Detaching Control from Data Models in Model-Based Generation of User Interfaces

Giorgio Brajnik[1,2(✉)] and Simon Harper[1]

[1] Computer Science School, University of Manchester, Manchester, UK
simon.harper@manchester.ac.uk
[2] Dipartimento di Matematica e Informatica, Università di Udine, Udine, Italy
brajnik@uniud.it

Abstract. A strength of IFML derives from its ability to support generation of a user interface by coupling data and control models. However, separation of concerns between different models (and in particular between models of control and of data) could be beneficial to better understand generation principles and expressivity limits, to support computation of design quality metrics, and to formulate intra-model transformation rules.

In the paper we show that such a separation is indeed possible using UML class and state diagrams. We present the generation rules that a compiler follows for producing user interfaces. Based on the adopted representation, we argue that certain expressive limits are due to the underlying foundation common of our approach and of IFML.

Keywords: Model-based user interface generation · Statecharts · UML · IFML · Low fidelity prototypes

1 Introduction

Model-based approaches for designing and developing user interfaces (UIs) often use platform independent meta-models that provide means to abstract data and control aspects. One of these approaches is Interaction Flow Modeling Language (IFML), an OMG standard aimed at modeling rich user interfaces [2] that is being used by industry to support automatic generation of web applications. The strength of IFML derives from its ability to support automatic generation of a web application and its user interface; this is made possible because it can express the abstract behavior of a UI through a very tight integration of data view components with a state transition representation.

In this paper we tackle the problem of detaching control from data specifications as they are used in IFML and see if, by providing a looser relationship between them, we can preserve the same expressive power and the ability to automatically generate UIs. This "separation of concerns" principle is often mentioned in the model-driven UI literature.

The are several reasons for doing so. A separate model of control helps understanding the conceptual framework upon which the generation principles can be

P. Cimiano et al. (Eds.): ICWE 2015, LNCS 9114, pp. 697–700, 2015.
DOI: 10.1007/978-3-319-19890-3_60

formulated; it can also help highlighting expressive limits, their reasons and possible solutions. A separate model of control can also be used in instrumenting the generation tool so that graph-theoretic metrics can be produced that bear upon usability (as suggested in [3,4]). These metrics would make interfaces and usability quantifiable (at least in part), and therefore would enable a data-driven approach to UI design that allows well founded comparisons of alternative designs. Finally, a separate model of control helps understanding if an algebra of models can be defined so that new models are automatically derived from existing ones. This could be extremely useful for deriving, for example, a mobile UI by applying certain operators on a model of a desktop UI.

However, at the moment, it is not clear if such a separation is possible without loosing the strong expressive power that IFML features. In the paper we show that such a separation is indeed possible, and that it can be achieved using UML class and state diagrams. Based on the adopted representation, we argue that certain expressive limits are due to the underlying foundation, that is common of our approach and of IFML. Limits deal with the inability to handle arbitrary undo/redo, to handle customizable toolbars, and to handle arbitrary instantiations of widgets.

2 Conceptual View of UICompiler

The following modeling elements are used by UICompiler, a tool that currently generates UIs starting from separate models of control and of data. (A) *State Machines*, expressed as UML state diagrams, which specify the behavior of each active UI component (such as a window of a browser). (B) *Events*, that represent user actions, events that are endogenous to the UI, or events pushed by an external agent. (C) *Data views*, which represent the conceptual model of the UI; *e.g.*, for a mail front end application the domain model comprises classes like `Message`, `Thread`, `Folder`, etc. They are defined as a system of classes with appropriate relationships. (D) *Annotations*, or XML fragments that are "attached" to model elements (to states, transitions or data views) and which are used to: (1) specify which data views should be used in a state to display information, and which specific variables of the state machine are used to populate them; (2) what is the domain, in terms of data views, of the parameters used in parametric events; (3) which widgets have to be embedded into states, and which transitions are embedded within widgets; (4) which custom templates (of custom widgets) to include within the prototype being built; (5) what is the effect on data of actions associated to transitions or states. State machines, events and data views are specified following UML 2.4.

The basic postulate underlying UICompiler is that the structure of the UI in terms of widgets and containers, and its changes over time, are determined by the state machine model. Said in other terms, available actions shape the structure of the UI. Two assumptions are important to support such a postulate: (1) A state should be identified by available user actions and non-user events. In other words, the set of transitions leaving a state should distinguish that state from

other ones. (2) A state and its ancestors should be annotated in such a way that all the information needed by the user to decide which action to take next are rendered.

