Engineering Device-spanning, Multimodal Web Applications using a Model-based Design Approach

Sebastian Feuerstack
Universidade Federal de São Carlos
Rodovia Washington Luís, km 235
São Carlos - São Paulo – Brasil
sfeu@dc.ufscar.br

Ednaldo Brigante Pizzolato
Universidade Federal de São Carlos
Rodovia Washington Luís, km 235
São Carlos - São Paulo – Brasil
ednaldo@dc.ufscar.br

ABSTRACT

Nowadays the web is an ubiquitously available source of information that can be accessed through a broad range of devices, such as smart phones, tablets and notebooks. Although web applications can be used through several devices, they are controlled and designed for a one-to-one connection type of interaction, which prevents device-spanning multi-modal interactions.

We propose a model-based run-time framework to design and execute multi-modal interfaces for the web. Different to a model-based design that implements reification, a process to derive concrete models from abstract ones by transformation, we design interactors that keep all design models alive at run-time. Interactors are based on finite state machines that can be inspected and manipulated at run-time and are synchronized over different devices and modalities using mappings. We show the expressiveness of state charts for modeling interactions, interaction resources, and interaction paradigms.

We proof our approach by checking its conformance against common requirements for multimodal frameworks, classify it based on characteristics identified by others, and present initial results of a performance analysis.

Categories and Subject Descriptors

H.5.2 **[Information Interfaces and Presentation]**: User Interfaces - *Input devices and strategies, Interaction styles, Prototyping*; D.2.2 **[Software Engineering]**: Design Tools and Techniques – *User Interfaces.*

General Terms

Design, Human Factors, Languages.

Keywords

Model-based User Interfaces, Multimodal Interfaces, Web Application Development, HCI.

1. INTRODUCTION

With the ubiquitously available word wide web information can be accessed from any place and any device connected to the internet. Just a web browser is required to search the web or even run web applications in the cloud.

Different to graphical interfaces of web applications that are often adapted to consider different screen sizes, the support for different control modes is often limited to the classic keyboard and mouse based input control. Input controls that go beyond basic touch gestures are not considered or are implemented for specific domains like an in-car control that uses wheels, a touchscreen or voice commands. Multimodal applications that combine several modes into one single application are even more complex to implement. Model-based software development has been successfully performed to support the semi-automated generation of interfaces for different platforms (like HTML [9], XHTML+Voice [9] or even 3D interfaces [9]). But to our knowledge there are still no frameworks available that support the model-based design and execution of multimodal interfaces that can enable a flexible combination of modes and media based on the preferences of the user.

We implement web applications that enable the user to switch devices and modalities or even combine them on demand. The basic idea is to offer a model-based design of custom interactors that are self-executable and that can be distributed and synchronized through several devices and modalities. Different to model-based user interface design (MBUID) that generates isolated interfaces for different platforms by a structured abstract-to-concrete modeling of interaction, we design interactors that are, once they have been designed, assembled by user interface builders to form a multi-modal interface. Therefore, we adapt classical user interface building instead of requiring the developer to learn new languages and processes to create interfaces.

The paper is structured as follows: The next section discusses related work. Thereafter, in section 3 we present our modeling approach that re-uses common MBUID models, such as task, abstract and concrete user interface models for the interactor design. Section 4 explains how the interactors get instantiated at run-time and are executed as part of a web server. Section 5 presents an evaluation of the platform based on practitioners' requirements (5.1), common characteristics for multimodal frameworks (5.2) that have been identified by others and states initial results of a performance evaluation (5.2).

2. RELATED WORK

MBUID has been applied for a long time to reduce the development costs of user interfaces to enable them running on different platforms. Languages, such as USIXML [14] and tools

such as [12] have been proposed to form a structured development process. These processes typical start from a very abstract description of the user interface tasks to a more concrete one considering the characteristics of certain modalities (a concrete user interface model) and end up with a final user interface to be executed on a specific platform [3]. This process through several abstractions requires anticipation skills to understand how changes on an abstract level will be reflected in the final user interfaces [14]. Most of the approaches that we are aware of are focusing on design-time support.

Different to these approaches our interactor-based interfaces can be flexibly extended to new modes and media just by adding new interactors and mappings to a running system. Our approach is inspired by the findings of the iCARE platform [13] that supports building multimodal interaction out of components that are connected based on the CARE properties. These properties describe the relations between different modes, such as their complementary, redundant or equivalent combination.

Some frameworks have been already proposed to ease the creation of multimodal interaction controls like the Open Interface Framework [10] that enables a developer to assemble multimodal controls out of components. It focuses on bridging different interaction technologies by enabling the interconnection of device drivers and signal processing algorithms to form new kinds of multimodal interaction setups. But the specification of how an application is controlled with an assembled interaction setup is still done at the source code level.

We are using the same model abstractions as initially summed up by the CAMELEON framework [3], but applying a development process that conforms to classical user interface construction: Using an user interface builder that offers a palette of widgets (that we call interactors) that can be directly assembled to form a multimodal interface. This is an approach that has been followed by Gummy [11] as well, but was focused to create different graphical interface for different sized screens and has to our knowledge no support for extending the existing widget set. User interface builders ease the development of interfaces by offering pre-defined widgets but lack the possibility to add new ones or require the developer to enhance the tool itself on the source code level. Our approach tackles this issue. Therefore, we propose to construct interactors based on State Chart XML (SCXML). SCXML[1] is a general-purpose event-based state machine language that is based on Call Control eXtensible Markup Language and Harel State Tables [7]. It is an easily understandable language composed of only few basic concepts (such as states, actions, transitions and events).

