Composite Event-Driven Programming
New Concepts for New Types of Interaction

Fredy Cuenca

School of Mathematical Sciences and Information Technology
Yachay Tech
San Miguel de Urcuquí, Ecuador
Email: fcuenca@yachaytech.edu.ec

***Abstract*—Implementing multi-touch and multi-modal systems requires splitting the code across several event handlers, which complicates programmers work. The present paper finds the root of this problem in the event-driven paradigm; more concretely, in the fact that event-driven languages lack abstractions for representing event sequences. It then suggests to augment event-driven languages so that programmers can have the possibility to define event sequences –herein called composite events– that can then be bound to event handlers. The main features of the composite event-driven language developed by the authors are outlined, as well as its benefits and problems. The paper suggests that, since its design, the event-driven paradigm was tailored for mouse-based interactions, and it may be important to question its suitability for implementing multi-touch and multi-modal interactions.**

Keywords–Multi-modal Systems; Multi-touch Systems; Interactive Systems; Event Languages; Composite Events.

I. INTRODUCTION

Many researchers agree that implementing (multi-)touch and multi-modal systems results in programs that are difficult-to-read and difficult-to-maintain [1][2][3]. In the domain of (multi-)touch systems, even simple gestures, such as the *pinch-to-zoom* gesture, require the system to handle a stream of *touch-down*, *touch-move*, and *touch-up* events, from within the intention of the user to enlarge a particular region of the touchscreen has to be unveiled. Similarly, multi-modal commands, like speech-and-pointing commands, will be perceived by multi-modal systems as a series of speech events and pointing events. Thus, such multi-modal systems have the difficult task of having to continuously identify which speech events and pointing events are part of the same command.

The implementation of those interactions that are reflected, in the system, as sequences of interrelated events, forces programmers to litter their code with a multitude of flags and global variables that have to be updated across different event handlers in order to keep track of the event sequences. The resulting difficult-to-read, difficult-to-maintain "callback-soup" [1][2] is not a consequence of bad programming habits or poor comprehension of event-driven principles. Rather, it is accidental complexity: complexity caused by the languages and tools chosen for programming [4]. This type of complexity can only be reduced by selecting or developing better programming languages and tools [4]. The present work intends to shed some light on how to develop better languages and tools.

We believe that the appearance of the "callback-soup" is largely due to the fact that event-driven languages, which are widely used to implement interactive systems [5][6], only offer abstractions, called events, for representing simple user actions, such as a touch-down or a speech input; but these languages do not offer abstractions for representing sequences of user actions.

This paper proposes a programming model that enables programmers to compose events. In the proposed model, there is an abstraction called composite event, this being a programmer-defined event sequence. Composite events are defined by connecting primitive events, such as touch events or speech inputs, through a set of operators, where each operator represents a temporal, spatial, or semantic relation among their operands. Composite events can then be bound to event handlers, callback functions that implement the system's runtime behavior.

For instance, two basic interactions with a touchscreen photo viewer can be described by binding the composite event *touch-flicking-left* to the event handler `ShowNextPhoto()`, and the speak-and-touch *remove-this* event to the event handler `RemovePhoto()`. The two aforementioned composite events would be defined by the programmer as a combination of touch events (former case) or as a mix of touch events and speech events (latter case).

In the proposed programming model, at runtime, the event handlers are to be launched every time their associated composite events are automatically detected by a composite event-driven tool. This model will save programmers from having to implement a supervisory mechanism for tracking event sequences; such a mechanism would be incorporated in the composite event-driven tool to be exploited by programmers. By delegating the detection of event sequences to the composite event-driven tool, programmers can clean their source codes of the flags and global variables that were necessary for this task when using event-driven languages.

The remainder of this paper proceeds as follows: In Section 2, the proposed approach is compared against others that also intend to ease the creation of multi-modal/multi-touch interaction. Section 3 outlines the main features, gains, and limitations of a composite event-driven tool that was implemented as part of a PhD project. Finally, Section 4 argues in favor of augmenting event-driven languages with composite events.

II. RELATED WORK

The benefits of using composite events for rapid prototyping of multi-modal systems have already been highlighted

 ISBN: 978-1-61208-535-7

[3][7]. Additionally, this paper reports similar gains when prototyping (multi-)touch gestures and, most importantly, proposes composite events as a unified solution to the "callback soup" problem that infects both multi-modal and multi-touch interactive systems.

