\mathrm{stmt}\ \mathbf{when}\ \mathrm{flow}) & = & \mathfrak{I}(\mathrm{stmt}) \cup \{\mathrm{flow}\} \\ \mathfrak{I}(\mathrm{constant}) & = & \varnothing \\ \mathfrak{I}(\mathrm{fct}(\mathrm{flow}_1, \mathrm{flow}_2, \ldots)) & = & \{\mathrm{flow}_1, \mathrm{flow}_2, \ldots\} \\ \mathfrak{I}(\mathrm{flow}_1\ ?\ \mathrm{flow}_2\ :\ \mathrm{flow}_3) & = & \{\mathrm{flow}_1, \mathrm{flow}_2, \mathrm{flow}_3\} \end{array} \quad (14)$$

and

$$\begin{array}{rcl} \mathfrak{O}(\mathrm{flow} = \mathrm{expr}) & = & \{\mathrm{flow}\} \\ \mathfrak{O}(\mathbf{next}(\mathrm{flow}_1) = \mathrm{flow}_2) & = & \{\mathrm{flow}_1\} \\ \mathfrak{O}(\mathrm{stmt}_1,\ \mathrm{stmt}_2) & = & \mathfrak{O}(\mathrm{stmt}_1) \cup \mathfrak{O}(\mathrm{stmt}_2) \\ \mathfrak{O}(\mathrm{stmt}\ \mathbf{when}\ \mathrm{flow}) & = & \mathfrak{O}(\mathrm{stmt}) \end{array} \quad (15)$$

In order to gain an explicit control of the signals visibility we add an extra construct – the component – to the list of available statements (7-10).

$$\mathrm{stmt} \mid \mathbf{c}(\mathrm{inputs}, \mathrm{outputs})\langle \mathrm{stmt} \rangle \quad (16)$$

Components input and output flows are given in their declaration:

$$\begin{array}{rcl} \mathfrak{I}(\mathbf{c}(\mathrm{inputs}, \mathrm{outputs})\langle \mathrm{stmt} \rangle) & = & \mathrm{inputs} \\ \mathfrak{O}(\mathbf{c}(\mathrm{inputs}, \mathrm{outputs})\langle \mathrm{stmt} \rangle) & = & \mathrm{outputs} \end{array} \quad (17)$$

Components interface definition (16) is subject to the following consistency conditions:

$$\mathrm{inputs} \supseteq \mathfrak{I}(\mathrm{stmt}) \text{ and } \mathrm{outputs} \subseteq \mathfrak{O}(\mathrm{stmt}) \quad (18)$$

2.3 HMI Model – Integration

The HMI model integrates the graphics and functional models defined in the sections 2.1 and 2.2. The core object of HMI models is the HMI element. It is either a graphics node or a functional statement.

$$\mathrm{elt} \coloneqq \mathrm{node} \mid \mathrm{stmt} \quad (19)$$

Graphics and functional definition of (2, 4, 6) and (7-13) are modified so that their right-hand side accept hmi elements instead of nodes and statements.

Graphics nodes being now integrated with the functional layer, data flows from the functional layer may drive the graphics state evolution. From the functional point of view, primitive node provide the following data flows:

$$\begin{array}{rcl} \mathfrak{I}(p(\mathfrak{G} \leftarrow \mathrm{val}_1, \mathrm{val}_2, \cdots)) & = & \{a_i \in \mathfrak{G} \mid \mathrm{val}_i \neq \perp\} \\ \mathfrak{O}(p(\mathfrak{G} \leftarrow \mathrm{val}_1, \mathrm{val}_2, \cdots)) & = & \varnothing \end{array} \quad (20)$$

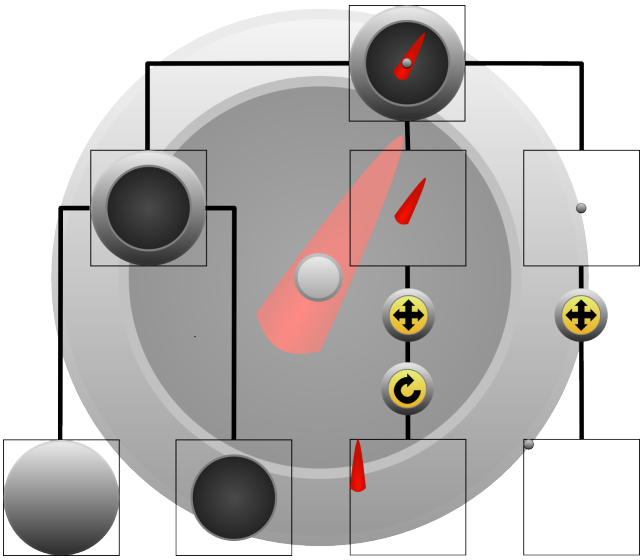

Figure 2: The scene graph representation of a gauge. As a data-driven scene graph, all but the needle rotation attribute may be assigned constant values. If the gauge is used as a speedometer, functional modelling may be used to convert the vehicle speed into an angle and to ensure that the maximal angle threshold is never exceeded.