Direct model execution has the advantage that even at run-time the models can be the targets of manipulation to consider context changes that could not be predicted at design-time. The unpredictable context-of-use has been named the effective context-of-use [1] as opposed to the predictive context-of-use that can be considered at design-time. We call MBUID approaches that enable direct model execution Model-driven Run-time-Environments (MRE). The Context Mobile Widget (COMET) architecture style [4] is an early MRE approach that offers self-descriptive interactors that are able to handle a set of different contexts-of-use. At run-time the COMET interactors benefit from

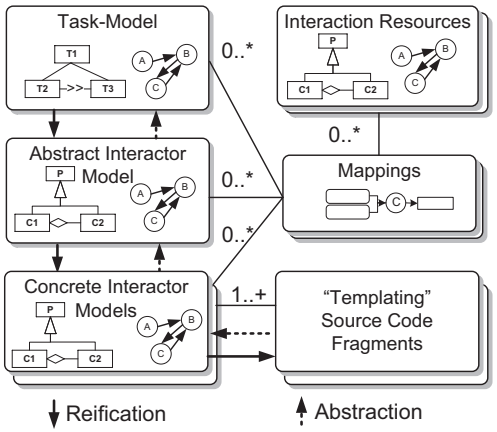

Figure 1. **The run-time models of a multimodal application.**

querying a semantic network to figure out a transformation path for an adaptation goal. The Multi Access Service Platform (MASP) [2] advances model execution one step further and provides all the logic that defines the behavior of an interface inside the models. Therefore, several meta-models (and a meta-meta-model) are provided that describe the construction of MBUID. Like COMETS, MASP has limited support for considering multimodal interfaces by supporting just equivalent multimodal relations.

3. MINT RUN-TIME MODELS

Figure 1 illustrates the six different types of models that we use to specify a multimodal interface. A task model identifies the tasks that are relevant for the application. An abstract model captures the basic behavior of the interface independent of the mode or media being used. Concrete user interface models complement the abstract model by considering platform specific behavior and data. Interaction resource models describe the characteristics and features of physical (e.g. a mouse, or a smart phone) or virtual devices (e.g. gesture recognition). A final interface model stores interface widget templates (source code fragments) for a specific platform. Finally, a mapping model specifies the way model elements are connected to each other. Mappings link at least interactors of two different models, whereas a source code fragment is directly connected to a corresponding concrete model interactor. If a mapping connects an interaction resource model interactor, we call the mapping "multimodal". Otherwise we refer to the mapping as a "synchronization mapping", as it is used to synchronized interactors between different levels of abstraction.

We call the basic model element an interactor and define it based on the YORK interactor [5] as

"an *executable* component in the description of an interactive system that encapsulates a state, the events that manipulate the state and the means by which the state is made perceivable to the user of the system."

Different to the original definition of the YORK interactor we explicitly require an interactor to be executable. This has the advantage that there is no gap between what is designed and executed and has the result that we instantiate interactors on all model abstraction levels at run-time. Thus, one abstract interactor that represents a choice between several options is instantiated together with its related concrete interactors, for instance a

[1] [1] http://www.w3.org/tr/2011/wd-scxml-20110426/, last checked august 25, 2011.

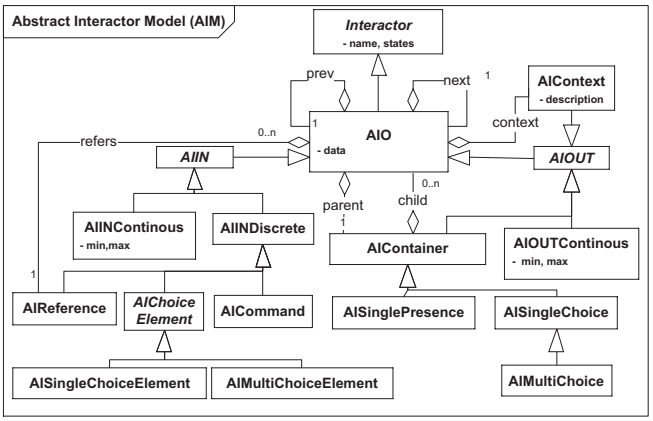

Figure 2. **The interactor relations on the abstract level.**

graphical pull-down menu or a speech prompt with several corresponding answers to choose from.

We specify interactors by state charts. On the one hand this reduces the entry barrier for developers since no new language has to be learned. On the other hand state charts can be directly transformed to state machines that can then be executed.

In the following subsections we describe the most interesting models in greater detail. A task model and templating mechanisms have been discussed earlier [12][1] and are out of the scope for this work.

3.1 Abstract Interactor Model (AIM)

The abstract interactor model describes the general behavior of all user interface elements that can be used in an interface independently from the control mode or the media being used for presenting the element to the user. Figure 2 depicts the entire interactor class hierarchy of the abstract interactor model. The basic abstract interactor object (AIO) that all other abstract interactors are derived from includes the basic functionality of the core interactor class: a unique name, an observable state, the functionality to instantiate itself as a state machine and to process events. Further on, an AIO interactor implements a basic interface navigation enabling the user to navigate between previous/next and parent/child (through the AIContainer) interactors. An AIO interactor usually is presented in an interface within a certain context (AIContext) that clarifies to the user the relevance of the interactor (e.g. by a textual description, or an interactive help). Additionally, an interactor has one or several related navigation shortcuts (AIReference) like a hotkey or a label that can be used to directly navigate to the interactor.

The two most important classifications of the AIM are the distinction between input and output as well as continuous and discrete types of interaction. The former ones are organizational structures if they are related to output, for instance: lists (AISingleChoice), containments (AIContainer) and commands, or choices like individual list elements if they are input-related. Different to a discrete interactor that defines an identifiable and referable setting, a continuous interactor can only be controlled and observed relatively to its previous settings. Examples are a dimmer to control the light level or a progress bar of a graphical interface.

