A Unifying Reference Framework and Model for Adaptive Distributed Hybrid User Interfaces

Audrey Sanctorum and Beat Signer
Web & Information Systems Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{asanctor,bsigner}@vub.be

***Abstract*—Over the last decade, research on adaptive and distributed user interfaces (DUIs) has increased. We also witness a growing number of Internet of Things (IoT) devices, allowing digital user interfaces (UIs) to communicate with physical objects and vice versa through so-called hybrid user interfaces. There exist various solutions to manage adaptive, distributed or hybrid UIs. However, none of them covers all three aspects and users have to deal with multiple applications and configurations when developing adaptive distributed hybrid user interfaces. We introduce the eSPACE reference framework and conceptual model unifying the domains of adaptive, distributed and hybrid interfaces. While our reference framework has been inspired by the CAMELEON reference framework, the conceptual model is based on the Resource-Selector-Link (RSL) hypermedia metamodel. We propose an approach for adaptive distributed hybrid user interfaces where users can author their user interfaces based on the different levels of abstraction introduced by our reference framework. We further present a use case illustrating the extensibility, flexibility and reusability offered by our unified approach and discuss potential future work.**

***Index Terms*—Adaptive user interfaces, distributed user interfaces, hybrid user interfaces, internet of things, end-user development, model-based approach**

I. Introduction

User interface design and development is a difficult and time-consuming task [1]. With the rise of new kinds of devices, input and output capabilities as well as user needs, the task has become even more complex. While the *Design for All* approach [2] might fit a majority of users in a traditional context, it does not adapt to recent use cases and often leads to a bad user experience [3].

Further, with the rise of smart objects or *things*, the scope of user interfaces (UIs) has grown from digital UIs to physical UIs. We refer to the combination of digital and physical UIs as *hybrid UIs*. Given that we are living in an era where UIs for smart environments (including IoT and tangible systems) are gaining in popularity, physical UIs should be integrated into our UI models. For this reason, we promote the *unification*, *creation* and *customisation* of adaptive, distributed and hybrid UIs via a model-based approach and aim to reduce the effort required to develop such interfaces by following an authoring rather than programming approach.

The research of Audrey Sanctorum is funded by a PhD grant of the Research Foundation Flanders (FWO).

II. Background

In the late 1980s an emerging number of modelling languages to create rich and accurate interface models led to *model-based user interface development (MBUID)* [4], reducing UI development and maintenance costs based on a layered architecture [5]. Szekely [6] introduced a generic MBUID architecture with the typical components of model-based interface development environments (MB-IDE), including modelling tools, the model, automated design tools and implementation tools. The model organises information on three levels, including the *task and domain model*, the *abstract user interface* specification and the *concrete user interface (CUI)* level. A CUI can further be translated into an executable representation (linked to some application logic) on the *final user interface (FUI)* [7] abstraction level.

Even though modelling languages became more expressive, due to the continuous evolution of UIs, model-based user interface development remains a challenge [4]. Two decades ago, Szekely [6] stated a number of challenges, including *task-centred interfaces*, *multi-platform support* and *interface tailoring*, which are still valid today. Another challenge is the support for *multimodal interfaces* to develop richer UIs.

By taking Szekely's challenges into account, we focus on MBUID in the domain of adaptive and adaptable UIs, distributed UIs (DUIs) as well as hybrid UIs. Adaptive and adaptable UIs address the first and third challenge, while DUIs tackle the second challenge. The third domain has been chosen since we think that an additional important challenge for MBUID are hybrid UIs in order to deal with IoT solutions that were not yet present when Szekely introduced his challenges.

Model-based adaptive and adaptable user interfaces aim to improve the user experience by adapting to a user's context of use at runtime [3]. Hereby, the context of use is defined by a triple consisting of the *user*, *platform* and *environment* entities [8]. While adaptive UIs automatically react to changes in context, adaptable UIs are tailored according to predefined options but usually require explicit human intervention. A key goal of adaptive UIs is *plasticity*, meaning that a UI preserves its usability across multiple contexts of use [7].