UICompiler embodies three general design principles: (A) The generated UI should include a representation of each active state, which should provide triggers for all the user actions that can be taken from that state. In this way each state represents an interaction context. (B) Nesting of states should reflect nesting of UI containers and widgets. In this way changes between interaction contexts depend on what actions are taken. (C) Concurrent regions should correspond to sibling components in the UI. This lets the user of the UI to work on any of the available interaction contexts, each corresponding to an active region

UICompiler does not attempt to produce visually appealing layouts and styles. Instead, the UI it generates can be easily styled with CSS (in the current desktop platform) thanks to classes and identifiers that are automatically injected into the DOM of the running UI. In this way, the designer can enhance at will aesthetics of the UI. Beyond providing flexibility to UI designers, in this way automated approaches to derive optimal layouts (*e.g.*, [5]) could be easily deployed.

UICompiler is a Java application that takes as input an XMI file that represents the state machine and class models; its content is parsed, several syntactic and semantic checks are applied (including checking that transitions leaving pseudo-states do not have event triggers, or that at most one event is specified for a compound transition), state machines are flattened, data views are transformed into JSON instances, and finally through a number of templates (for the top level state machine, for each state and transition) the final code is generated. At the moment only a desktop platform is supported, and code is generated in HTML, CSS and Javascript (jQuery).

3 Discussion

We are instrumenting UICompiler so that it can compute graph-theoretic metrics that bear upon usability (as done, for example, in [4]). In [1] we present preliminary results where the compiler uses the flattened version of the UML state machine model to compute the "betweenness" metric to identify states or transitions that are central in an interaction structure and that therefore should be free from usability defects.

So far we have applied UICompiler to a dozen of applications, and have found that all the UML state machine constructs were needed at one point or another. This suggests that the representation is minimal. We also discovered the following conceptual limitations.

Instances of Components. Right now it is not possible to specify, using a single UML state machine, that a model could include an arbitrary number of instances of a sub-state machine. For example, in catalog browser of an e-commerce website there could be n products, all sharing a particular interaction structure (such as expanding/collapsing, rotating, comparing). Right now one has to duplicate

a composite state n times, and n has to be known at modeling time. We plan to overcome this limitation by allowing the designer to dynamically instantiate UML classes that represent UI components, each with its own interaction structure. When doing that, however, static analysis of the model with graph-theoretic metrics would suffer, because the graph changes whenever these components are instantiated.

Toolbar Customizations. Another limit of UML state machines used for UI specification lies in the fact that models cannot specify that a user can customize a UI by changing its toolbars. This requires that new states are dynamically created, with new transitions. In other words, that the state machine model changes dynamically. We plan to overcome this limit again by the ability to dynamically instantiate new state machines.

Undo and Redo. At the moment these capabilities can be included in prototypes generated by UICompiler only if they are programmed within the delivery platform (*e.g.*, explicitly programmed in Javascript). Because UML state machines are based on finite state machines, the stack-oriented management of history required by undos and redos cannot be expressed. However, it would be nice if they could be specified explicitly in the models so that graph-theoretic metrics could consider them.

These limitations are not specific to UICompiler, as they apply to all the approaches based on finite state representations, including IFML. Another fundamental limit of all these approaches is the inability to represent "natural UIs", namely those where the user interacts through a continuous stream of events, such as when using an eye tracker, a body movement tracker, or remote controllers whose position and orientation can be tracked.

References

1. Brajnik, G., Harper, S.: Model-based engineering of user interfaces to support cognitive load estimation in automotive applications. In: Kun, A., Froelich, P. (eds.) Cognitive Load and In-Vehicle Human-Machine Interaction Workshop; adjuct Proceedings of the 5th Int. Conference on Automotive User Interfaces and Interactive Vehicular Applications. ACM Press, Eindhoven (October 2013)
2. OMG: Interaction flow modeling language (IFML), ftf - beta 1. Tech. rep., OMG (March 2013). http://www.omg.org/spec/IFML/1.0
3. Thimbleby, H.: Press on: principles of interaction programming. The MIT Press (2007)
4. Thimbleby, H., Oladimeji, P.: Social network analysis and interactive device design analysis. In: Proc. of Engineering Interactive Computing Systems 2009, pp. 91–100. ACM Press (2009)
5. Zeidler, C., Lutteroth, C., Weber, G.: Constraint solving for beautiful user interfaces: how solving strategies support layout aesthetics. In: CHI New Zealand 2012. ACM Press, Dunedin (2012)