Other (mostly visual) languages have also been proposed with the aims of easing the "callback soup" problem. One important contrast is that while our composite event-driven model allows describing interactions in terms of events and event binding, other existing languages require concepts (e.g., Petri nets and block diagrams) and programming practices (e.g., depicting visual models) that may be unfamiliar to programmers of interactive systems. As the rankings of programming popularity published by IEEE [6] and TIOBE [5] attest, programmers are more accustomed to textual, event-driven languages. Given that familiarity with a language has a strong, positive influence on programming language adoption (even stronger than intrinsic properties of the language, such as performance, reliability, and simple semantics) [8], the proposed programming language retains the textual and event-driven nature of mainstream programming languages. Other more concrete differences of our approach and existing languages can be found below.

A. Multi-modal interaction description languages

In Squidy [9], multi-modal interactions are represented as block diagrams that programmers can use to channel and transform the data coming from different input modalities to the application. One issue of this model is that each modality has its own independent channel. Data from different modalities must be collected in the application as the human-machine interaction occurs. With our approach, a composite event can be defined by combining events from the same or different modalities. The data carried by these events is stored in parameters that arrive to the application all at once –no need for queuing events in the application.

Similar to our model, SMUIML [10] allows composing events so that each composite event can be bound to one event handler. At runtime, these handlers are launched once their associated composite events have occurred. We generalize this approach by allowing programmers to attach event handlers to very specific stages of a composite event —in our model, event binding has a time component. This allows launching several event handlers at different moments of the lifecycle of a composite event, which makes it possible to provide the end user with partial feedback.

ICO [11] is a formal language for modeling both multi-modal and multi-touch interactions. It has an underlying mathematical apparatus that allows predicting properties of the interaction in static time, without having to run the ICO model. One of its drawbacks is that Petri nets were not tailored for modeling interaction, thus ICO models do not map close to the problem domain. ICO users have to tweak their interaction models to fit them into a Petri net. Our approach, instead, makes use of a domain-specific notation that has designated symbols for representing time constraints among events and special keywords for specifying modalities.

B. (Multi-) touch gesture definition languages

Besides the already reported gains experienced when prototyping multi-modal systems [3][7], composite events also bring about advantages over existing, salient gesture definition languages, such as GDL [12], Proton [1] and GestIT [2]. To a greater or lesser extent, all these languages have proven to ease the description of touch gestures: programmers can define gestures in a declarative fashion without having to write fine-grained code for tracking the gesture state.

GDL [12] is intended for simple description of touch gestures that can be used across multiple hardware platforms. A touch gesture is defined as a set of rules that must be met by the raw touch data along with the value(s) to be returned when the gesture is detected. GDL allows defining multi-stroke gestures (e.g., a cross) as long as the strokes can be issued sequentially. Our language, in contrast, allows defining gestures involving both sequential and parallel strokes (e.g., simultaneous vertical flicks). Furthermore, unlike the proposed language, GDL does not allow specifying temporal constraints.

On the other hand, Proton allows users to represent gesture interaction as tablatures. A user study proved that tablatures are easier-to-comprehend than event-callback code [13]. The expressiveness of Proton was shown by implementing multi-user touch-based applications like the classic Pong game, a tennis-like game between two opponents [13]. One issue with Proton is its lack of time variables, which makes it difficult to calculate the duration of a gesture, for instance. In contrast, our approach allows defining and maintaining different types of variables (including time variables) throughout the composite event lifecycle.

As to the Petri nets-based language GestIT, partial feedback is only possible by decomposing gestures definitions into smaller, sub-gestures definitions. This is because GestIT only launches event handlers at the end of a (sub-)gesture. Our approach does not force programmers to make such decompositions because multiple event handlers can be bound to different stages of one single gesture. Furthermore, Proton and GestIT do not include timeout events, which make it unnecessarily complex to define recursive gestures. For instance, the single, double, and triple tap require three separate definitions with Proton and GestIT. In our approach, timeout events exist and can be connected with any input event (e.g., touch events or speech inputs) to be part of a composite event. By using timeout events, our approach makes it possible to describe the three aforementioned gestures with one single definition: a sequence of N taps that end after a period of "silence".

C. Languages for reactive systems

It should be made clear that the goal of the proposed approach differs from that of languages such as P [14], Esterel [15], or Lustre [16].

P is oriented more towards the development of distributed systems. Therefore, the type of interaction to be modeled with P is among the components of the intended system. Such a component-component interaction is uncoordinated and consists of messages sent from different sources. In contrast, the proposed language is designed for human-machine interaction, a type of interaction where the inputs are coordinately issued by one single agent, the end user. Due to this fundamental difference, P focuses on forcing asynchronous events to be responded within a reasonable timeframe. For this, P includes notations for explicit declaration of event deferrals. Our language, instead, focuses on describing relations between user actions and system responses, which can be done concisely with the proposed composite event binding notations.