The first multi-context reference framework applying a model-based approach was introduced by Calvary et al. [7].

An extended version later gave rise to the CAMELEON reference framework (CRF) [8]. CRF structures the development life cycle into four levels of abstraction and supports UI plasticity. Following these abstraction levels, approaches to develop adaptive UIs such as Akiki et al.'s IDE [9], Paternò et al.'s tools [10] and research based on UsiXML [11] emerged. Inspired by CRF and multiple research domains, the Multi-Access Service Platform (MASP) [12] addresses the deployment and runtime issues when developing adaptable, distributed and multimodal UIs for smart environments. While adaptive and adaptable UIs may adapt to a platform, they do not support the creation of UIs over a set of heterogeneous cooperating devices. Further, the majority of existing solutions is less flexible in terms of the UI granularity since it is impossible to show only parts of a UI on a device.

With the growing amount of devices, research on *model-based distributed UIs (DUIs)* tries to take advantage of these cross-device ecosystems. Elmqvist [13] defined a DUI as *"a user interface whose components are distributed across one or more of the dimensions input, output, platform, space, and time"*. Balme et al. [14] presented CAMELEON-RT, an architecture reference model for DUIs which can be used to compare and reason about existing as well as to create new runtime infrastructures for distributed, migratable and plastic UIs. Some adaptive UI approaches evolved towards systems that include DUI support, such as the distributed MARIA language [15]. Melchior et al. [16] proposed DUI support at different levels of granularity, including action/service, widget, windows and application level. In contrast to Manca and Paternò [15], Melchior et al. focus on GUIs and do not provide support for other modalities such as voice. MASP provides support for UI distribution with limited multi-user interaction and no control of the distribution granularity [17].

The rise of smart devices such as smart power plugs, light bulbs or thermostats led to dedicated applications to control these devices. Further, the idea of adding physical objects to interactive environments shows many benefits, such as intuitiveness, ease of learning and naturalness. Therefore, smart things should be considered in the UI development process via *model-based hybrid user interfaces*. Reference architectures defining a set of building blocks and addressing requirements for IoT environments have been proposed. Essential requirements are interoperability, device management, dynamic discovery, context-awareness, scalability, large volume data management, data security, integrity and privacy as well as dynamic adaptation [18].

While most existing approaches omitted support for runtime IoT adaptation, it is addressed by Hussein et al.'s [19] work on adaptive IoT systems and by the MDE4IoT framework [20]. The integration of physical objects is supported in a development framework for physical UIs by Varela et al. [21]. Further, Coyette and Vanderdonckt [22] focussed on the prototyping of UIs combining the digital and physical worlds.

Various systems have been developed to support model-based user interface development. Some approaches cover parts of the UI development process by providing tools to support specific abstraction levels of the CRF, while others, such as Cedar Studio IDE [9], CAMELEON-RT or MASP focus on the whole development cycle. Most of the presented solutions focus either on adaptive, distributed or physical UIs, with some exceptions such as MASP, distributed MARIA and Dandelion [21] spanning over multiple domains; namely adaptive and DUIs for the first two and DUIs and physical UIs for the latter. Nevertheless, none of the existing systems covers *all three* closely-related domains and provides support for adaptive, distributed and hybrid UIs. In addition, the *extensibility* of adaptive and distributed behaviour is limited or completely absent and most systems provide a fixed set of distributable elements. Only Cedar Studio [9] fully supports the extensibility of adaptive behaviour. Most approaches also lack *flexibility* with respect to adaptation and distribution rules, distribution granularity in DUI systems and dynamic adaptation in IoT systems [18]. The *reusability* of model components, rules, UI parts and distributed settings is often not supported. One of the few systems providing a fine distribution granularity and high reusability has been presented by Melchior [17] but it is limited to GUIs only. Last but not least, existing solutions mainly address designers and developers, with less support for *end-user development*.