Figure 3 illustrates the behavior of the AICommand interactor by a state chart. This interactor is used to enable a user initiating single actions. It's general lifecycle has been derived from the abstract AIO interactor that specifies an interactor to be "initialized" upon startup, "organized" if the navigational relationships to its neighbors have been computed, "presenting" while it is part of an active user interface, and finally "suspended" after it disappeared from the active user interface presentation. During the time an interactor is in the "presenting" state the navigation is possible by moving the user focus from one interactor to the next or previous one. Thus, as soon as an interactor receives a "next" event while it is "focused", it figures out its next neighbor and defocuses itself after sending a "focus" event to the next interactor.

What makes the AICommand interactor special is its capability to store an activation state that runs in parallel to the "presenting" super state. If the interactor is in state "presenting" and in the focus of the user it can be "activated". A possible concrete

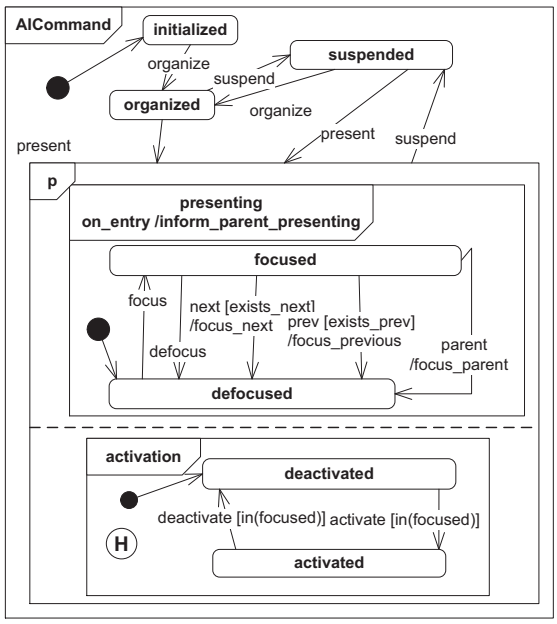

Figure 3. Behavior of the abstract AICommand interactor.

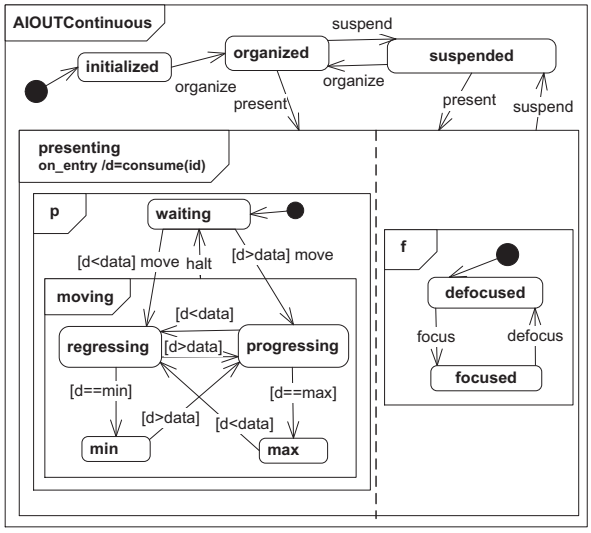

Figure 4. **Behavior of the AIOUT Continuous interactor**

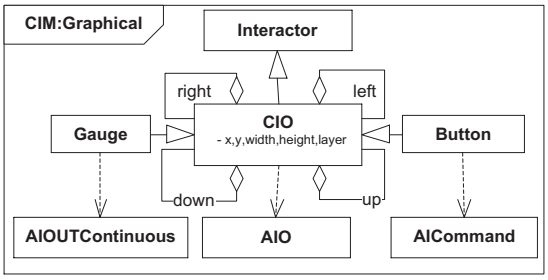

Figure 5. Excerpt from a graphical CIM

interactor that matches this abstract one is a button that we will present later on.

An example of a continuous interactor is shown by figure 4. It implements the same overall lifecycle as the AICommand interactor and represents an output-oriented interactor. Therefore, the interactor listens to data from a streaming source while it is being presented, which we specify by an "on_entry" statement. Depending on the data "d" that it retrieves, the interactor changes its state to be in "regressing" or "progressing". Upon initialization a minimum, maximum as well as a threshold can be defined.

3.2 Concrete Interactor Model (CIM)

The concrete model is in charge of representing an interface on a specific mode and media. Each platform has its own concrete model with two constraints:

1. It can only enhance the data structure and behavior of abstract interactors. Thus, no overlapping definitions between AIM and CIM are allowed.
2. It is required to specify at least one concrete interactor for each AIM interactor. Thus, the AIM is the least common denominator for all mode and media and it is therefore ensured that for an AIM based interface design there is always a complete reification to a CIM one.