Finally, the graphics group defined in (3) and function component defined in (16) are replaced by a single construct – the hmi group component – that acts as a HMI element container and merges the graphics and functional hierarchies:

$$\mathbf{gc}(\text{graphics}, \text{interface})\langle\text{elt}\rangle \tag{21}$$

where

$$\text{graphics} \;\coloneqq\; \mathfrak{G} \leftarrow \text{val}_1, \text{val}_2, \cdots \tag{22}$$

$$\text{interface} \;\coloneqq\; \text{inputs},\ \text{outputs} \tag{23}$$

Its inputs and outputs are given by:

$$\mathfrak{I}(\mathbf{gc}(\text{graphics}, \text{interface})\langle\text{elt}\rangle) = \text{inputs} \cup \{a_i \in \mathfrak{G} \mid \text{val}_i \neq \bot\} \tag{24}$$

$$\mathfrak{O}(\mathbf{gc}(\text{graphics}, \text{interface})\langle\text{elt}\rangle) = \text{outputs} \tag{25}$$

This hmi group component interface (21) is subject to the consistency conditions (18).

2.4 Model Format and Serialization

The format used to represent HMI models – whose structure was described in the previous sections – has a considerable practical impact. In this section we present the motivations behind the design of the EDONA/HMI format, explain why we have selected a subset of an existing graphics standard – *Scalable Vector Graphics* (SVG) – as the basis for our format and how we extended it with appropriate HMI-specific constructs.

Format Selection. The selection of SVG as the modelling language for the description of graphic content was driven by our research for solutions that would improve interoperability in HMI design. Despite significant differences, many leading solutions for embedded HMI design (such as VAPS XT, ALTIA Design or Scade Display [16]) have adopted a similar model-driven process. Their model for graphics share the most important structural features such as a tree-like document structure, support for affine transforms, basic and complex shapes, styling, etc. They are however based however on proprietary formats. On the contrary, SVG is an open and mature recommandation from the World Wide Web (W3C) consortium, an international standards organization known for standards such as the XML technology.

SVG is a language for describing two-dimensional graphics in XML. While the original SVG – version 1.0 in 2001 – was designed to support vector graphics on the Web, SVG version 1.1 is used nowadays for all kind of vector graphics descriptions. It has a large feature set that proved to be suitable for the kind of graphics content that was presented in section 2.1. It also has given birth to two standard subsets – named *profiles* in the SVG recommandation – SVG Tiny and SVG Mobile. They are specifically designed for mobile devices, platforms that are primarily characterized by specific constraints in terms of CPU speed, memory size, color support, etc. The family of mobile devices obviously contains mobile phones and personal digital assistants but the target platforms for embedded HMI in vehicles share similar constraints. The usage scenarios for those profiles explicitely include the modelling of graphical user interfaces.

More generally, SVG 1.1 is not a monolithic standard as it provides consistent policies to *profile* (restrict) and extend the standard in order to adapt to specific usage scenarios. As SVG 1.1 is described as a set of independent modules, the definition of profiles is simple. The standard extensibility policy recommands the use foreign namespaces to complement SVG data with application-specific content. The extra information is simply ignored by conformant SVG applications. This policy is used effectively in generic SVG authoring tools [12] and also in SVG models exported by some embedded interface designers.

The existence of several SVG authoring tools makes the design of new HMI graphic content a simple task. SVG being an authoritative description for vector graphics, some HMI design tools used in the automotive industry already have a partial support for it. Therefore, the selection of SVG as a core format has the potential to increases interoperability between generic and application-specific design tools. Beyond existing authoring tools, we also greatly benefit from software libraries that support SVG such as the Apache Batik

SVG Toolkit [1]. Existing SVG software reduce the development cost of EDONA/HMI model editors as well HMI graphics rendering engines.