III. Use Case

In order to illustrate the potential of our unified approach for adaptive distributed hybrid UIs, we present a use case which is currently not supported by existing MBUID solutions. Lucas wakes up around 7:30 a.m. every morning. For supporting his daily morning routine, he developed a *leaving home application* running on a smartwatch as shown in the mock-ups in Figure III.1. Once Lucas leaves and locks the front door, the application checks whether the clothes iron, lights and TV are turned off and—if one of them is still on—sends a message to Lucas, who can turn the device off via the smartwatch user interface consisting of the three parts shown in Figure III.1.

Lucas' second application is the *cats application*. He has two cats for whom he bought a smart feeder, a pet Wi-Fi camera, and a customised food storage box with LEDs, a weight sensor and a push button. The food storage box glows green when full and red when almost empty to remind Lucas when to buy food. When pressing the push button, cat food is added to his grocery list. With his self-made application shown in Figure III.1, he can control the feeding time and see his cats when he is not at home via the video stream, which also contains a record/snapshot button to record a video or take a picture. He also added a button to access the image gallery containing the captured images and videos.

Further, Lucas installed a motion sensor behind the couch to get notified when the cats potentially scratch the couch in order that he can activate a water spray to stop them. A temperature GUI component can be used to set the temperature in the apartment. Once the temperature drops to 16 degrees Celsius, Lucas gets a notification. An *administration panel* is used to edit the applications, manage their rules and distribution properties as well as the user rights.

Fig. III.1: Mock-up of Lucy's *cats application* (smartphone), Lucas' version (tablet), his *leaving home application* (smartwatch) and the administration panel

Today Lucas is heading to work and normally his *leaving home application* would notify him that he forgot to turn off the lights. However, since Lucas' girlfriend Lucy moved in, they made some changes to Lucas' applications so that Lucy can use them as well. She created a profile with her preferences and adapted some of his applications to her needs via the administration panel. Given that Lucy leaves for work later, Lucas' *leaving home application* did not check whether the appliances are turned off. Later Lucy leaves and gets notified that the TV is still on. She opens the *leaving home application* on her phone—which adapts to the new device as well as to her preferences—and turns off the TV. Note that Lucy is colour blind and therefore her application shows blue and orange buttons instead of the green and red buttons.

During the day, Lucas receives a notification that the cats are scratching the couch. After a quick look at the video stream of the *cats application*, he realises that the cats are just playing near the couch and presses the `funny` button. The `funny` button has been added by the couple to notify each other about funny content and is shown in the tablet and phone mock-ups in Figure III.1 (smiling cat image). Whenever Lucas presses this button, Lucy receives a notification. As illustrated in the smartphone mock-up, Lucy's UI of the *cats application* only contains the video stream and the `funny` as well as a `gallery` button. Further, Lucy shared the video stream UI element with her best friend and whenever she presses the `funny` button, she notifies Lucas as well as her best friend.

Once home, Lucas fills the feeder with cat food and recognises that the food box is glowing red. Therefore, he presses the button on the box to add "cat food" to his grocery list. Later, the couple shows each other pictures by displaying them on the TV by using a *touch-throw* gesture on their phone.

IV. eSPACE Reference Framework

In the following we present our multi-layered *end-user Smart PlACE (eSPACE)* reference framework shown in Figure IV.1, which unifies adaptive, distributed and hybrid user interfaces. The different layers of the CAMELEON reference framework have been used as inspiration to divide and structure our reference framework into similar abstraction layers. Unlike CRF, we are not aiming for a purely MBUID process but want to make the most of both worlds by combining manual and model-based techniques. To promote flexibility, extensibility and reusability of the different components of