Figure 5 illustrates an excerpt of a graphical CIM with the relations to the AIM interactors. Graphical interfaces organize elements in a coordinate system. This is reflected by the Concrete Interactor Object (CIO) that adds the data structure to store positions and sizes. Its lifecycle structure is similar to the AIO, which eases a direct synchronization, but the state semantic is different: It starts by being "initialized", then enters the "positioning" phase during which it is decided e.g. by the task model, what other interactors are presented simultaneously. After the coordinates for each interactor have been calculated based on the layout and screen constraints, it enters "positioned" and

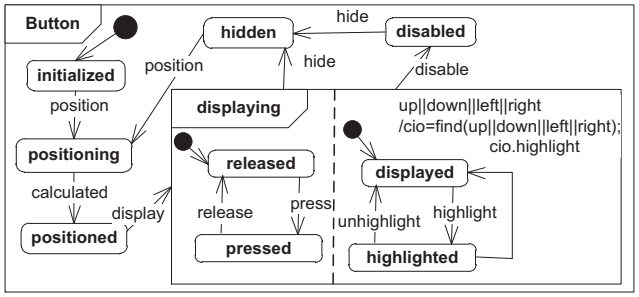

Figure 6. A graphical button interactor

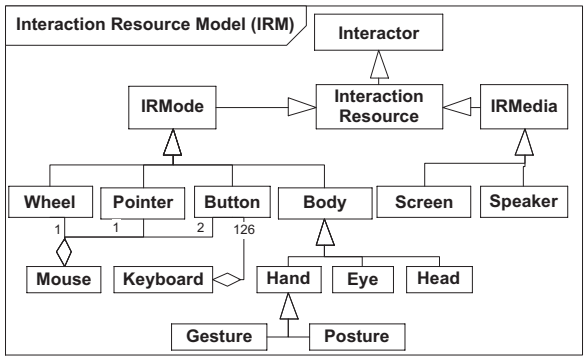

Figure 7. Part of the Interaction Resource Model

"displayed" as soon it is presented to the user. Thereafter it can be "disabled" or "hidden" from the interface.

A coordinate system enables directed navigation options by "up", "down", "left", and "right" events. But usually only little additional behavior needs to be added by the CIM model, since most of the overall behavior is already specified by the AIM. Examples for CIO interactors are a gauge meter that implements a continuous output or a button that represents a command (depicted by figure 6).

3.3 Interaction Resource Model (IRM)

Interactor resource interactors specify characteristics of physical or virtual devices that are used to interact with a system. Physical devices are for instance a mouse, a joystick, or a keyboard. Virtual devices describe software that controls hardware like a gesture or body movement recognition driven by a webcam or a light or a speaker connected to a home automation system.

Figure 7 shows a part of the IRM and depicts the relations between some of the interaction resources (IRs) that we have and will present in this section. Like for the other models, IRs are modeled as interactors. The IRM distinguishes between mode and media IRs. The former represent multimodal control options of the user, the latter specify output-only media.

Figure 8 depicts an aggregated IR representing a mouse. It is composed of a pointer, a button and a wheel. Different to physical devices like the mouse for which the IR specification is fixed and often simple, virtual device IRs are more complex and often are designed with a specific application in mind. Figure 9 depicts an example of a virtual IR that enables basic user interface navigation and selection of elements using the four different hand postures and movements shown on the left of figure 9. Green colored gloves and a webcam are used to capture the different

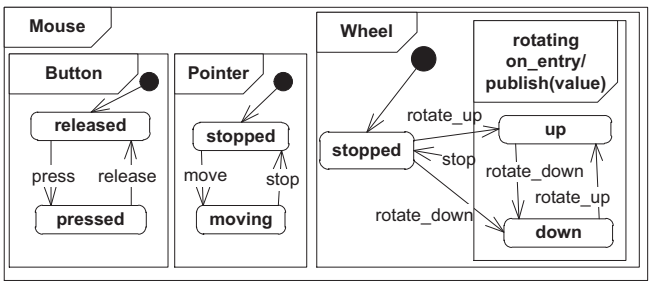

Figure 8. Mouse Interactor

postures and their movements in relation to the cam ("closer" and "farther"). By the state chart-based design specification design decisions can be captured in great detail: First of all, the navigation works with only one hand. Next, there is a fixed default speed of one second per element specified. Therefore, while the previous or next posture is shown, the focus in the interface moves to the previous or next element with each "tick". Then, there is a way to dynamically manipulate the navigation speed (see "movement" super state of figure 8), which replaces the default fixed ticker by considering hand movements (closer to or farther from the cam) to generate "tick" events. Finally, there is an element selection specified, that on the one hand stops the navigation as soon as the select posture is shown and on the other hand requires a second posture to actually confirm the selection (e.g. for security reasons).

Now that the three interface models have been presented they need to be linked together to actually form a dynamic multimodal interaction. Whereas abstract and concrete interactors are synchronized by mappings that are currently defined inside the state charts, the relations between different IRs and the abstract or concrete model are specified by multimodal mappings that we present in the following sub-section.

3.4 Mappings

We use mappings to connect the various models at system run-time and distinguish between two elementary types of mappings: multimodal mappings and synchronization mappings. The former ones a target of a typical application design process, as they are used to describe how an interaction resource (mode) is used to control an interaction. The latter one are used to synchronize the different levels of interactor abstraction and are pre-defined together with most of the interactors.

3.4.1 Multimodal Mappings

Multimodal mappings connect abstract or concrete interactors with interaction resources. They are defined on three different levels of abstraction using a custom, flow-like notation:

- *Application level*: A mapping that connects specific interactor instances; e.g. "If the 'confirm reservation' button is pressed then close the window."

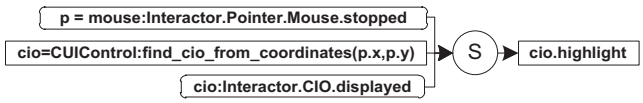

Figure 10. Basic pointer mapping.

- *Interactor level*: A mapping that is pre-defined together with the interactor designs ("the meta level") to be used later on for concrete application development. E.g. "If a button is pressed then play a 'click' sound".
- *Paradigm level*: A mapping that specifies an interaction paradigm, such as e.g. a "drag-and-drop" or a "double-click" that was prepared to work with the designed interactor set.