EDONA/HMI SVG profile. We finally adopted a custom mobile profile, very similar to SVG Tiny 1.1. This profile brings a welcome simplification with respect to the full standard. Notably, the style of graphic nodes can only be set through individual XML attributes – called *presentation attributes* – and not the 'style' or CSS styling mecanism. This simplification allows a uniform implementation of context assignment in data-driven scene graphs. The handling of numerical data is also simpler because length data have no units and always refer to the local coordinate system. Text graphics nodes are also more manageable because a single style can be applied to them.

The study of HMI design use cases has also taught us that the SVG Tiny profile supports almost every features we needed and conversely that many of the graphics elements and context attributes that have been removed (filters, the support for "group opacity", etc., see [17] for a detailled list of supported constructs) were costly to render and not adapted to the context of dynamic document rendering. We have only reintroduced three features that all belong to the full SVG 1.1 specification: gradients, opacity and clipping. We however limit them to their simplest form: gradients and opacity constructs are supported as in the SVG Tiny 1.2 candidate recommandation and clipping as in SVG Basic 1.1. We believe that this set of constructs is currently a good trade-off between model simplicity and the support for a large range of HMI models.

On the other hand, we exclude from the EDONA/HMI profile the support for declarative animation of SVG Tiny 1.1: the kind of dynamic document that these constructs allow are strictly contained in our full HMI model provided that a 'time' input is available to HMI components. On the other hand, the data-driven model support direct update of the graphic state that cannot be expressed within the time-based model.

Data Flows Format. The EDONA/HMI format extends the SVG graphics format to make the scene graph data-driven and to describe functional constructs. The extension design is compatible with W3C recommandation and enables conformant SVG interpreters and viewers to properly process all HMI graphics content. First of all, all data flow elements belong to the EDONA/HMI namespace `http://www.edona.fr/hmi` namespace, so that non-graphic data are ignored during SVG processing. Then, instead of mapping group components to an XML element in the EDONA/HMI namespace, we implement them as a svg group that contain an XML element with local name 'component' in the EDONA/HMI namespace. As a consequence, graphic content in group components are always visible to SVG interpreters and properly processed as groups.

The data-driven scene graph model of section 2.1 is described through the embedding of `input` XML elements into graphics node elements that specify the context attribute they act upon. Although this is an aspect of the model that was not described in the previous sections, a symmetric `output` element exists to make available the initial graphics state to the functional layer.

Functional expressions (constants, function calls and the "if-then-else" construct) as well as the delay operator are mapped to XML elements and conditional execution is handled at group component level exclusively. Functional expressions have explicit input and outputs interfaces. Sharing data flows between statements is handled by a list of links in the parent component element. This simple block-diagram model allows easy generation and processing of functional model at the expense of a verbose XML model. In particular, no micro-language is used to encode functional information in attributes as the SVG format does for path, transform, etc. so that all functional processing may be done with standard XML tools.

A standard library of functions has also been defined. It contains the classical logic, numeric and comparison operators. It also provides text processing function and type conversions operators between numeric and text types – the only primitive types that we actually consider. SVG array-like attributes – such as 'path' and 'transform' – are not handled with specific types. Instead they are either considered as text attributes and as such are fully mutable or considered as numeric arrays with a fixed structure, the numeric content being then their only mutable part.

3 EDONA/HMI Software Tools

Tool support is crucial to ensure the success of the EDONA/HMI model. Our EDONA/HMI Prototyping toolchain includes a set modelling tools, a generator of HMI software components and a runtime architecture based on JAVA. This platform is specialized for simulation in the design loop, component testing and deployment on car prototypes, particularly hosts of intelligent systems transportation (ITS) applications. While the runtime component of this platform is not compatible with usual embedded systems constraints, it is open-source and intended to simplify the generation of EDONA/HMI components for such targets.

3.1 Model Management

Several types of software contributions were made to support the EDONA/HMI design process. Among them, a schema-based validator that checks the consistency of XML model files against the EDONA/HMI grammar, several gateways between EDONA/HMI models and other formats, as well as internationalization tools and a documentation generator.

Moreover, a Python library was developed that performs XML data binding on HMI model files: it automatically establish a mapping between HMI model constructs and Python classes. This library was designed to be a suitable basis for many EDONA/HMI-related development – for example, to speed up the development of a HMI model editors. Internally, it was used for automatic generation of HMI models and model transformations. The need we had to start with models that may be incorrect – for example with SVG constructs that do not belong to the mobile profile – or are not complete models and will not be until the final steps of their design – has motivated the design of a lazy data binding scheme where instances of EDONA/HMI constructs are only bound when they are effectively accessed. As a consequence, it may be used as the foundation of a validators that goes beyond grammar-level consistency checks.