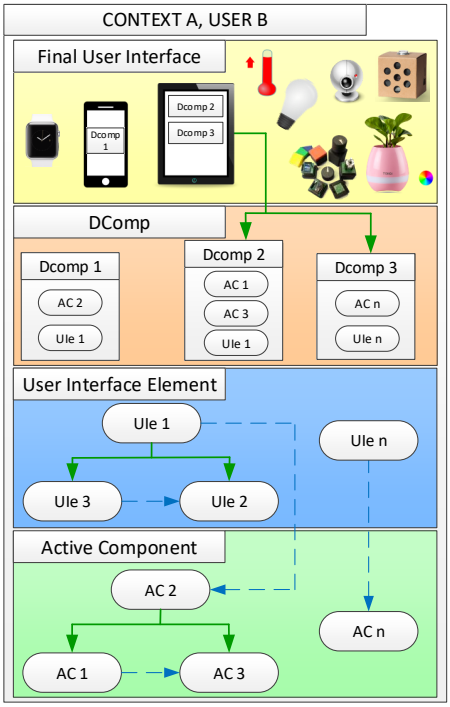


Fig. IV.1: eSPACE reference framework

a UI, we provide a loose coupling between UI elements and their tasks by placing them in separate layers. The layer containing the "tasks" is called the *active component* layer. Tasks are represented as active components (ACs), a concept introduced by Signer and Norrie [23]. Each AC represents a piece of program code that executes an action, such as *turning on the light*. ACs can be linked to each other to execute more complex interactions, as shown by the blue dashed arrow in Figure IV.1. `AC 1` could, for example, be a scratching detector AC linked to an AC that sends notifications so that Lucas is notified when a cat might be scratching his couch. To reuse this functionality, a new AC can be created composed out of these two ACs, as shown with `AC 2` that is composed of `AC 1` and `AC 3`. If Lucas wants to reuse this scratching notifier in other applications, he can just refer to it as `AC 2`. The `notifier` AC is a generic notifier that is reusable in other applications with different configurations.

The next layer containing UI elements (UIes) such as text fields, buttons, video frames or a combination of different ele-