Figure 10 gives an example for a multimodal mapping that is defined at the interactor level between an IR (a pointer) and interface interactors on the concrete, graphical user interface level: It defines to change the highlighted graphical UI elements based on the current pointing position. It is worth mentioning that some IRs, like the pointer, can only be connected to the CUI model level and not to the abstract one. This is because the pointer is moved in a coordinate system and therefore depends on a graphical interface.

Mappings are specified by a custom flow chart like notation that offers three basic elements: observations, actions and operators. Boxes with rounded edges describe "observations" of state changes. Boxes with sharp edges are used to define actions, which are backend function calls or the triggering of events. The mapping observes the state of the pointer interactor and gets triggered as soon as the pointer enters the state "stopped". There are several operators available that we derived based on the findings of the CARE properties that are used to describe multimodal relations. The sequence operator "S" that is used in figure 10 defines a strict top-down sequence of observations and actions. Thus, the mapping of figure 10 waits for the mouse pointer to stop, then looks up the concrete interactor that is at the current mouse pointer positions and finally checks if this interactor is not already in highlighted and therefore in state "displayed" (compare with the CIO interactor of figure 6).

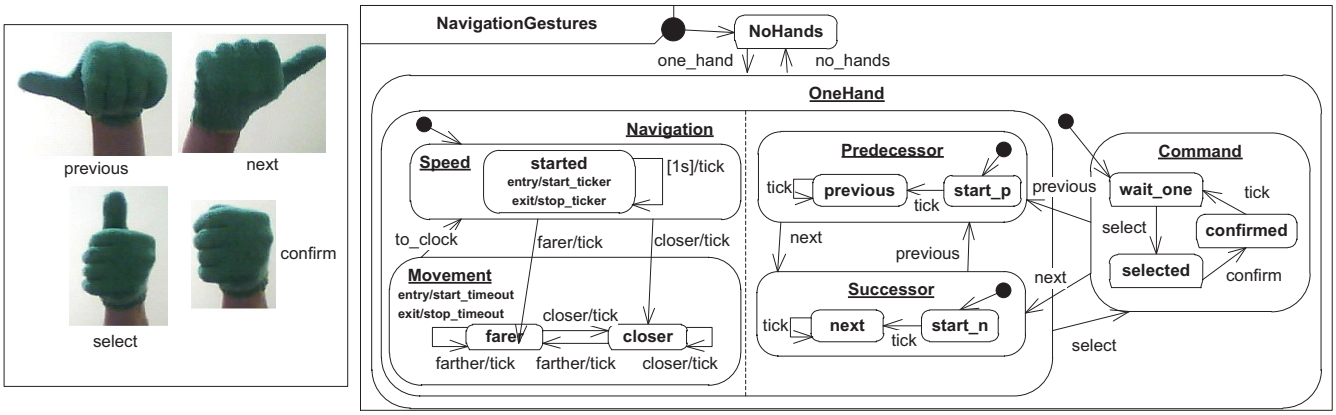

Figure 9. Used postures (left) and a gesture-based user interface navigation IR

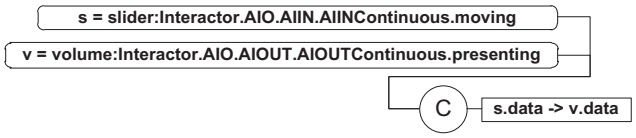

Figure 11. Application level mapping.

Mappings can fail and are then reinitialized automatically. Thus, the pointer mapping of figure 10 can fail for instance if no CIO can be found at the coordinates where the mouse pointer has been "stopped".

Figure 11 gives an example for an application level mapping that has been designed for a specific application in that a slider and a progress bar should be connected so that the slider movements are reflected by the progress bar. Even though the example mapping can be thought to be used for a concrete a slider and a progress bar the mapping itself is defined on the abstract, platform independent level by connecting an AIINContinuous with an AIOUTContinuous interactor (see figure 4). Thus, a "slider" could be as well a dimmer control, and the interactor named "volume" a dimmable light for instance.

The "C" operator that is used in this example defines a "complementary" relation. Therefore, the mapping initiates its action only if both observations are true at the same time: the slider interactor needs to be in state "moving" and the volume interactor in state "presenting". The complementary relation is continuous, which is defined by using lines to connect the operator "C" without arrows indicating a direction. A continuous mapping activates the actions as long as all operations are true and deactivates it otherwise. The mapping of figure 10 forwards all updates of the slider (stored inside the "data" variable") to the "data" variable of the "volume" interactor) as long as the slider is "moving" and the volume is in "presenting" state.

Finally, figure 12 depicts an example for a paradigm level mapping that defines a "drag-and-drop" to be used with an interaction resource that offers a pointer and a button, a mouse for instance. The pointer is implicitly included by using the mapping of figure 10 and a synchronization of "highlighting" to the abstract "focused" state. The Drag-and-Drop mapping utilizes temporary window (Tw) constraints to define the maximum timespan in which all observations are required to be evaluated to true. Further on, a fail statement can be attached to an operator to list the actions to recover from a failed observation. For the Drag-and-Drop such a fail statement is used to recover if the button is released while dragging without a target choice in focus. In this case the fail statement ensures that all dragged elements are re-added to the origin choice list.

3.4.2 Synchronization Mappings

Synchronization mappings are defined to propagate information about state changes between the different abstraction levels of one

Figure 13. Synchronization Mapping.

interactor. Mainly between it's abstract and it's several concrete model representations. Figure 13 depicts such a mapping that is specified in the same notation as a multimodal mapping. It ensures that a "highlighted" interactor of the concrete model (that has been highlighted e.g. by the mouse pointer (see pointer mapping, figure 10) is assumed to be in the user's focus in its abstract model representation.