 ISBN: 978-1-61208-535-7

Similar contrasts can be found against Esterel or Lustre, which are intended to develop real-time, embedded systems. Thus, these are much closer to P than to the proposed language.

III. A FIRST COMPOSITE EVENT-DRIVEN TOOL

In the context of a PhD research, a composite event-driven language along with its supporting tool was developed and evaluated for rapid prototyping of multi-modal systems [3][17][7].

A. Automatic detection of composite events

A composite event-driven tool must be in charge of tracking every programmer-defined composite event and launching its associated event handler(s) in a timely manner. In our particular implementation, every composite event is internally represented as a finite state machine [3]. The human-machine interaction is described with a textual notation, as a mapping of composite events to event handlers, and, under the hood, the proposed tool generates a set of finite state machines by means of specialized algorithms [3]. As the constituent events of a composite event occur, in the specified order, its reciprocal finite state machine switches to different states. Finally, the end-node of this machine is reached when its reciprocal composite event has occurred. Programmers can attach event handlers to every node or link of a finite state machine when writing event binding code [17]. Given that a composite event can be reused in the definition of other, more complex composite events, our finite state machines are hierarchical.

B. Experimental results

This language was compared against C#, a mainstream event-driven language, by means of a within-subjects experiment. A user study involving twelve participants (experienced developers) was conducted to compare programming efficiency. After modifying an interaction model with both the composite event-driven language and the baseline language, it was revealed that the former leads to higher completion rates, lower completion times, and less code testing [7]. Another study with non-developers is being conducted to measure whether the proposed language is simple enough to be understood and used as a discussion tool within multidisciplinary teams (e.g., in a robotics project). We have not yet conducted experiments about tool performance (e.g., recognition rate or recognition speed of composite events). For now, our focus is on evaluating the feasibility and efficacy of composite event-driven programming rather than the efficiency of our particular implementation.

C. Expressiveness

The expressiveness of the proposed language has already been evaluated by implementing a variety of interactions involving mouse gestures, keystrokes, and speech inputs [3][17]. Later, as part of a PhD research, we implemented a proof-of-concept application that recognizes hand gestures, body movements, and a variety of touch gestures (e.g., single-stroke, multi-stroke, free-form, and multi-touch). More recently, composite events have also been applied in the field of robotics [18]. The size of the developed applications is small: we always used less than 30 composite events in our applications. We still do not have indications, neither in favor nor against, of whether the easiness-to-maintain will increase linearly with the size of the applications or not.

D. Limitations

The current version of the proposed composite event-driven language does not include general-purpose constructs (e.g., for's and if's). It is a declarative language that includes notations for describing human-machine interaction as mappings of user actions to system responses or, more concretely, as mappings of composite events to event handlers. But the fine-grained code required to implement the event handlers and the graphical user interface has to be written with a general-purpose language, as part of a canned application that has to be imported into our composite event-driven tool. The separation of interaction code from application code brought about many problems, such as the inability to create autonomous executable files (e.g., the imported application is developed with a general-purpose language whose syntax is unknown to the developed composite event-driven tool), the excessive amount of function calls required to exchange data between the external application and the composite event-driven tool (e.g., application variables have to be maintained by calling functions; these variables cannot be set directly from the composite event-driven language), and the difficulty to debug a program that is broken into two separate, independent pieces (e.g., composite event variables are traced within the proposed tool whereas external application variables are traced with external tools), among others.

IV. DISCUSSION

The proposed shift up from events to composite events would be the reflection of a fundamental change in human-computer interaction: in the past, systems were mainly commanded by simple user actions such as clicks on widgets; but modern multi-modal/multi-touch systems are intended to be commanded by several coordinated user actions, such as pointing-and-speech. These sets of coordinated actions would be abstracted as composite events in the proposed model.

It is true that existing event-driven languages have enough expressiveness to develop multi-modal and multi-touch interactive systems, but the complexity of the resulting code can be reduced by using composite events: the interaction state that has to be updated manually with event-driven languages is maintained automatically by composite event-driven tools. This benefit was noticed by twelve participants of a comparative user study: All of them agreed that the task of modifying interaction descriptions was simpler when using composite events than when using C#, a mainstream event language [7].

The event-driven paradigm was inspired in academic research (e.g., University of Alberta UIMS and Sassafras) carried out in the 80's [19], when mouse-based interactions were predominant. By 2000, the mouse-based interactions introduced by the Apple Macintosh, in 1984, had been widely adopted by almost all applications [19]. Mainstream event-driven languages such as Visual Basic and Java were released within that period; more concretely, in 1991 and 1995, respectively. Therefore, it should not be a surprise to realize that the event-driven paradigm with all its underlying concepts was likely tailored to deal with a type of interaction that is much simpler than multi-touch and multi-modal interaction, which now claim silently their own paradigm and tooling.