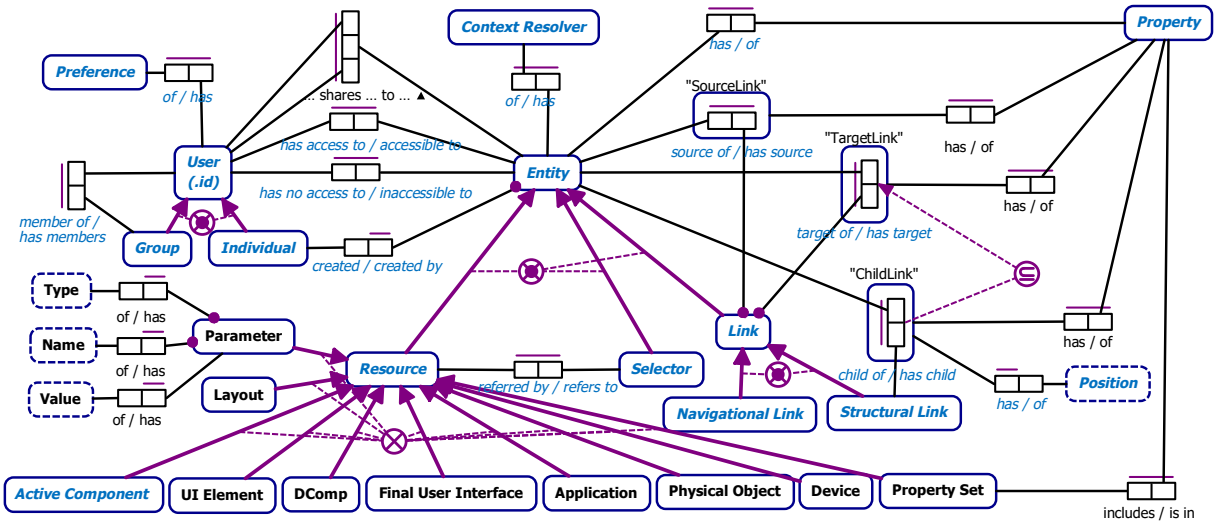


Fig. V.1: Conceptual model based on RSL

ments is called the *user interface element* layer. A UIe can also be a physical UI, such as a light switch, a physical button or a slider. Similar to the previous layer, different UI elements can be linked together to form more complex UIes. One can use links to navigate between different UI views, such as in Lucas' cats application where when he presses the `gallery` button, a link is followed to the `photo gallery` view. Links can also be used to arrange UI elements in one view. For example, a video player can be created by linking a `video frame` UI element with a `button` UI element. This composition can be saved as a `video player` UIe, which can be reused in different applications. Compositions are represented by green solid arrows. Figure IV.1 illustrates `UIe 1` as a composition of `UIe 2` and `UIe 3`. Last but not least, links can be used to bind some functionality to UI elements by linking them to ACs. For example, linking a `button` UIe to an AC that turns on the light, will enable the button to act as a light switch.

The third layer is the *DComp* layer consisting of *distributed components (DComps)*. These DComps consist of the "models" created by linking ACs and UI elements, together with the configuration (metadata) for the corresponding ACs and UI elements. Each DComp holds at least one AC or UI element and is distributable in real time. Lucas' cat application consists of multiple DComps, one of them for the live stream component used to watch his cats. This DComp holds a `video player` UIe that is linked to a `show stream` AC. Note that the DComp can be distributed to any another device, such as a smartphone, in order to see the video stream. Distribution can happen at different levels of granularity. For instance, Lucas could distribute the `capture/record` button to his smartwatch for recording the cats by using his watch. Since DComps are device independent, this layer can be compared to the CUI layer of the CRF, with the difference that DComps do not have to include any layout information.

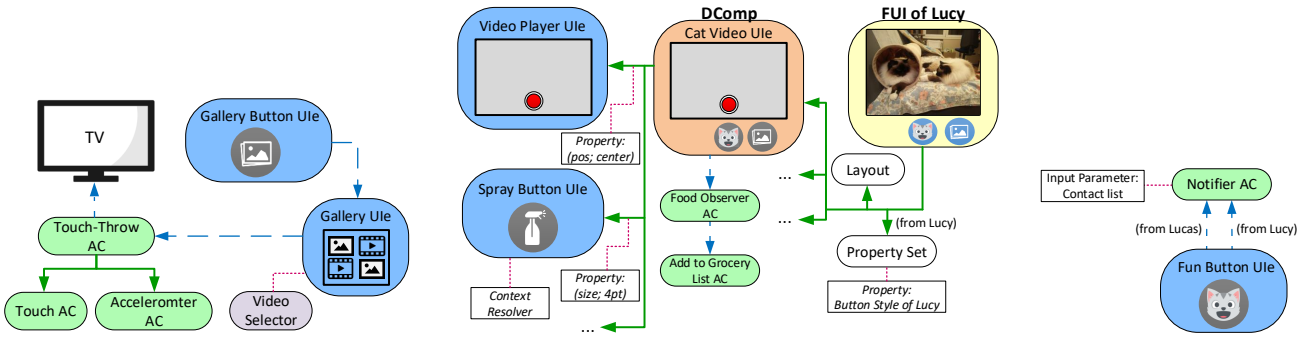
The last layer represents the *final user interface* which, similarly to the CRF, contains the final user interfaces. Once a DComp is attached to a device, it will get its look and feel depending on this device and become part of the final user interface (FUI). Multiple DComps can be placed on a device with a certain layout and be saved in one FUI. For example, the leaving home application FUI contains three simple DComps to control the lights, the TV and the clothes iron. A FUI can still be adapted depending on the context of use. If Lucy uses Lucas' smartwatch (she has to be logged in), the buttons of the *leaving home application* adapt to her colour preferences.

Our framework supports the *reusability* of its components over multiple applications and endorses *flexibility* and *extendibility* by allowing these components to be linked together for navigation and to create new compositions. We further promote end-user development by allowing UIes to be created and customised by simply linking different components.