In an earlier approach, we used "on_entry" and "on_exit" state actions to synchronize AIM with CIM interactors and did not use separate mappings for model synchronization. This had the advantage that all synchronizations get automatically added if an interactor is used.

The disadvantage was an un-wanted dependency of the AIM to the CIM models, which prevented introducing new concrete model without touching the existing AIM model.

In this section we gave an overview about the models we are using to design and combine multimodal interactors by an abstract, a concrete and a mapping model. Physical or virtual devices are specified by an interaction resource model. The next section introduces the MINT platform, which implements a run-time environment for the models and enables to directly execute them, without a transformation to source code.

4. MINT PLATFORM 2012

Model-driven design of software is performed by designing models and transforming them from platform independent to platform dependent ones. Usually this is a process that is performed during design-time before the application is started the first time and that ends with a final model-to-code transformation.

There are two main reasons, which made us decide to implement a run-time instead of a design-time process: First, different to software backend development, user interface development is a highly interactive and iterative process. Improving the efficiency in including changes in a single iteration would improve the overall development speed since changes can be discussed while giving the targeted audience the opportunity to experience the interface.

Second, modern user interfaces need to consider context changes: different groups of users, devices, environments, cultures, or modalities. Often all these potential context-changes cannot be considered during design-time; therefore problems often first occur during run-time when it is too late to adjust the interface using a traditional model-driven development approach.

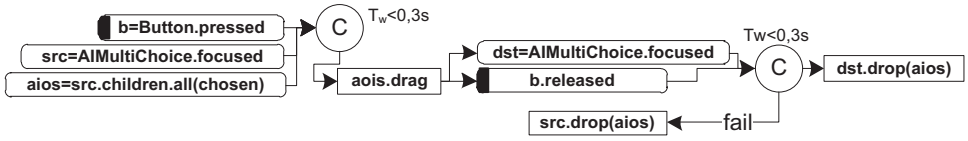

Figure 12. Drag-and-Drop paradigm level mapping.

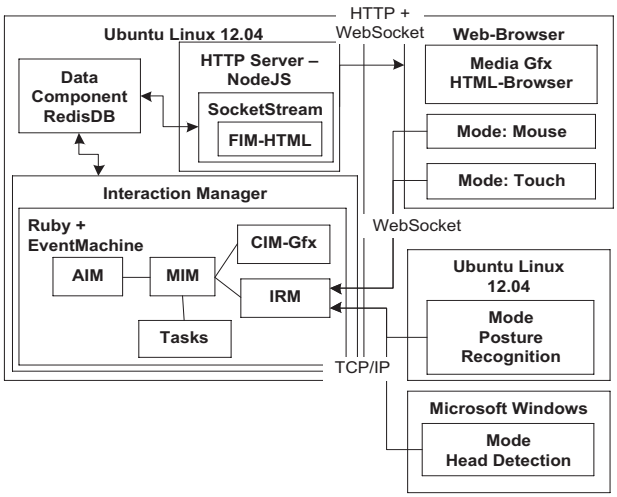

Figure 14. MINT 2012 Basic technical architecture.

Because of that, we implemented a model-based run-time environment (MRE) that is executed as part of a web-server. The main tasks of the MRE are 1) to instantiate the interactors of the Task, AIM, CIM and IRM as state machines, 2) to connect them through the mappings, and 3) to synchronize the modes and media utilized in an interaction by the user. The platform implements the structure of the W3C multimodal framework architecture (W3C-MMI) [17] and its basic component-based architecture is depicted in figure 14. Different to the W3C approach we strictly distinguish between mode and media. Thus, media components such as a web page are pure output components whereas modes implement components that handle user inputs. Therefore, even mouse and keyboard events are synchronized inside the interaction manager on server side instead of by the browser. The left part of the figure shows the server side that consists of a No-SQL data store with publish/subscribe functionality (RedisDB), an asynchronous HTTP Server written in Javascript (NodeJS) and an Interaction Manager that manages all the model instances written in ruby and the Event Machine Framework written in C that enables event-driven I/O based on the reactor pattern with high scalability. Since the main processing and fusion of the different connected modes and media happens on the server-side, all server-side functionality is processed completely asynchronous and event-driven.

Two components of figure 14 have not been discussed so far: The Tasks model in the interaction manager and the final HTML interface model (FIM-HTML) in the HTTP server. The former one is an implementation of a task model interpreter that can process a task tree specified in the ConcurTaskTree notation [12] and is used to define the overall process of an application. It distinguishes between three types of tasks: Interaction In, Interaction Out and Application tasks. Each task type has a behavior specification lifecycle that has been described earlier by the WebTaskModel [1]. The latter one, the final interface model (FIM), represents the graphical CIM model interactors by HTML templates and a widget library (we use jQuery).

The client side consists of the real interaction resources implementing modes and media, like a browser, head movement, posture and gesture recognition, a mouse or a multi-touch surface. User input from the modes is forwarded to the IRM, gets processed there and then is stored and published in the Redis database. The HTTP server subscribes itself to the database to receive relevant interactor updates and forwards them to the connected browsers.

5. Evaluation

Earlier works have suggested several requirements that have to be fulfilled for such a framework to be applicable. We will use these requirements (5.1) and classification criteria (5.2) to discuss the pros and cons of our work and proof the feasibility of a server-sided procession by an inital performance analysis (5.3). Further on, we indent to publish the 2012 version as open source as we did with the previous version [7] as well for others to proof and further evaluate our approach.

5.1 Requirements

As part of the W3C multimodal initiative, [17] has identified several requirements a multimodal toolkit should consider from a practitioner perspective that we discuss and relate in the following to our approach.