The XML data binding library is also used in the early stage of the code generation process: the functional layers of HMI models are simplified into equivalent models with no hierarchy and no conditional activations. Graphic constants that feed the functional layer are also extracted to fully decouple the functional model from the graphics one so that the functional code that is generated is testable and also supports the master-slave model explained in the next section.

3.2 Runtime Architecture

HMI execution policy. Both graphics and functional part of HMI components may be generated with different execution policy, the two extreme choices being either a full interpreter architecture that executes the model at runtime or a more extensive code generation phase that produces model-independent executable programs. We discuss in this section the pros and cons of both options.

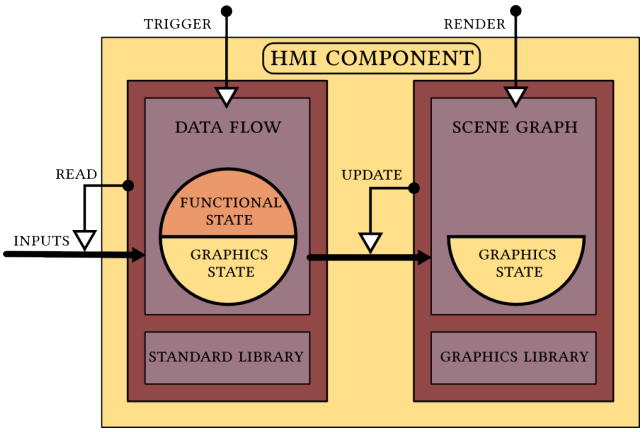

Figure 3: architecture of the EDONA/HMI Prototyping runtime

Our strategy for the generation of the functional part of HMI components is straightforward: it consists in a model simplification step (see section 3.1) followed by a code generation step. Code generation is a simple matter for such data-flow models where static scheduling information is available [4]; its is also more efficient at runtime than an interpreter when many low-level operations are performed. More importantly, in the context of HMI model prototyping, it is also simpler to implement: the complexity of hierarchy and conditional activation have been eliminated in the model simplification step while these constructs have to be maintained in the interpreter scheme.

The optimal execution policy for the graphics part of HMI component is a more complex issue and largely depends on design process and target requirements. For HMI as software components in a context with strong embedding and safety constraints, the simplicity, size and performance that result from an extensive code generation process is likely to be preferred. A typical code generation would then linearize the scene-graph and translate graphic primitives into calls to a low-level hardware-accelerated graphics platform such as OpenGL ES, a subset of OpenGL that targets embedded platforms. Considering the SVGL toolkit, an OpenGL based SVG library, display lists can also be used to drastically speed up rendering performances for static parts of an SVG document [5]. On the other hand, in the context of simulation in the design loop, testing and prototyping, the availability of debugging information and model snapshots, complete with the current graphics state is crucial. Due to the complexity of the actual graphics model, maintaining in memory the graphics data in a structure similar to the model is the simpler strategy. The significant performance loss that may result from this strategy may be partially offset by pre-rendering of the static graphics content – something the EDONA/HMI model is totally suitable for.

The EDONA/HMI Prototyping component generator therefore uses a mixed strategy: JAVA code is generated for the functional layer whereas the graphics model is interpreted and rendered using the BATIK SVG toolkit – a component of the APACHE XML GRAPHICS project – in a way that updates of the graphics model are always accessible through a standard XML API. This strategy allows – at any time during execution – to serialize the state of a HMI component as a new model and also allows for several graphics backends: no graphics (for testing purposes), JAVA AWT and image buffer.

There also exist some alternatives to the BATIK SVG toolkit, including the SVGL toolkit as mentioned earlier using OpenGL as backend. The RSVG library is also a SVG toolkit based on Cairo, a software library used to provide a vector graphics-based, device-independent API for software developers. Cairo is designed to provide primitives for 2-dimensional drawing across a number of different backends and it is de-

signed to use hardware acceleration when available.