V. Conceptual Model

Our conceptual model for adaptive distributed hybrid user interfaces is based on the Resource-Selector-Link (RSL) hypermedia metamodel [24] and its concept of active components [23]. RSL was chosen due to its support for the loose coupling of resources via different types of links and offers features for user and context management. The main components of the RSL metamodel are the `Resource`, `Selector` and `Link` entities which are subtypes of the `Entity` type as shown in Figure V.1 (original RSL components are coloured in blue). Note that while we use the ORM modelling language [25] for our conceptual model, the original RSL metamodel is based on the OM data model [24].

In the previous section we expressed the need to compose ACs and UI elements as well as to navigate between them. This is achieved by using `Structural Links` and `Navigational Links` (solid green and dashed blue arrows in Figure IV.1). For example, we can link ACs via navigational links to execute multiple actions after each other or UIs can be linked to navigate between views, as shown for the `gallery` button and `gallery` view in Figure V.2a. Further, navigational links can be used to link UI elements



(a) AC Composition and Image Selector (b) Lucy's FUI composition (c) Notifier AC

Fig. V.2: Examples of conceptual model instantiations

to ACs. By using structural links, one can create structures as illustrated in Figure V.2a. They can be used to compose UIs containing multiple UI elements, such as the `video player` UIe that is composed of a `button` UIe and a `video frame` UIe. New ACs with combined functionality can be created as well (e.g. linking a `touch` AC with a `accelerometer` AC to create a `touch-throw` AC used to move pictures to the smart TV). RSL `Selectors` can be used to refer to a piece of a resource. A `video` selector can, for example, be used to transfer parts of a video rather than the entire video from the `gallery` view and thus enables video excerpts to be distributed to the TV, as shown in Figure V.2a. Note that, instead of using multiple links to structure ACs and UIs, we use RSL's multi-targeted links. The children of a structural link are ordered via a position attribute and the `child of` relationship is a subset of the `target of` relationship (via a pair-subset constraint in the ORM scheme).

Structural links are further used to structure `DComps` and `FUIs` as illustrated in Figure V.2b. Depending on the context of use, the composition of the `DComps` and `FUIs` might be different. Context can be modelled by using the `Context Resolvers` provided by RSL, which act as gateways. The `water spray` button has a context resolver which verifies at run-time who is using the application. If Lucy is using the application, her `FUI` will not contain this button. Context resolvers can also be used on links, to set conditions on when a link needs to be followed.

Entities in RSL can have multiple `Properties`, which are key-value pairs. They might, for example, be attached to a UI element to define its size. In addition, RSL offers user management, where users have preferences and can be part of a group. Each group or individual user can be granted or denied access to specific entities.

In order to define properties over the individual sources or targets of a link, we objectified these relationships and added a `has` relationship. This is useful to have distinct properties on the link to every source or target of many-to-many links, as shown on the structural link with as targets the `video player` UIe and `spray` button. There the property is used to set the position of the UI elements for this specific composition. In addition, we added a `sharing` relationship between `User` and `Entity` to enable users to share their entities with each other. This enables extensibility via a growing shared entity set as users add their self-created `ACs`, `UI Elements` and `DComps`.

Based on our reference framework, we differentiate between different types of resources in our model. As shown in Figure V.1, we have the `Application`, `DComp`, `Active Component`, `Final User Interface`, `UI Element`, `Parameter`, `Property Set`, `Layout`, `Device` and `Physical Object` resources. An `Application` consists of one or more `FUIs`, which in turn are composed of at least one `DComp` that contains at least one `AC` or `UI Element`. ACs can be used to verify information, such as *"is the door locked?"* or to perform an action on a `Device` or `Physical Object`, such as *"turn the light off"*.