5.1.1 Structuration Mechanism

[15] notes that structuration mechanisms are required to "define the overall user interface composition" and mentions that there is currently no fixed overall model and no fixed model to start with as well.

Our approach uses reference integration through mappings that can be defined separately from the other models to link their elements together based on state change observations that can trigger events (actions). A current limitation of our approach is the synchronization mappings that sync between the different abstraction levels (e.g. between the abstract concrete model) of a single interactor. These are currently defined inside the interactor to reduce the amount of mappings that need to be created when designing an interface. This problem is known as the mapping problem [14].

5.1.2 Explicit Control Structures

[15] mentions explicit control structures as a crucial requirement. They are 1) hardly distinguishable from other facets, 2) difficult to track, and 3) difficult to locate if there are several threads of control.

We proposed a three-folded control structure: a macro-level and an inter-interactor as well as an intra-interactor control on the micro-level. The macro-level that specifies the overall control flow, we rely on a task model, where each task consists of a set of active interactor. On the micro-level finite state machines implement the control flow (the behavior) of individual interactors. Between interactors we use mappings to define their control relations.

5.1.3 Extensible Event Definition Mechanisms

[15] requires a multimodal toolkit to use events to connect interaction channels from different modalities and demands a mechanism that "should be extensible so that when a new input or output channel is created it is not necessary to rewrite the event model" [15].

Our approach defines an abstract interactor model (AIM) that specifies the minimal set of interactors that are required to be considered by a new modality (or a certain combination of modes) to allow the user a complete control of all interfaces that have been designed with our approach. Since we rely on SCXML

Table 1. Processing performance of pointer movement (n=250)

Device	Mean Time	Deviation	Variance
Server	9,39 ms	0,72 ms	0,57e-6
iPad Gen. 1 WLAN	14,39 ms	1,9 ms	3,6e-6
NoteBook WLAN	10,21 ms	1,1 ms	1,2e-6

based state charts interactor designs, the basic API is to observe these state changes or trigger them by events. An extension of the event definitions usually has to happen at the concrete model level. Thus, an abstract interactor could be extended to react on new events that are specific to a certain modality.

5.1.4 Data Modeling

[15] requires an explicit data model.

We distinguish between two types of data models. One that is used to capture the data from and present the information to the user and another one that is used by the functional backend of the applications. The former one is designed using class diagrams to express the data and additional functionality that is implemented by each interactor. The latter one depends on the application domain and could be a database, an enterprise server or web services for instance.

5.1.5 Reusable Components

User interfaces are made out of components containing visual parts, interactive behavior and control structures that repeat themselves in different parts of the interface [15].

In our approach interactors encapsulate these functionalities and represent interface elements as re-usable components. Interactors are designed once to specify a concrete way of interaction, like e.g. by a widget on a specific platform, a general way of interacting on the abstract model or by capturing the interaction capabilities of an interaction (as a resource interactor). Once these interactors have been designed they could be part of the palette of a classical user interface builder to enable a designer to assemble interface based on these interactors.

Mappings that glue together the interactors can be re-used as well. As interactor-level mappings they can be automatically established when the interactor is instantiated as part of an application's interface or even can specify overall interaction paradigms that implement a certain way of interacting with the application.

5.2 Programming Framework Characteristics

In a recent publication Dumas et. all. presented a survey of principles, models, and frameworks to create multimodal interfaces [6]. They classified existing frameworks based on their architecture trails, their ease of reusability and four further characteristics: extensibility, pluggability, reusable components, and availability as open source.

From the eight available frameworks to create multimodal interfaces, they state that only three have been made available as open source for others to proof and extend the authors work. During our initial investigations we figured out that the ICARE-Toolkit, differently than mentioned by the study, has not been made available by the authors. From the remaining two frameworks, only one is still being actively developed (Open Interface [10]), whereas the development of the other one (Papier-Mache) has stopped 7 years ago (according to the changes tracked in their open source repository).

Classifying our approach by the suggested criteria, the MINT platform is based on finite-state machine processing and still misses enhanced fusion mechanisms like frame support or symbolical-statistical fusion. It offers a low-level API in ruby, but supports higher-level programming features such as a graphical state-chart design based on the scxmlgui tool and a visual mapping editor to specify the mappings. We have not focus on supporting visual programming so far.

5.3 Processing Performance

Our main concern was the performance loss by processing all user action on the server-side that is required to fuse and distribute information from various connected devices. The most performance critical part of interface is the tracking and processing of user movements. For graphical interfaces this is the mouse pointer. To get an idea about the delay that is caused by the communication overhead, we tracked on the client side how long it takes to capture, send and receive back mouse pointer movements. The results are listed in table 1 for three different devices: (1) Our webserver running on a Intel Core2Duo P8600 notebook using a local browser, a first generation 1 Pad and an Intel Dual Core T2400 Notebook that are connected via WLAN to the server.

The results revealed that server-sided processing is a feasible approach when running the server on a router in a local network. For most applications the cursor tracking does not need to be pixel-precise (different to the pixel precise evaluation test, in our applications even a five pixel movement threshold was not discovered by the users) the delay can be further reduced.

The performance analysis experiment has been performed 500 times based by an automated test, while presenting an interface with 24 parallel running interface interactors, which we consider as medium complexity (like an email client main view for instance). Currently we have only implemented a very small set of interactors, but we intend to evaluate more complex interfaces like the Microsoft Word 2010 main interface view as future work.

6. CONCLUSION

This paper presents the MINT 2012 framework, a run-time platform enabling developers to create multi-modal interactors for the web. Research about model-based user interface design has been previously done to generate interfaces for different platforms. MBUID applies a structured process based on abstract-to-code transformations through several (transient) models. Multimodal interface generation has been considered by MBUI to a very limited extend only. Recent surveys revealed that only a handful of frameworks have been proposed. From those only two were made available for others to use and extend as open source and only one is still being actively developed.