In a seminal paper, published in 2000, when discussing event-driven languages, Myers et al. [19] foretold that, in order to deal with the then-emerging modalities, such as touch and

 ISBN: 978-1-61208-535-7

speech, a new paradigm may be needed. To the best of our knowledge, no one has yet given a clue about how to start building the new programming paradigm. Based on 4+ years of research, this paper is suggesting one direction: to extend the fundamental concept of event to composite events.

We expect that the first, positive results obtained after implementing our programming model can encourage other researchers and practitioners to create more full-fledged composite event-driven tools, which, aside from including code editors, runtime environment, and debugging tools, like our tool, must also include interface builders and a language enriched with general-purpose constructs. With such a set of tools, programmers will no longer need to separate application code from interaction code and, thus, the aforementioned limitations might disappear.

ACKNOWLEDGEMENTS

We would like to acknowledge the effort of our former colleagues of Hasselt Universiteit, namely, Jan Van der Bergh, Kris Luyten, and Karin Coninx, for helping us implement a first version of the vision exposed in this paper.

REFERENCES

[1] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala, "Proton: multitouch gestures as regular expressions," in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'12), 2012.

[2] L. Spano, A. Cisternino, F. Paterno, and G. Fenu, "Gestit: a declarative and compositional framework for multiplatform gesture definition," in Proceedings of the EICS'13. ACM, 2013.

[3] F. Cuenca, J. Van den Bergh, K. Luyten, and K. Coninx, "A domain-specific textual language for rapid prototyping of multimodal interactive systems," in Proceedings of EICS'14. ACM, 2014.

[4] C. Scholliers, L. Hoste, B. Signer, and W. De Meuter, "Midas: a declarative multi-touch interaction framework," in Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction. ACM, 2011.

[5] "TIOBE Index," http://www.tiobe.com/tiobe-index/, 2016, [Online; accessed 21-December-2016].

[6] "IEEE Spectrum," http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages/, 2016, [Online; accessed 21-December-2016].

[7] F. Cuenca, J. V. d. Bergh, K. Luyten, and K. Coninx, "A user study for comparing the programming efficiency of modifying executable multimodal interaction descriptions: a domain-specific language versus equivalent event-callback code," in Proceedings of the PLATEAU'15. ACM, 2015.

[8] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," ACM SIGPLAN Notices, vol. 48, no. 10, 2013.

[9] W. König, R. Rädle, and H. Reiterer, "Interactive design of multimodal user interfaces," Journal on Multimodal User Interfaces, vol. 3, no. 3, 2010.

[10] B. Dumas, D. Lalanne, and R. Ingold, "Description Languages for Multimodal Interaction: A Set of Guidelines and its Illustration with SMUIML," Journal of multimodal user interfaces, vol. 3, no. 3, 2010.

[11] D. Navarre, P. Palanque, J.-F. Ladry, and E. Barboni, "ICOs: A Model-Based User Interface Description Technique dedicated to Interactive Systems Addressing Usability, Reliability and Scalability," ACM Transactions on Computer-Human Interaction, vol. 16, no. 4, 2009.

[12] S. H. Khandkar and F. Maurer, "A domain specific language to define gestures for multi-touch applications," in Proceedings of the 10th Workshop on Domain-Specific Modeling. ACM, 2010.

[13] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala, "Proton++: a customizable declarative multitouch framework," in Proceedings of the 25th annual ACM symposium on User interface software and technology. ACM, 2012.

[14] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, "P: safe asynchronous event-driven programming," ACM SIGPLAN Notices, vol. 48, no. 6, 2013.

[15] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," Science of computer programming, vol. 19, no. 2, 1992.

[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," Proceedings of the IEEE, vol. 79, no. 9, 1991.

[17] F. Cuenca, J. Van den Bergh, K. Luyten, and K. Coninx, "Hasselt uims: a tool for describing multimodal interactions with composite events," in Proceedings of the EICS'15. ACM, 2015.

[18] J. Van den Bergh, F. Cuenca Lucero, K. Coninx, and K. Luyten, "Toward specifying human-robot collaboration with composite events," 2016.

[19] B. Myers, S. E. Hudson, and R. Pausch, "Past, present, and future of user interface software tools," ACM Transactions on Computer-Human Interaction (TOCHI), vol. 7, no. 1, 2000.

 ISBN: 978-1-61208-535-7