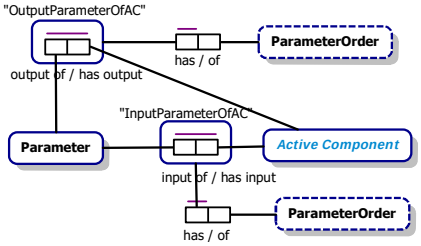


Fig. V.3: AC parameters model

The `Parameter` resource is used to represent the input and output parameters of `ACs` and `UI Elements` as illustrated in Figure V.3 for the model ofs `AC` parameters. Each `Parameter` contains a name, type and value. The `notifier` AC has a `contact list` as input parameter as shown in Figure V.2c. When Lucy logs in, the navigational link is triggered, instantiating the AC with her contacts as input parameter. Further, the `Layout` resource is used to set the layout style, such as grid or column layout and the `Property Set` is used to group properties. These are mainly used to hold styling properties as shown in Figure V.2b, where the `FUI` is composed of `DComps` with a certain `Layout` and user-specific `Property Set`.

With our conceptual model we show how *reusability*, *flexibility* and *extensibility* can be provided using the RSL link service, the introduced resources and concepts. In contrast to

existing systems, we do not distinguish between physical and digital UIs in our core model. Finally, we want to promote end-user development by allowing users to create UIs by simply linking components.

VI. Discussion and Future Work

We have proposed a new unifying approach for the development of user-defined adaptive distributed hybrid UIs where users can create smart UIs based on the strengths of both model-based and manual techniques. A major issue of MBUID is the complexity of the models and notations which makes the tools hard to learn and use [5]. Therefore, we reduced the modelling to a single expressive model where users can link ACs and UIes to form a user interface. However, this is still complex for end users and more work is needed to investigate the appropriate metaphors for hiding the complexity of the underlying model. Our goal is to include end users in the UI development process. For this reason we plan to create an end user authoring tool based on the concepts introduced in this paper.

From related work we have seen many existing systems for adaptive, distributed or hybrid UIs. However, none of them supports the creation of UIs that are adaptive, distributed as well as hybrid and we would have to use a combination of the existing systems with different models, setups and configurations for each of them. In order to simplify this process, we introduced a *unified approach* supporting *all* three aspects. Therefore, as a first contribution, we proposed the eSPACE *reference framework*, which helps structuring the development process of smart UIs and supports runtime adaptation and distribution as well as physical UIs. Further, we have seen that most of the existing systems are limited in terms of *flexibility*, *customisation*, *extensibility*, *distribution granularity* and *reusability*. We tackle *all* these limitations with our *conceptual model* that is based on the RSL hypermedia metamodel. The potential of our model has been illustrated with some examples. Our model can adapt to evolving user needs, technologies and environments by introducing new `Preferences`, `ACs`, `Devices` and `Physical Objects`, which shows the flexibility and extensibility of our approach. Based on our conceptual model, adaptive distributed hybrid UIs which support distribution at a fine level of granularity can be created by using `DComps` and `Selectors`. In addition, our model supports runtime adaptation based on `Context Resolvers` and a user's `Preferences`. By using an RSL-based link service, our components can easily be reused by simply linking to them. This allows for new compositions and hereby promotes extendibility and reuse of adaptation and distribution rules which is not offered by most existing work. While we have a functional implementation based on the presented model, in future work we plan to provide a development framework and end-user authoring tool as well as a proof-of-concept application. Finally, with our reference framework and conceptual model, we want to support the community in providing a new way of thinking about user interface development for adaptive distributed hybrid UIs.

References

[1] B. A. Myers and M. B. Rosson, "Survey on User Interface Programming," in *Proc. of CHI 1992*, June 1992.

[2] C. Stephanidis, "Towards the Next Generation of UIST: Developing for All Users," in *Proc. of HCI 1997*, Aug. 1997.

[3] P. A. Akiki, A. K. Bandara, and Y. Yu, "Adaptive Model-Driven User Interface Development Systems," *ACM Computing Surveys*, vol. 47, no. 1, May 2014.

[4] G. Meixner, F. Paternò, and J. Vanderdonckt, "Past, Present, and Future of Model-based User Interface Development," *Journal of Interactive Media*, vol. 10, no. 3, 2011.

[5] P. P. Da Silva, "User Interface Declarative Models and Development Environments: A Survey," in *Proc. of DSV-IS 2000*, June 2000.