HMI runtime interface. A pure synchronous interface for HMI components would be conceptually simple: a sequence of activation signals, coming with new values for the HMI inputs, would trigger an execution step for the functional layer and at the same time update of the HMI graphics. However this scheme would have a severe drawback: by synchronizing the rendering of the graphics with the input flow, it may impose a fast rendering cycle, beyond what the target is able to execute due to the cost of this step. In the most common use case, a fixed frame rate is given that provides a sufficient quality in the user experience and no extra rendering steps should be performed, or the rendering should be synchronized with an external video stream. For other situations, such as execution for testing purposes, rendering is a step that should occur on specific breakpoints or explicit demands, the functional layer being most of the time the only active component.

Therefore, to avoid unnecessary performance issues and provide the needed flexibility, to the synchronous activation signals is added a – possibly asynchronous – rendering signal. To ensure the correctness of executions, a duplication of the graphics state is required in the functional component so that every new set of input values may update the graphics state. This state is however only transferred to the graphics layer upon request. This partial decoupling and resulting layers architecture is depicted in figure 3.

3.3 Intelligent Transportation System Use Case

Finally, to adress the needs of intelligent transportation systems (ITS) embedded applications, we extended the BATIK SVG toolkit to support the display of graphic and textual information as video overlays as well as the inclusion of embedded controls. A full-fledged HMI was also designed for the LOVE [7, 14] project. LOVE aims to use multi-sensor tracking system to improve the road safety for pedestrians. Our HMI interface was used to display video streams and – through specifically designed interfaces – data such as pedestrians location and risk of collision. Real-time data acquisition and management is performed by RTMaps[2], as well as the communication with the HMI stack.

This use case validated our approach, showing its expressivity to quickly create custom components, and the possibility to quikly integrate these components into a full consistent system. External video streams were also integrated, showing its capacity to interface with an asynchronous environment.

[2] "Real-Time Mutisensor Advanced Prototyping Software", `http://www.intempora.com`.

Figure 4: EDONA-LOVe interface for the safety of pedestrians

4 Conclusion

We presented the EDONA/HMI design process for the design of Human-Machine Interfaces on screens in car cockpits. This method successfully addresses typical industrial issues in the domain of HMI design: interoperability, proper modelling, flexibility and rapid prototyping. In addition to the design process efficiency gain, our solution enables to promote the design process of HMI in cars to the same level as other safety-critical domains like avionics or power plants. Safety-critical issues are addressed by relying on a data driven scene graph model for the graphics part and a synchronous model of computation for the functional part of the EDONA/HMI system.

References

[1] Apache XML Graphics – Batik SVG Toolkit. `http://xmlgraphics.apache.org/batik/`.

[2] S. Boisgérault, M. O. Abdallah, and J.-M. Temmos. SVG for automotive user interfaces. In *SVG Open*, Nuremberg, Germany, 2008.

[3] F. Boussinot and R. de Simone. The Esterel language. *IEEE*, 79(9):1293–1304, 1991.

[4] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, January 1987. ACM.

[5] S. Conversy and J.-D. Fekete. The svgl toolkit: enabling fast rendering of rich 2d graphics. Technical Report 02/1/INFO, École des Mines de Nantes, janvier 2002.

[6] EDONA: Environnements de développement ouverts aux normes de l'automobile – website. http://www.edona.fr.

[7] G. Gate, A. Breheret, and F. Nashashibi. Centralized fusion based algorithm for fast people detection in dense environment. In *Proc. of the IEEE International Conference on Robotics and Automation*, Kobe, Japan, May 2009.

[8] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[10] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering (TSE), Special issue: Specification and Analysis of Real-Time Systems*, 18(9):785–793, September 1992.

[11] Human Machine Interface – Work Package 4. EDONA HMI Format. Report, EDONA, 2008.

[12] Inkspace - open source scalable vector graphics editor. Technical report. http://www.inkscape.org/.

[13] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *IEEE*, 79(9):1321–1336, 1991.

[14] LOVe: Logiciel d'Observation des Vulnérables, project website. http://love.univ-bpclermont.fr.

[15] V. Papailiopoulou. Automatic test generation for lustre/scade programs. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 517–520, Washington, DC, USA, 2008. IEEE Computer Society.

[16] The standard for the development of critical embedded display software. http://www.esterel-technologies.com/products/scade-display/.

[17] Mobile SVG Profiles: SVG Tiny and SVG Basic. W3C recommendation, W3C, June 2009. http://www.w3.org/TR/SVGMobile/.

[18] Scalable Vector Graphics (SVG) 1.1 specification. W3C recommendation, W3C, January 2003. http://www.w3.org/TR/SVG.