We proposed a new approach based on a model-based design of interactors to represent multimodal interface elements. Interactors are well-known for user interface development and are considered as mature. We decided to use state charts and SCXML, an upcoming W3C standard, to specify the interactors. Different to other approaches all models are kept alive and get synchronized at run-time by mappings. This enables interactor inspection and introducing changes like switching modalities or adding new

devices at run-time and closes the often occurring gap between what has been designed and what has been implemented. Interactors run and communicate on the server side and all client-sided devices and modalities are synchronized with the server.

We have focused on a performance evaluation, because we think that it is one basic requirement for user acceptance (how fast a framework is able to process user inputs) and we haven't found performance analyses for other multimodal frameworks that we could proof. The current state of the framework is, that is enables the model-based creation of multimodal interfaces. To proof the efficiency and effectively of the development process we require development tools that we are currently working on.

7. ACKNOWLEDGMENTS

We would like to thank FAPESP for funding the travel grant that enabled Sebastian Feuerstack to attend the WebMedia 2012 conference and present the paper. Further on, Sebastian Feuerstack is grateful to the Deutsche Forschungsgemeinschaft (DFG) for the financial support of his work.

8. REFERENCES

[1] Bomsdorf, B. (2007), The WebTaskModel approach to web process modelling, *in* 'Proceedings of the 6th international conference on Task models and diagrams for user interface design', Springer-Verlag, Berlin, Heidelberg, pp. 240--253.

[2] Blumendorf, M. Lehmann, G. et al.: Executable Models for Human-Computer Interaction; in Interactive Systems. Design, Specification, and Verification: 15th International Workshop, DSV-IS 2008 Kingston, Canada, July 16-18, 2008; S. 238–251; Berlin, Heidelberg; 2008; Springer-Verlag.

[3] Calvary, G., Coutaz J., Thevenin D. ,Limbourg, Q., Bouillon L., and Vanderdonckt J. A unifying reference framework for multi-target user interfaces. Interacting with Computers, 15(3):289–308, 2003.

[4] Demeure, A., Calvary, G., and Coninx, K.: COMET(s), A Software Architecture Style and an Interactors Toolkit for Plastic User Interfaces; S. 225–237; 2008; Design, Specification, and Verification, 15th International Workshop, DSV-IS 2008, Springer Berlin / Heidelberg..

[5] Duke, D. Faconti, G., Harrison, M., and Paternó, F. Unifying views of interactors. In *AVI '94: Proceedings of the Workshop on Advanced Visual Interfaces*, pages 143–152, New York, NY, USA, 1994. ACM, ISBN:0-89791-733-2.

[6] Dumas, B.; Lalanne, D. and Oviatt, S. (2009), 'Multimodal interfaces: A survey of principles, models and frameworks', *Human Machine Interaction*, 3--26.

[7] Feuerstack, S. and Pizzolato, E.B. (2011), Building Multimodal Interfaces out of Executable, Model-based Interactors and Mappings, *in* J.A. Jacko, ed., 'Proceedings of the HCI International 2011; 14th International Conference on Human-Computer Interaction, Part I', Springer, Heidelberg (2011), Hilton Orlando Bonnet Creek, Orlando, Florida, USA., pp. pp. 221—228.

[8] Gonzalez-Calleros, J.; Vanderdonckt, J., and Muoz-Arteaga, J. (2009), A Structured Approach to Support 3D User Interface Development, *in* 'ACHI '09. Second International Conferences on Advances in Computer-Human Interactions, 2009.', pp. 75 -81.

[9] Harel, D. Statecharts: A visual formalism for complex systems. Sci. Comput. Program., 8(3):231–274, 1987.

[10] Lawson,J.-Y.L. , Al-Akkad, A.A., Vanderdonckt, J., and Macq, B. An open source workbench for prototyping multimodal interactions based on off-the-shelf heterogeneous components. In EICS '09: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems, pages 245–254, New York, NY, USA, 2009. ACM.

[11] Meskens, J. , Vermeulen, J., Luyten, K., and Coninx, K.. Gummy for multi-platform user interface designs: shape multiply fix use me. In Proceedings of the working conference on Advanced visual interfaces, AVI '08, pages 233–240, New York, NY, USA, 2008. ACM.

[12] Mori, G., Paternò, F., and Santoro, C.. Ctte: Support for developing and analyzing task models for interactive system design. IEEE Trans. Software Eng., 28(8):797–813, 2002.

[13] Nigay, L., and Coutaz, J.: Multifeature Systems: The CARE Properties and Their Impact on Software Design; in Intelligence and Multimodality in Multimedia Interfaces; 1997.

[14] Limbourg Q., and Vanderdonckt, J.(2004), Addressing the mapping problem in user interface design with UsiXML, *in* 'TAMODIA '04: Proceedings of the 3rd annual conference on Task models and diagrams', ACM Press, New York, NY, USA, pp. 155--163.Jean Vanderdonckt and Pierre Berquin. Towards a very large model-based approach for user interface development. In UIDIS, pages 76–85, 1999.

[15] Sire, S., Chatty, C.: The Markup Way to Multimodal Toolkits. In: W3C Multimodal Interaction Workshop (2002)

[16] Stanciulescu,A. (2008), 'A Methodology for Developing Multimodal User Interfaces of Information Systems', PhD thesis, Université Catholique de Louvain.

[17] W3C. Multimodal architecture and interfaces. http://www.w3.org/tr/2011/wd-mmi-arch-20110125/, last checked may 21, 2012, 2012.