[6] P. Szekely, "Retrospective and Challenges for Model-based Interface Development," in *Proc. of DSV-IS 1996*, June 1996.

[7] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt, "A Unifying Reference Framework for Multi-Target User Interfaces," *Interacting with Computers*, vol. 15, no. 3, 2003.

[8] G. Calvary, J. Coutaz, L. Bouillon, M. Florins, O. Limbourg, L. Marucci, F. Paternò, C. Santoro, N. Souchon, D. Thevenin, and J. Vanderdonckt, "The CAMELEON Reference Framework," CAMELEON Project Deliverable 1.1, September 2002.

[9] P. A. Akiki, A. K. Bandara, and Y. Yu, "Cedar Studio: an IDE Supporting Adaptive Model-Driven User Interfaces for Enterprise Applications," in *Proc. of EICS 2013*, June 2013.

[10] F. Paternò, C. Santoro, and L. D. Spano, "MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments," *ACM Transactions on Computer-Human Interaction*, vol. 16, no. 4, 2009.

[11] J. Vanderdonckt, "A MDA-compliant Environment for Developing User Interfaces of Information Systems," in *Proc. of CAiSE 2005*, June 2005.

[12] M. Blumendorf, "Multimodal Interaction in Smart Environments: A Model-based Runtime System for Ubiquitous User Interfaces," Ph.D. dissertation, Berlin Institute of Technology, 2009.

[13] N. Elmqvist, "Distributed User Interfaces: State of the Art," in *Distributed User Interfaces: Designing Interfaces for the Distributed Ecosystem*, ser. Human-Computer Interaction Series, 2011.

[14] L. Balme, A. Demeure, N. Barralon, J. Coutaz, and G. Calvary, "CAMELEON-RT: A Software Architecture Reference Model for Distributed, Migratable, and Plastic User Interfaces," in *Proc. of EUSAI 2004*, Nov. 2004.

[15] M. Manca and F. Paternò, "Extending MARIA to Support Distributed User Interfaces," in *Distributed User Interfaces: Designing Interfaces for the Distributed Ecosystem*, 2011.

[16] J. Melchior, J. Vanderdonckt, and P. V. Roy, "A Model-based Approach for Distributed User Interfaces," in *Proc. of EICS 2011*, June 2011.

[17] J. Melchior, "A Model-based Approach for Dynamically Distributing Graphical User Interfaces Based on their Properties, Graphs, and Scenarios." Ph.D. dissertation, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2016.

[18] E. Cavalcante, M. P. Alves, T. Batista, F. C. Delicato, and P. F. Pires, "An Analysis of Reference Architectures for the Internet of Things," in *Proc. of CobRA 2015*, May 2015.

[19] M. Hussein, S. Li, and A. Radermacher, "Model-Driven Development of Adaptive IoT Systems," in *Proc. of MODELS 2017*, Sep. 2017.

[20] Federico Ciccozzi and Romina Spalazzese, "MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering," in *Proc. of IDC 2016*, Oct. 2016.

[21] G. Varela, A. Paz-Lopez, J. A. Becerra, and R. J. Duro, "The Generic Interaction Protocol: Increasing Portability of Distributed Physical User Interfaces," *Romanian Journal of Human-Computer Interaction*, vol. 6, no. 3, 2013.

[22] A. Coyette and J. Vanderdonckt, "Prototyping Digital, Physical, and Mixed User Interfaces by Sketching," in *Workshop on User Interface eXtensible Markup Language UsiXML*, France, Paris, June 2010.

[23] B. Signer and M. C. Norrie, "Active Components as a Method for Coupling Data and Services: A Database-Driven Application Development Process," in *Proc. of ICOODB 2009*, July 2009.

[24] B. Signer and M. C. Norrie, "As We May Link: A General Metamodel for Hypermedia Systems," in *Proc. of ER 2007*, Nov. 2007.

[25] T. Halpin and T. Morgan, *Information Modeling and Relational Databases*. Morgan Kaufmann, 2010.